ANALYSIS AND IMPLEMENTATION CONSIDERATIONS OF KRYLOV SUBSPACE METHODS ON MODERN HETEROGENEOUS COMPUTING ARCHITECTURES

A Dissertation
Submitted to
the Temple University Graduate Board

---

in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

---

by
Andrew James Higgins
May 2024

ABSTRACT

Krylov subspace methods are the state-of-the-art iterative algorithms for solving large, sparse systems of equations, which are ubiquitous throughout scientific computing. Even with Krylov methods, these problems are often infeasible to solve on standard workstation computers and must be solved instead on supercomputers. Most modern supercomputers fall into the category of "heterogeneous architectures", typically meaning a combination of CPU and GPU processors. Thus, development and analysis of Krylov subspace methods on these heterogeneous architectures is of fundamental importance to modern scientific computing.

This dissertation focuses on how this relates to several specific problems. The first analyzes the performance of block GMRES (BGMRES) compared to GMRES for linear systems with multiple right hand sides (RHS) on both CPUs and GPUs, and modelling when BGMRES is most advantageous over GMRES on the GPU. On CPUs, the current paradigm is that if one wishes to solve a system of equations with multiple RHS, BGMRES can indeed outperform GMRES, but not always. Our original goal was to see if there are some cases for which BGMRES is slower in execution time on the CPU than GMRES on the CPU, while on the GPU, the reverse holds. This is true, and we generally observe much faster execution times and larger improvements in the case of BGMRES on the GPU. We also observe that for any fixed matrix, when the number of RHS increase, there is a point in which the improvements start to decrease and eventually any advantage of the (unrestarted) block method is lost. We present a new computational model which helps us explain why this is so. The significance of this analysis is that it first demonstrates increased potential of block Krylov methods on heterogeneous architectures than on previously studied CPU-only machines. Moreover, the theoretical runtime model can be used to identify an optimal partitioning strategy of the RHS for solving systems with many RHS.

The second problem studies the s-step GMRES method, which is an implementation of GMRES that attains high performance on modern heterogeneous machines by generating s Krylov basis vectors per iteration, and then orthogonalizing the vectors in a block-wise fashion. The use of s-step GMRES is currently limited because the algorithm is prone to numerical instabilities, partially due to breakdowns in a tall-and-skinny QR subroutine. Further, a conservatively small step size must be used in practice, limiting the algorithm's performance. To address these issues, first a novel randomized tall-and-skinny QR factorization is presented that is significantly more stable than the current practical algorithms without sacrificing performance on GPUs. Then, a novel two-stage block orthogonalization scheme is introduced that significantly improves the performance of the s-step GMRES algorithm when small step sizes are used. These contributions help make s-step GMRES a more practical method in heterogeneous, and therefore exascale, environments.

In memory of Sunny, Agnes, Helga, and Willi

ACKNOWLEDGEMENTS

One decade at Temple and one global pandemic later, I understand my journey up to this point would have certainly been impossible without the help of many people that I have been extremely fortunate to have in my life.

I have grown significantly and learned the way of the academic world under the guidance of my advisor Daniel Szyld. He taught me how to transition from simply a student to an independent researcher. His encouragement to expand my viewpoint outside the scope of the pure mathematics of my work has given me the necessary perspective and drive to become a balanced researcher in both the numerical linear algebra and high-performance computing communities. Additionally, I cannot thank him enough for his everlasting patience throughout my years under his supervision, and the opportunities he has given me over the years that I would not have without him. Thanks to him, I feel prepared for a fruitful research career.

I am extremely fortunate to have been hosted and mentored by Erik Boman and Ichi Yamazaki at Sandia National Laboratories in Albuquerque, NM for two summers. I am very grateful to have made and maintained these strong relationships over the years, and have learned a great deal about high-performance scientific computing from them. Working with them has helped me understand practical issues of high-performance numerical linear algebra that I would never have encountered in the classroom, and has enabled me to develop competitive algorithms in high-performance settings.

I would like to thank Professors Benjamin Seibold and Gillian Queisser at Temple University for their guidance over the years. Between advice they have given outside the classroom and incredible instruction on a wide range of topics in applied mathematics, they have emphasized maintaining the perspective of how numerical methods are used in the real world while still understanding their technicalities.

For being there for me over the years and helping me maintain a few remaining threads of sanity, I would like to thank Tim, Cait, Matt, Bryan, Peter, John, Sean, Benji, Elana, and Cody. I cannot thank my parents Tim and Frauke, along with Joan and Curt Beckett enough for their support both through my good and bad times. I am extremely lucky to have them in my support network.

Finally, I thank Lauren, for her empathy, support, patience, and love over the years. You gave me something to look forward to every day, and for this I am forever grateful.

All of the work on this thesis was conducted in collaboration with Daniel B. Szyld and Erik G. Boman, while the work discussed in Chapters 5– 6 was also done in collaboration with Ichitaro Yamazaki.

The research funding for this thesis came from several sources. Part of the work on this thesis was conducted at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This work was in part supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

The remainder of the thesis was completed at Temple University, which includes calculations carried out on HPC resources at Temple University supported in part by the National Science Foundation through major research instrumentation grant number 1625061 and by the US Army Research Laboratory under contract number W911NF-16-2-0189. Finally, all the research was integrated into this thesis under the Dissertation Completion Grant awarded by Temple University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

At the core of scientific computing, solving a system of linear equations of the form $Ax = b$ is imperative, where $A \in \mathbb{C}^{n \times n}$. For example, within physics, chemistry, biological, or engineering simulations, such systems often arise from the discretization of an ordinary differential equation (ODE) or partial differential equation (PDE) describing the physical phenomena via a finite difference (FD) or finite element (FE) method. The specific systems that arise from such discretizations are often *sparse*, meaning that the coefficient matrix A consists of mostly zero entries. More specifically, let nz be the number of non-zero entries of A. We will consider A *sparse* provided $nz \ll n^2$. Such matrices can be stored in a efficient manner, where only the non-zero entries and their respective locations are stored.

While classical direct methods like Gaussian elimination for solving such systems exist, applications such as kinetic equations and computational fluid dynamics problems give rise to systems large enough to render such methods infeasible even on large, distributed heterogeneous computers. Instead, the state-of-the-art methods for solving large, sparse systems in high performance environments are Krylov subspace methods.

1.1 Outline and Structure

In Chapter 2, necessary background on Krylov subspace methods with a focus on the GMRES algorithm is introduced, and a discussion of the key concepts and considerations for their implementation on modern heterogeneous machines is given.

Thereafter, in Chapter 3, we introduce and analyze an existing variant of GMRES designed to solve systems of equations with multiple right hand sides (RHS), called block GMRES (BGMRES). It is well known that if one wishes to solve a system of equations with s RHS, BGMRES can indeed outperform GMRES on each of the s RHS separately ($s \times$ GMRES), but not always [61, 62, 73]. Our original goal was to see if this paradigm changes when we implement these algorithms on graphics processing units (GPUs). In particular, we wanted to see if there are some cases for which BGMRES is slower in execution time on the central processing unit (CPU) than $s \times$ GMRES on the CPU, while on the GPUs, the reverse holds. This held true, and we generally observed much faster execution times and larger improvements in the case of BGMRES. We also observed that for any fixed matrix, when the number of RHS increase, there is a point in which the improvements start to decrease and eventually any advantage of the (unrestarted) block method is lost.

The reason for this latter observation is the higher data movement cost of $s \times$ GMRES weighed against the higher cost of the floating point operations in the BGMRES, as required by the block Arnoldi procedure and the QR factorizations needed to orthogonalize the block basis of the block Krylov subspace. The balance between a richer block Krylov subspace and the increased computational cost of BGMRES was first mentioned by Vital when she proposed the algorithm [73].

In this analysis, our contribution consists of first confirming that on GPUs BGMRES is not only faster, but is more advantageous than $s \times$ GMRES in most practical cases. Second, we observed that there is an optimal number of RHS where BGMRES takes most advantage of the GPU architecture. Finally, we present a new computational model which helps us explain why this is so. The significance of this analysis is that it first demonstrates increased potential of block Krylov methods on heterogeneous architectures than on previously studied CPU-only machines. Moreover, the theoretical runtime model can be used to identify an optimal partitioning strategy of the RHS for solving systems with many RHS.

In Chapter 4, we introduce the s-step GMRES algorithm, which is an existing variant of GMRES designed to attain higher performance on distributed machines than standard GMRES. We will discuss the difficulties of existing implementations of s-step GMRES, primarily related to the block orthogonalization procedure it uses.

We address one of these difficulties in Chapter 5, where we introduce a novel randomized method to improve the stability of a tall-and-skinny QR factorization used within the block orthogonalization procedure without sacrificing performance. This consists of proof-of-concept of the algorithm's design to justify why it should be expected to have similar (or better) performance compared to the current state-of-the-art tall-and-skinny QR algorithm, along with rigorous mathematical roundoff error analysis to prove that it is significantly more stable than the current state-of-the-art algorithm with high probability. Finally, we analyze our algorithm's performance against its competitors on a modern high-performance GPU, and demonstrate that its performance in practice aligns with our expectations from the algorithm's design, showing its performance is nearly as good, and sometimes even better, than the current state-of-the-art scheme. The significance of this contribution is that we develop a high performance tall-and-skinny QR algorithm, and show that it is significantly more stable than the current state-of-the-art without sacrificing performance, both in theory and practice.

Then, we discuss a novel scheme to further improve the performance of the rest of the block orthogonalization in Chapter 6. In order to improve the performance of block orthogonalization using a small step size, we introduced a two-stage algorithm, which pre-process the s basis vectors at a time to maintain the well-conditioning of the basis vectors but delay the orthogonalization until enough basis vectors are generated to obtain higher performance. We presented numerical and performance results to demonstrate its potential.

The significance of the work in Chapters 5–6 is that improving stability of the tall-and-skinny QR factorization would help stabilize s-step GMRES, while improving the performance of the block orthogonalization for small block sizes boosts the method's performance, both of which are contributions that improve the practicality of s-step GMRES, and make it an attractive alternative to GMRES in exascale environments. Additionally, this work makes advances in the cutting edge area of randomized Krylov methods, which have only very recently been studied in the past couple of years.

CHAPTER 2
BACKGROUND ON KRYLOV METHODS, HETEROGENEOUS COMPUTERS

This section begins with a breakdown of convenient notation that will be used throughout the thesis. Next, background on Krylov methods from a high-level and GMRES from a detailed perspective are given. Additionally, the strengths and weaknesses of modern heterogeneous machines are discussed both within and outside the context of Krylov methods.

2.1 Notation

We first establish a few basic notational conventions in Table 2.1, to separate matrices, vectors, and vector spaces, along with a way to distinguish block-structured matrices, which will appear frequently throughout this thesis, from generic unstructured ones.

typeset	**meaning**	**examples**
upper-case letters	matrices with non-specificed structure	A, V
lower-case letters	vectors	x, b
bold upper-case letters	matrices with block structure	$\boldsymbol{V}$
caligraphy	vector space	$\mathcal{V}$, $\mathcal{K}$

Table 2.1: Symbol typeset used throughout the thesis

Notation	**Description**
v_k	k^{th} column of the matrix V
$h_{j,k}$	scalar at row j and column k of the matrix H
$V_{k:m} = [v_k, \dots, v_m]$	sub-matrix of V with columns between k and m
$V_m = V_{1:m} = [v_1, \dots, v_m]$	sub-matrix of V up to column m
$H_{j:k,p:q}$	sub-matrix of H using rows j–k and columns p–q
$H_{j,p} = H_{1:j,1:p}$	sub-matrix of H up to row j and column p
$V_{:,p:q}$	sub-matrix of V including all rows and columns p–q
$V_{j:k,:}$	sub-matrix of V including rows j–k and all columns
$\boldsymbol{V}_k$	k^{th} block of the block matrix $\boldsymbol{V}$
$\boldsymbol{V}_{j:k}$	blocks j–k of the block matrix $\boldsymbol{V}$
$\boldsymbol{H}_{j,k}$	vertical blocks j and horizontal block k of block matrix $\boldsymbol{H}$
$\boldsymbol{H}_{j:k,p:q}$	vertical blocks j–k and horizontal blocks p–q of block matrix $\boldsymbol{H}$
$v_j^{(k)}$	j^{th} column of the k^{th} block of block matrix $\boldsymbol{V}$
$v_{p,q}^{(k)}$	entry (p, q) of the k^{th} block of block matrix $\boldsymbol{V}$
$h_{j,k}^{(p,q)}$	entry (j, k) of the $(p, q)^{th}$ block of block matrix $\boldsymbol{H}$
$\sigma_k(V)$	k^{th} largest singular value of the matrix V
$\kappa(V)$	ℓ_2 condition number of V
Π	vector space of polynomials (infinite degree)
$\mathbf{u}$	unit roundoff

Table 2.2: Matrix and other common linear algebra notation used throughout the thesis

In Table 2.2, we define some specific notations for matrices and other common spaces/concepts used heavily in numerical linear algebra. For matrices or block matrices with a single subscript, it is assumed the only subscript corresponds to some indexing of the column(s) of the matrices. In the case of two sets of indices specified, separated by a comma, the first set of indices corresponds to the row(s) while the second corresponds to the column(s).

At first glance, the notational differences for the block matrices in Table 2.2 are daunting. To disambiguate them, I provide a few diagrams (Figures 2.1–2.2) representing what each of the block matrix notations mean. Suppose we have two different block-structured matrices $\boldsymbol{V}$ and $\boldsymbol{H}$. First, assume $\boldsymbol{V} \in \mathbb{C}^{n \times m \cdot s}$ has a column-wise block structure, containing m blocks of s columns each. Then $\boldsymbol{V}$ can be described using Figure 2.1. In contrast, assume $\boldsymbol{H} \in \mathbb{C}^{n \cdot s \times m \cdot s}$ has a block-wise structure both in its rows and columns, consisting of n row blocks and m column blocks of width s in both directions. In other words, $\boldsymbol{H}$ has a block structure partitioned into $s \times s$ matrices. Then $\boldsymbol{H}$ can be described using Figure 2.2.

$$\boldsymbol{V} = \Bigg[\underbrace{\begin{matrix} | & | & & | \\ v_1^{(1)} & v_2^{(1)} & \dots & v_s^{(1)} \\ | & | & & | \end{matrix}}_{\boldsymbol{V}_1} \Bigg| \underbrace{\begin{matrix} | & | & & | \\ v_1^{(2)} & v_2^{(2)} & \dots & v_s^{(2)} \\ | & | & & | \end{matrix}}_{\boldsymbol{V}_2} \Bigg| \cdots \Bigg| \underbrace{\begin{matrix} | & | & & | \\ v_1^{(m)} & v_2^{(m)} & \dots & v_s^{(m)} \\ | & | & & | \end{matrix}}_{\boldsymbol{V}_m} \Bigg]$$

$$= \left[\, \boldsymbol{V}_1 \,\middle|\, \boldsymbol{V}_2 \,\middle|\, \dots \,\middle|\, \boldsymbol{V}_m \,\right]$$

Figure 2.1: Notational schematic for matrix with column-wise block structure

$$\boldsymbol{H} = \left[\begin{array}{c|c|c|c} \boldsymbol{H}_{1,1} & \boldsymbol{H}_{1,2} & \dots & \boldsymbol{H}_{1,m} \\ \hline \boldsymbol{H}_{2,1} & \boldsymbol{H}_{2,2} & \dots & \boldsymbol{H}_{2,m} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline \boldsymbol{H}_{n,1} & \boldsymbol{H}_{n,2} & \dots & \boldsymbol{H}_{n,m} \end{array}\right],$$

$$\boldsymbol{H}_{j,k} = \begin{bmatrix} h_{1,1}^{(j,k)} & h_{1,2}^{(j,k)} & \dots & h_{1,s}^{(j,k)} \\ h_{2,1}^{(j,k)} & h_{2,2}^{(j,k)} & \dots & h_{2,s}^{(j,k)} \\ \vdots & \vdots & \ddots & \vdots \\ h_{s,1}^{(j,k)} & h_{s,2}^{(j,k)} & \dots & h_{s,s}^{(j,k)} \end{bmatrix}$$

Figure 2.2: Notational schematic for matrix with generalized block structure

2.2 Krylov Subspace Methods

Krylov subspace methods are a broad class of iterative schemes for solving a linear system. While many Krylov subspace methods exist (exceptional overviews are given by Liesen and Strakoš [42], and by Saad [58]), the high-level premise of most Krylov subspace methods is to find an approximate solution to the linear system by:

1. building successively larger Krylov subspaces at each iterate,
2. orthogonalizing the vectors within the Krylov subspace with respect to some inner product(s), and
3. finding a "good" approximate solution within the subspace.

Before going into further details, we first define Krylov subspaces and provide their subspace nesting property.

Definition 2.1. *Let $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$.* Krylov subspaces *generated by A and b, denoted by $\mathcal{K}_m(A,b)$, are defined as*

$$\mathcal{K}_m(A,b) = \mathit{span}\{b, Ab, \ldots, A^{m-1}b\}.$$

Proposition 2.2 (Nested Subspace Property)**.** *Given $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$, and a positive integer m,*

$$\mathcal{K}_m(A,b) \subset \mathcal{K}_{m+1}(A,b).$$

Provided A is non-singular, Krylov subspaces are of mathematical interest, because it is straightforward to show that by the Caley-Hamilton theorem [4], there exist constants $c_0, \ldots, c_{n-1} \in \mathbb{C}$ so that $A^{-1}b = \sum_{k=1}^{n} -\frac{c_k}{c_0}A^{k-1}b \in \mathcal{K}_n(A,b)$. By definition, $\mathcal{K}_n(A,b)$ is formed by the span of at most n vectors, and therefore the true solution lies in a Krylov subspace of dimension at most n. We next define two concepts to more precisely define the dimension of the Krylov subspace containing the true solution.

Definition 2.3 (Grade of a vector). *Let $A \in \mathbb{C}^{n\times n}, v \in \mathbb{C}^n$. The* grade *of v is the degree of the minimum polynomial of v with respect to A. Specifically,*

$$\text{grade}(v) = \min_{q\in\Pi}\{\deg(q) : q(A)v = 0\}.$$

Proposition 2.4 (Dimension of a Krylov Subspace). *Let $A \in \mathbb{C}^{n\times n}, b \in \mathbb{C}^n$. Then $\mathcal{K}_{\text{grade}(b)}(A, b)$ is invariant under A, and therefore*

$$\mathcal{K}_m(A, b) = \mathcal{K}_{\text{grade}(b)}(A, b) \quad \forall m \geq \text{grade}(b), \tag{2.1}$$

$$\dim(\mathcal{K}_m(A, b)) = \min\{m, \text{grade}(b)\}. \tag{2.2}$$

The proof for Proposition 2.4 can be found in [58], though it is a simple consequence of Proposition 2.2 and the definition of the grade. Additionally, the Caley-Hamilton theorem can be used again to prove that grade$(b) \leq n$ for any $b \in \mathbb{C}^n$.

Therefore the true solution should be recovered by the Krylov subspace method in grade$(b) \leq n$, iterations, provided the method is designed so that some "good" approximation from the Krylov subspace is chosen at each iteration. The specific way the approximation is chosen–which is typically done so that the approximate satisfies some set of attractive properties, such as an error or residual minimization property with respect to some norm–is what defines the specific Krylov subspace method.

2.3 Background on GMRES

The majority of this thesis will be spent on variants of the Generalized Minimum Residual (GMRES) algorithm, which is a specific Krylov subspace method from Saad and Schultz [59] which chooses its approximate solution at each iteration by minimizing the ℓ_2 norm of the residual over all vectors in the Krylov subspace. Pseudocode for a typical backwards stable GMRES implementation is given in Algorithm 1 [54].

Algorithm 1 m iterations of GMRES with MGS Orthogonalization

```
    Input:   Right hand side $b \in \mathbb{R}^n$, initial guess $x_0 \in \mathbb{R}^n$
    Output: Approximate solution $x_m \in \mathbb{R}^n$
 1: $r_0 = b - Ax_0$
 2: $\beta = \|r_0\|_2, v_1 = r_0/\beta$
 3: for $j = 1, \dots, m$ do
 4:     $w = Av_j$
 5:     for $i = 1, \dots, j$ do
 6:         $h_{i,j} = w^* v_i$
 7:         $w = w - h_{i,j} v_i$
 8:     end for
 9:     $h_{j+1,j} = \|w\|_2$, $v_{j+1} = w/h_{j+1,j}$
10: end for
11: $y_m = \arg\min_{y \in \mathbb{R}^m} \|r_0 - AV_m y\|_2$
12: $x_m = x_0 + V_m y_m$
```

The GMRES algorithm begins with some initial guess x_0 and the right hand side (RHS) b, to form the initial residual $r_0 = b - Ax_0$ in step 1. The initial residual is then normalized, and stored as the first basis vector v_1 for the Krylov subspace

$\mathcal{K}_m(A, r_0)$. In lines 3-10, the algorithm executes its main loop where it performs the *Arnoldi* process, which is simply the process of forming the rest of the Krylov basis vectors for the space $\mathcal{K}_m(A, r_0)$ by performing a sparse matrix-vector multiplication of A to the previous Krylov basis vector and then orthogonalizing the new vector against the previous ones.

Remark 2.5. Typically, whenever a sparse matrix-vector multiplication is performed in Krylov methods, it is also done with a preconditioning step to improve convergence and/or stability of the solver on systems with an ill-conditioned coefficient matrix A. Throughout this thesis, it is therefore assumed that the sparse matrix-vector multiplication done to generate the Krylov basis and the preconditioning are combined into a single operation that is much more expensive than a standard sparse matrix-vector multiplication. Thus, when we refer to "sparse matrix-vector multiplication" with respect to the coefficient matrix A, we mean "sparse matrix-vector multiplication with preconditioning". The only exceptions to this occur when specified otherwise, or in the case of sparse random sketches described in Chapter 5.

Algorithm 1 demonstrates this Arnoldi process being executed using a Modified Gram-Schmidt (MGS) orthogonalization procedure in lines 5-8, though this specific orthogonalization procedure can be replaced with a myriad of others. Finally, in step 11, one solves a least squares problem to obtain a vector $y_m \in \mathbb{C}^m$, and uses the result of this least squares problem to compute the new approximate solution

$x_m = x_0 + V_m y_m$ in step 12. Since

$$r_m := b - Ax_m = b - A(x_0 + V_m y_m) = r_0 - AV_m y_m,$$

and $x_0 + V_m y \in x_0 + \mathcal{K}_m(A, r_0)$ for any $y \in \mathbb{C}^m$, it follows that

$$\begin{aligned}\|r_m\|_2 &= \min_{y \in \mathbb{C}^m} \|r_0 - AV_m y\|_2 = \min_{y \in \mathbb{C}^m} \|b - A(x_0 + V_m y)\|_2 \\ &= \min_{x \in x_0 + \mathcal{K}_m(A, r_0)} \|b - Ax\|_2,\end{aligned}$$

and therefore x_m is chosen to minimize the residual's ℓ_2 norm over all possible solutions in $x_0 + \mathcal{K}_m(A, r_0)$. One can restart this process, using x_m as the new initial guess for another pass of GMRES until an adequate convergence is reached, which is called *restarted GMRES*.

While many potential Krylov subspace methods can be defined to minimize the residual norm, GMRES sets itself apart from the others in the fact that it is able to do so in a computationally efficient manner. More specifically, using a clever exploitation of the orthogonality of the Krylov basis vectors, sequential Givens rotations can be applied to quickly solve a Hessenberg least squares problem that is equivalent to the original GMRES least squares problem (i.e., line 11 of Algorithm 1) while simultaneously providing the residual norm without needing to compute another sparse-matrix vector product. Define $e_1 = [1, 0, \ldots, 0]^T \in \mathbb{C}^{m+1}$. It is straightforward to show (e.g., in [58]) that in exact arithmetic, the Arnoldi process in Algorithm 1 produces orthonormal matrices V_k for each k, and an upper Hessenberg matrix $H_{m+1,m}$ such that

$$AV_m = V_{m+1}H_{m+1,m}, \tag{2.3}$$

and therefore

$$V_{m+1}^* AV_m = V_{m+1}^* V_{m+1} H_{m+1,m} = H_{m+1,m}, \tag{2.4}$$

Moreover, by the orthonormality of the Krylov basis vectors $v_1, \ldots, v_m, v_{m+1}$ and the fact that $v_1 = r_0/\beta$, it follows that

$$V_{m+1}^* r_0 = \beta V_{m+1}^* v_1 = \beta e_1. \tag{2.5}$$

Using (2.4), (2.5), and the fact that the ℓ_2 norm is invariant under orthonormal transformations [69], the least squares problem in line 11 of Algorithm 1 reduces to,

$$\begin{aligned} y_m &= \underset{y \in \mathbb{C}^m}{\arg\min} \|r_0 - AV_m y\|_2 = \underset{y \in \mathbb{C}^m}{\arg\min} \|V_{m+1}^*(r_0 - AV_m y)\|_2 \\ &= \underset{y \in \mathbb{C}^m}{\arg\min} \|\beta e_1 - H_{m+1,m} y\|_2. \end{aligned} \tag{2.6}$$

Thus, the GMRES least squares problem can be solved in relatively few operations, as it is equivalent to solving an $m+1 \times m$ Hessenberg least squares problem, which can be easily transformed to an upper triangular system using Givens rotations successively formed at each iteration and solved in only $O(m^2)$ operations [59]. Even better, if one applies the same successive Givens rotations to the vector βe_1 from the least squares problem, the magnitude of the final entry of the resulting vector is the new approximate's residual norm, which gives an extremely cheap to check convergence criteria without having to perform another sparse matrix-vector product or solve a least squares problem at each iterate [59].

2.4 Performance Considerations of Modern Heterogeneous Machines

Modern supercomputers and high-performance clusters typically fall into the category of distributed heterogeneous machines, which are becoming increasingly GPU-dominated due to their incredible theoretical floating point performance [68]. These machines, or even single GPUs, achieve this theoretical floating point performance by exploiting massive parallelism [50]. Specifically, modern GPUs have hundreds to thousands of processors each, making the architectures attractive for highly parallelizable tasks such as BLAS-3 (matrix-matrix) operations [11].

While this massive parallelism allows for extremely efficient matrix-matrix operations and overall high theoretical performance, the parallelism comes at a cost. Namely, the cost of *communication*, which I will use as a blanket term for both the cost of transferring information between levels of the memory hierarchy and the cost of synchronizing parallel processes, remains high and is often the major performance bottleneck of these machines in practice. Therefore, in order to achieve anywhere close to the theoretical peak performance of these machines with a Krylov subspace method, one needs to ensure their implementation is crafted mindfully to reducing communications, or in other words, a *communication-avoiding* algorithm. Generally speaking, ideal Krylov methods amenable for these new architectures require as few communications as possible, and consist of as many BLAS-3 tasks as possible to maximize the parallelism the machines offer.

More specifically, GPUs and therefore heterogeneous machines have much higher floating point throughput (typically measured in floating point operations per second, or *FLOP/s*) than memory bandwidth, meaning that relatively speaking, the cost of doing one floating point operation is much cheaper than transferring one byte of data throughout the memory for computations. When the time required to transfer data through memory is higher than the time required to execute the floating point operations, we consider the operation *memory-bound*, and when the opposite occurs, we consider the operation *compute-bound*. Because GPUs can execute floating point operations far more efficiently than performing memory transfers, an operation is only compute-bound if the number of floating point operations executed is much higher than the number of bytes transferred to execute the operation. The ratio of floating point operations performed to bytes transferred during the operation is sometimes referred to as *arithmetic intensity*, and high arithmetic intensity is prerequisite for high performance on a modern heterogeneous machine.

Specifically, an operation is compute-bound on a specific machine if the arithmetic intensity of the operation must exceed the ratio of FLOP/s to memory bandwidth that the machine offers, otherwise it is memory-bound.

For example, sufficiently large matrix-matrix multiplications and other BLAS-3 operations are typically compute-bound and therefore perform excellently on GPUs and by extension heterogeneous machines [51]. To see this, if one needs to multiply an $n \times m$ matrix by a $m \times k$ matrix, one must perform $O(nmk)$ floating point operations while only transferring $O(nm + mk)$ bytes of data, thereby giving high arithmetic intensity. On the other hand, matrix-vector multiplication and other BLAS-1/2 operations are memory-bound, because to multiply a $n \times m$ matrix by a m-dimensional vector, one must perform $O(nm)$ floating point operations while transferring $O(nm)$ bytes of data, thereby offering very low arithmetic intensity and therefore low performance on modern machines [51].

Ideally, our application will be mostly compute-bound, and therefore, Krylov methods amenable to modern high performance machines should ideally minimize communications and leverage as many compute-bound (e.g., BLAS-3) operations as possible. In practice, it is best if these compute-bound operations are available in a hardware-optimized implementation by the chip manufacturer, which is conveniently the case with BLAS-3 operations [46].

2.5 Shortcomings of Traditional Krylov Methods on Modern Machines

Widely used Krylov methods designed for general non-Hermitian systems typically either:

1. use an Arnoldi-based orthogonalization procedure, where each new Krylov vector must be orthogonalized against each previous vector, or
2. use a Lanczos bi-orthogonalization scheme, which uses short-recurrence relation to generate the Krylov vectors.

While Lanczos bi-orthogonalization based methods' use of a short-recurrence to generate the new Krylov vectors significantly reduces the computational cost and the amount of information that must be stored and transferred, the downsides of such methods are that they do not fully orthogonalize the Krylov basis and that they require multiple sparse matrix-vector multiplications per iteration [58]. Typically, this latter challenge alone can make them unattractive in practice compared to Arnoldi-based methods which only require one sparse matrix-vector multiplication per iteration. A consequence of the non-orthogonal Krylov bases that Lanczos bi-orthogonalization based methods rely on is that the methods do not minimize the residual norm at each iteration. As a result, Krylov methods using a Lanczos bi-orthogonalization have unpredictable convergence behavior (e.g., BiCGSTAB, [65, 71]), or their lack of optimality simply leads to a convergence delay compared to GMRES (e.g., QMR or TFQMR [25, 26]), which is another reason Arnoldi-based methods remain popular.

However, Arnoldi-based methods like GMRES have downsides as well. On top of imposing additional memory requirements to store the entire Krylov basis, the Arnoldi process done in lines 3-10 of Algorithm 1 consist primarily of vector-vector operations (BLAS-1), which are memory-bound operations, as explained in Section 2.4. This version of GMRES can be parallelized in a slightly better fashion while maintaining numerical stability by replacing the modified Gram-Schmidt (MGS) procedure in lines 5-8 of Algorithm 1 with a reorthogonalized classical Gram-Schmidt (CGS2) orthogonalization within the Arnoldi process, given in lines 5-7 of Algorithm 2. The advantage of this approach is that many of the BLAS-1 operations are replaced with more parallelizable matrix-vector (BLAS-2) operations. However, the Arnoldi procedure in lines 4-8 of Algorithm 2 has low arithmetic intensity and thus is still memory-bound, and therefore the orthogonalization procedure of the Krylov vectors is sub-optimal on modern GPU systems and poses a significant performance bottleneck in practice.

Algorithm 2 m iterations of GMRES with CGS2 Orthogonalization

```
    Input:  Right hand side $b \in \mathbb{C}^n$, initial guess $x_0 \in \mathbb{C}^n$
    Output: Approximate solution $x_m \in \mathbb{C}^n$
 1: $r_0 = b - Ax_0$
 2: $\beta = \|r_0\|_2, v_1 = r_0/\beta$
 3: **for** $j = 1, \ldots, m$ **do**
 4:     $w = Av_j$                                  // Generate next Krylov vector
 5:     $\hat{h} = V_j^* w$, $w = w - V_j \hat{h}$            // Orthog. against prior vectors
 6:     $\bar{h} = V_j^* w$, $w = w - V_j \bar{h}$
 7:     $h_{1:j,j} = \hat{h} + \bar{h}$
 8:     $h_{j+1,j} = \|w\|_2$, $v_{j+1} = w/h_{j+1,j}$                        // Normalize
 9: **end for**
10: $y_m = \arg\min \|b - AV_m y_m\|_2$
11: $x_m = x_0 + V y_m$
```

CHAPTER 3
OPTIMAL SIZE OF THE BLOCK OF BLOCK GMRES ON GPUS

In contrast to the communication-intensive standard GMRES algorithm, the block version of GMRES (BGMRES) requires less communications by relying heavily on BLAS-3 operations and reducing the number of accesses to the linear systems' coefficient matrix but performs more floating point operations than its single right hand side (RHS) counterpart. Thus, BGMRES is most advantageous over the standard single RHS GMRES when the cost of communication is high while the cost of floating point operations is not. This is the particular case on modern Graphics Processing Units (GPUs) and by extension modern heterogeneous machines, while it is generally not the case on traditional Central Processing Units (CPUs).

In this chapter, experiments on both GPUs and CPUs are shown that compare the performance of BGMRES against GMRES as the number of RHS increases. The experiments indicate that there are many cases in which BGMRES is slower than GMRES on CPUs, but faster on GPUs. Furthermore, when varying the number of RHS on the GPU, there is an optimal number of RHS where BGMRES is clearly most advantageous over GMRES. A computational model is developed using hardware specific parameters, showing qualitatively where this optimal number of RHS is, and this model also helps explain the phenomena observed in the experiments.

3.1 Introduction

We consider the problem of solving a linear system with s right hand sides (RHS): namely, $AX = B$, where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times s}$ where $n \gg s$ (i.e., B is a tall-and-skinny matrix). We assume all vectors and matrices are real in this chapter, though everything presented also extends to the complex case. Generally, A is sparse, so let $nz \ll n^2$ be the number of non-zeros in A.

Block Krylov methods, such as block GMRES (BGMRES), are versions of traditional Krylov subspace methods used for solving such systems with multiple RHS, or for accelerating the convergence of a system with one right hand side [53], [58, Section 6.12]. We have this latter situation very much in mind in our investigations. In particular, one can accelerate the convergence of GMRES by leveraging a richer BGMRES solution space (usually by adding random vectors to the original RHS b to produce a block RHS B).

It is well-documented in existing literature that BGMRES can outperform the classical GMRES method s times separately, which we will refer to as "$s\times$GMRES" throughout the remainder of the chapter, but not always [61, 62, 73]. Algorithms 3 and 4 explicitly describe the BGMRES and $s \times$ GMRES algorithms, respectively.

Our original goal was to see if this paradigm changes when we implement these algorithms on Graphics Processing Units (GPUs). In particular, we wanted to see if there are some cases for which the block method is slower in execution time on the Central Processing Unit (CPU) than the single right hand side method repeated

Algorithm 3 BGMRES: m iter.

```
 1: $R_0 = B - AX_0$
 2: $[\boldsymbol{V}_1, \beta] = \text{qr}(R_0)$
 3: for $j = 1, \ldots, m$ do
 4:     $W = A\boldsymbol{V}_j$
 5:     for $i = 1, \ldots, j$ do
 6:         $\boldsymbol{H}_{i,j} = W^T \boldsymbol{V}_i$
 7:         $W = W - \boldsymbol{V}_i \boldsymbol{H}_{i,j}$
 8:     end for
 9:     $[\boldsymbol{V}_{j+1}, \boldsymbol{H}_{j+1,j}] = \text{qr}(W)$
10: end for
11: $Y_m = \arg\min \|\boldsymbol{H}Y - E_1\beta\|_F$
12: $X_m = X_0 + \boldsymbol{V}_{1:m} Y_m$
```

Algorithm 4 $s \times$ GMRES: m iter.

```
 1: for $k = 1, \ldots, s$ do
 2:     $r_0 = b_k - A(X_0)_k$
 3:     $\beta = \|r_0\|_2, v_1 = r_0/\beta$
 4:     for $j = 1, \ldots, m$ do
 5:         $w = Av_j$
 6:         for $i = 1, \ldots, j$ do
 7:             $h_{i,j} = w^T v_i$
 8:             $w = w - h_{i,j} v_i$
 9:         end for
10:         $h_{j+1,j} = \|w\|_2, v_{j+1} = w/h_{j+1,j}$
11:     end for
12:     $y_m = \arg\min \|Hy - \beta e_1\|_2$
13:     $(X_m)_k = (X_0)_k + V_m y_m$
14: end for
```

multiple times on the CPU, while on the GPUs, the reverse holds. This holds true, and we generally observe much faster execution times and larger improvements in the case of the block method. We also observe that for any fixed matrix, when the number of RHS increase, there is a point in which the improvements start to decrease and eventually any advantage of the (unrestarted) block method is lost.

The reason for this latter observation is the higher data movement cost of the single right hand side GMRES weighed against the higher cost of the floating point operations in the block method, as required by the block Arnoldi procedure and the QR factorizations needed to orthogonalize the block basis of the block Krylov subspace. The balance between a richer block Krylov subspace and the increased computational cost of BGMRES was first mentioned by Vital when she proposed the algorithm [73]. Here we develop a simple computational model for the run-time of each algorithm, which is described in Section 3.3. This model qualitatively shows that this phenomenon should be expected.

Algorithm 5 GMRES-LI with s RHS: m iter.

```
1:  $R_0 = B - AX_0$
2:  for $k = 1, \dots, s$ do
3:      $\beta_k = \|(R_0)_k\|_2$
4:  end for
5:  $\boldsymbol{V}_1 = R_0 * \texttt{diag}(1./\beta)$
6:  for $j = 1, \dots, m$ do
7:      $W = A\boldsymbol{V}_j$
8:      for $i = 1, \dots, j$ do
9:          $\boldsymbol{H}_{i,j} = \texttt{diag}(\texttt{diag}(W^T\boldsymbol{V}_i))$
10:         $W = W - \boldsymbol{V}_i\boldsymbol{H}_{i,j}$
11:     end for
12:     $\boldsymbol{V}_{j+1} = W * \texttt{diag}(1./\texttt{diag}(\boldsymbol{H}_{j+1,j}))$
13: end for
14: $Y_m =$ least squares solution for each RHS
15: $X_m = X_0 + \boldsymbol{V}_{1:m}Y_m$
```

Additionally, we discuss an alternative "loop-interchanged" formulation of $s \times$ GMRES, which we refer to as "GMRES-LI" and is given explicitly in Algorithm 5, that avoids unnecessary accesses of the coefficient matrix to reduce data movement costs. This idea is not new, as loop-interchanged Krylov methods have been proposed in [56] and elsewhere under a variety of names, such as "pseudo-Block Krylov methods" [10]. We compare the loop-interchanged algorithm to block GMRES to study the advantage that the block Krylov subspace method provides over the standard single RHS Krylov subspace method separate from the reduction in data movement costs.

In other words, we compare BGMRES to both standard GMRES repeated sequentially ($s \times$ GMRES) and a loop-interchanged GMRES because BGMRES can outperform standard GMRES for two major reasons:

1. due to reduced iteration counts (and therefore reduced computational cost) from the block Krylov space and
2. due to reduced communication costs from the block Arnoldi procedure.

BGMRES and standard GMRES repeated sequentially ($s \times$ GMRES) are two approaches to solving a large sparse non-symmetric problem with multiple RHS that should naturally be compared. We compare BGMRES to the loop-interchanged GMRES (GMRES-LI) as well, because the dominant communication costs of the algorithms are the same, but BGMRES uses a richer block search space and hence often requires fewer iterations to converge. Equalizing the communication costs eliminates any benefits of BGMRES over GMRES due to reason 2, and therefore comparing BGMRES to GMRES-LI identifies when the speedups can be solely attributed to reason 1.

When GMRES converges slowly, BGMRES is expected to perform well due to reason 1. We confirm this on the GPU by experimenting with GMRES-LI. The analysis is also interesting in cases when GMRES converges quickly. In these cases BGMRES is often slower than GMRES on CPUs, as BGMRES will incur more floating point operations than GMRES and communication costs are generally relatively low on CPUs, which eliminates communication considerations. Conversely, GPU memory latency is high, making communication costs significant [3]. Thus, on GPUs we observed that BGMRES can outperform GMRES even when GMRES converges quickly due to reason 2. We confirm that the communication costs are responsible for the speedups of BGMRES over GMRES in these cases by experimenting with GMRES-LI again.

In this chapter, our contribution consists of first confirming that on GPUs BGMRES is not only faster, but is more advantageous than $s \times$ GMRES in most practical cases. Second, we observe that there is an optimal number of RHS where BGMRES takes most advantage of the GPU architecture. We present a new computational model which helps us explain why this is so.

3.2 Theoretical Properties of the BGMRES, $s \times$ GMRES, and GMRES-LI Algorithms

Before proceeding, we define block Krylov subspaces to highlight the theoretical differences between BGMRES and $s\times$ GMRES.

Definition 3.1 (Block Krylov Subspace)**.** *Given some block vector $B \in \mathbb{R}^{n\times s}$, we first define the space,*

$$\mathcal{B}_m(A, B) = \mathcal{K}_m(A, b_1) + \cdots + \mathcal{K}_m(A, b_s).$$

Then block Krylov subspaces *generated by the matrix A and RHS B is defined as,*

$$\mathcal{B}_m^{\square}(A, B) = \underbrace{\mathcal{B}_m(A, B) \times \cdots \times \mathcal{B}_m(A, B)}_{s \text{ times}}. \tag{3.1}$$

Now, let $R_0 \in \mathbb{R}^{n\times s}$ be a block vector whose columns are the initial residual vectors used for each RHS; i.e., R_0 is the initial residual for BGMRES and its columns are the initial residuals for $s\times$GMRES. Let X_{BG}, X_G be the solution block vectors generated after m iterations of BGMRES and $s\times$ GMRES, respectively.

By design, $R_{\text{BG}} := B - AX_{\text{BG}}$ where $X_{\text{BG}} \in \mathcal{B}_m^{\square}(A, R_0)$ minimizes the residual's Frobenius norm over all solutions $X \in \mathcal{B}_m^{\square}(A, R_0)$ [32]. In other words:

$$X_{\text{BG}} = \underset{X \in \mathcal{B}_m^{\square}(A,R_0)}{\arg\min} \|B - AX\|_F \tag{3.2}$$

Now, $R_{\text{G}} := B - AX_{\text{G}}$, where

$$X_{\text{G}} \in \mathcal{K}_m(A, (R_0)_1) \times \cdots \times \mathcal{K}_m(A, (R_0)_s) \tag{3.3}$$

is selected so that each column of X_{G} minimizes each column of the residual's Euclidean norm [59]. In other words,

$$\begin{aligned} X_{\text{G}} &= [x_1, \ldots, x_s] \\ x_k &= \underset{x \in \mathcal{K}_m(A,(R_0)_k)}{\arg\min} \|b_k - Ax_k\|_2 \text{ for } k = 1, \ldots, s, \end{aligned}$$

which is also equivalent to

$$X_{\text{G}} = \underset{X \in \mathcal{K}_m(A,(R_0)_1) \times \cdots \times \mathcal{K}_m(A,(R_0)_s)}{\arg\min} \|B - AX\|_F.$$

To understand when BGMRES may outperform the $s \times$GMRES, we first review the dominant costs of each algorithm, similar to analysis performed by Vital when she proposed the method [73]. In Tables 3.1 and 3.2, we summarize the dominant computational costs of BGMRES and GMRES respectively. In this analysis, we assume that in each case we compute m iterations. We emphasize that this comparison is for *the same* number of iterations m, though in practice BGMRES should converge in fewer iterations; see Proposition 3.2.

Floating Point Operations

	Op. per Step	No. of Steps	Total Op.
`spmv`	$O(nz \cdot s)$	m	$O(nz \cdot ms)$
Block Orthogonalization	$O(ns^2)$	$O(m^2)$	$O(nm^2s^2)$
MGS QR Factorization	$O(ns^2)$	m	$O(nms^2)$
Reflections	$O(s^3)$	$O(m^2s)$	$O(m^2s^4)$

Data Movement

	Op. per Step	No. of Steps	Total Op.
`spmv`	nz	m	$nz \cdot m$

Table 3.1: Computational costs of m iterations of BGMRES

In BGMRES (Algorithm 3), the sparse coefficient matrix A with nz non-zeros is accessed once per iteration, so m times total. The dominant floating point costs occur in the Block Arnoldi procedure (lines 3-10 in Algorithm 3), in which we multiply A by the previous Krylov basis (block) vector of size $n \times s$, which requires $2 \cdot nz \cdot s$ floating point operations, and happens m times. Then, in the innermost loop in the Block Arnoldi procedure (which occurs $O(m^2)$ times), we perform a matrix-matrix multiplication of an $s \times n$ and an $n \times s$ matrix, requiring $2ns^2$ floating point operations. We also perform a block vector update (AXPY of $n \times s$ matrices), which requires the same work as the aforementioned matrix-matrix product. In the outer loop (which occurs m times), we compute a Modified Gram-Schmidt (MGS) QR factorization to orthogonalize the vectors within the new block Krylov basis vector, which requires $O(ns^2)$ floating point operations. Lastly, the cost of performing the Householder reflections for solving the least squares problem requires $O(m^2s^4)$ floating point operations.

Floating Point Operations

	Op. per Step	No. of Steps	Total Op.
`spmv`	$O(nz)$	ms	$O(nz \cdot ms)$
Orthogonalization	$O(n)$	$O(m^2s)$	$O(nm^2s)$
Normalization	$O(n)$	ms	$O(nms)$
Reflections	$O(m^2)$	ms	$O(m^3s)$

Data Movement

	Op. per Step	No. of Steps	Total Op.
`spmv`	nz	ms	$nz \cdot ms$

Table 3.2: Computational costs of m iterations of $s \times$ GMRES

We use a similar analysis for $s \times$ GMRES (Algorithm 4). In GMRES, the coefficient matrix A with nz non-zeros is accessed once per iteration, so $m \cdot s$ times total. The dominant floating point costs occur in the Arnoldi procedure (lines 4-11 of Algorithm 4), where we multiply A by the previous basis vectors of size n, which requires $2 \cdot nz$ floating point operations, and happens $m \cdot s$ times. Then, in the innermost loop of the Arnoldi procedure (which occurs $O(m^2) \cdot s$ times), we perform a dot product and an AXPY, both requiring $O(n)$ floating point operations. Other operations used in the outer loop (e.g., the normalization of the basis vectors) occur ms times and require $O(n)$ floating point operations. Lastly, applying Givens rotations requires $O(m^2)$ floating point operations, repeated $m \cdot s$ times.

Referring to Tables 3.1 and 3.2, we see that m iterations of BGMRES requires more floating point operations in the orthogonalization procedure than m iterations $s \times$ GMRES. However, $s \times$ GMRES accesses A more often and uses the same number of floating point operations as BGMRES in the `spmv` step.

Since BGMRES uses a richer block Krylov subspace to find its approximate solution than the s separate Krylov subspaces used in $s \times$ GMRES, BGMRES converges in at most as many iterations (per RHS) as $s \times$ GMRES. Although this fact is well-known, we include a brief proof for completeness.

Proposition 3.2. *Fix some positive integer m, and let $R_{BG} \in \mathbb{R}^{n \times s}$ be the residual block vector after m iterations of BGMRES and let $R_G \in \mathbb{R}^{n \times s}$ be a block whose columns are the s residual vectors after applying m iterations of GMRES to each of the s RHS. Then $\|R_{BG}\|_F \leq \|R_G\|_F$.*

Proof. Recall $R_{\text{BG}} = B - AX_{\text{BG}}$ for $X_{\text{BG}} \in \mathcal{B}_m^{\square}(A, R_0)$ that minimizes the residual's Frobenius norm over all solutions $X \in \mathcal{B}_m^{\square}(A, R_0)$, as in (3.2). On the other hand, $R_{\text{G}} = B - AX_{\text{G}}$, where (3.3) holds.

Since $\mathcal{K}_m(A, (R_0)_j) \subset \mathcal{B}_m(A, R_0)$ for each RHS $j = 1, \ldots, s$, we have that

$$\mathcal{K}_m(A, (R_0)_1) \times \cdots \times \mathcal{K}_m(A, (R_0)_s) \subset \mathcal{B}_m^{\square}(A, R_0)$$

Therefore, $X_{\text{G}} \in \mathcal{B}_m^{\square}(A, R_0)$, but since X_{BG} is minimizes the residual's Frobenius norm for all block vectors in this space, it follows that $\|R_{\text{BG}}\|_F \leq \|R_{\text{G}}\|_F$, and therefore the convergence criteria will be reached by BGMRES no later than when it is reached by $s \times$ GMRES. $\square$

Ideally BGMRES converges in far fewer iterations than $s \times$ GMRES by leveraging the richer block Krylov subspace, perhaps even in s times fewer iterations. In practice, this is often not the case. Instead, we see a small to moderate reduction in the iterations by using BGMRES, and in some cases the two algorithms require the same number of iterations. Understanding precisely how much of an advantage the block method offers in terms of iteration counts is difficult to predict and depends on a variety of factors, including the number of RHS s used in the system and the difficulty of the problem.

Even without knowing the iteration reduction that BGMRES offers over $s\times$GMRES, it follows from Tables 3.1 and 3.2 that BGMRES is most advantageous if accessing the coefficient matrix A is very costly and floating point operations are very inexpensive. This is generally the case for GPUs, while it is not the case for CPUs.

GPUs are higher-latency machines with extremely high floating point throughput. Hence, we can expect floating point operations to be cheaper and accessing information between levels of the memory hierarchy to be more expensive on a GPU compared to a similarly high-performance CPU. For this reason, one could expect to find that the relative performance of BGMRES compared to $s \times$ GMRES is better on a high-performance GPU than it is on a high-performance CPU.

While BGMRES is an attractive option to address the issue of reducing memory accesses of the coefficient matrix, it is not the only one. Namely, one can carefully interchange outer loops of the $s \times$ GMRES algorithm to perform the `spmv` operations for all of the RHS at once like BGMRES, while being mathematically equivalent to $s \times$ GMRES. We refer to this version of $s \times$ GMRES as "Loop-Interchanged GMRES" or GMRES-LI, and provide an explicit algorithm in Algorithm 5. We reiterate that this idea is not new, as loop-interchanged Krylov methods have been proposed elsewhere under a variety of names, including Loop-Interchanged Krylov methods or pseudo-Block Krylov methods [10, 56].

Using Algorithm 5, it is straightforward to find the computational costs of GMRES-LI in a similar manner as before, which we report in Table 3.3.

Floating Point Operations

	Op. per Step	No. of Steps	Total Op.
`spmv`	$O(nz \cdot s)$	m	$O(nz \cdot ms)$
Orthogonalization	$O(ns)$	$O(m^2)$	$O(nm^2 s)$
Normalization	$O(ns)$	m	$O(nms)$
Reflections	$O(m^2)$	ms	$O(m^3 s)$

Data Movement

	Op. per Step	No. of Steps	Total Op.
`spmv`	nz	m	$nz \cdot m$

Table 3.3: Computational costs of m iterations of GMRES-LI

GMRES-LI combines the lower number of floating point operations of $s \times$ GMRES with the reduced data movement cost of BGMRES. Thus, the relative performance of BGMRES compared to GMRES-LI is driven entirely by iteration counts of each algorithm, which depend on how much more of a reduction in the residual norm that the richer block Krylov subspace provides.

3.3 Qualitative Runtime Model

To qualitatively analyze the runtime of the BGMRES and single RHS GMRES algorithms, we use a simple "latency-bandwidth model" used by Hoemmen in his PhD thesis [37]. Namely, we assume communication time is dictated by passing of messages. A *message* is a sequence of m words, where a *word* is a piece of data (e.g., floating point numbers, integers, etc.). The time required to send a message of m words can be modeled by the following function:

$$\text{Time}_{\text{Message}}(m) = \alpha + \beta \cdot m$$

where m is the number of words, α is the latency (in seconds), and β is the inverse bandwidth (in seconds per word). The time to execute floating point operations is modeled in a straightforward way via:

$$\text{Time}_{\text{flops}}(m) = \gamma \cdot m$$

where m is the number of floating point operations, and γ is the inverse floating point throughput (in seconds per floating point operations, or $1/\text{flops}$).

Putting these together, we can construct a simple model for the runtime of the GMRES and BGMRES algorithms by counting the floating point operations required to execute each algorithm and by counting the number of times data must be moved between different levels of the memory heirarchy. Before doing so, we make several additional assumptions to simplify our model:

1. We assume all memory aside from the sparse coefficient matrix in our linear system resides in fast memory (i.e., cache). Hence, all memory transfers occur when the coefficient matrix A is accessed.

   This is not always the case, for example if the number of non-zeros nz in A, the number of rows n in A, the number of RHS s, and the number of BGMRES iterations m satisfy $nz \approx n \cdot m \cdot s$. In this case, the matrix storing the block Krylov basis $V \in \mathbb{R}^{n \times m \cdot s}$ has roughly the same size of the coefficient matrix. Hence, data transfers would be required to access this information as well.

2. There are times that the GMRES and BGMRES algorithms require particular values of an array to be modified (e.g., modifying the first value of a vector when performing a Householder reflection). If these arrays are stored on the GPU, we cannot directly modify particular entries of an array without using a parallel operation. Instead, we must copy these arrays back to the CPU,

modify the value on the CPU, and then send the updated information back to the GPU again. While this process certainly can be costly, we neglect these costs here, as experimental data indicates that the cost of the floating point operations dominates these costs in our implementations of BGMRES and GMRES, as our algorithms were written to avoid performing such direct modifications of arrays whenever it was convenient to do so.

3. The cost of applying a preconditioner is neglected. This can be a non-trivial cost, but in our experiments, we use a Jacobi preconditioner which can be applied with minimal computational work, and hence will not dominate the computational cost of our algorithms.

Since the latency-bandwidth model was designed for CPU performance rather than GPU performance, our runtime models are not meant to be predictive of the precise runtime of the BGMRES and GMRES algorithms executed on the GPU. Modelling GPU runtime accurately is far more complicated than this model, and requires software to simulate the architecture's execution times [3]. Instead, our model is intended to provide a strictly qualitative understanding of how the different algorithms perform relative to each other as the number of RHS change.

Considering the above assumptions, we now introduce an explicit version of our runtime model. In Tables 3.1 and 3.2, we summarized the key computational costs of BGMRES and GMRES respectively. Using the costs described in Table 3.1, we obtain the following computational runtime model for BGMRES:

$$\begin{aligned}\text{Time}_{\text{BG}}(n, nz, m, s) \approx &\overbrace{\alpha \cdot m + \beta \cdot nz \cdot m}^{\text{Data Movement}} \\ &+ \gamma \cdot (\underbrace{2nz \cdot ms + 2nm^2s^2}_{\text{Block Arnoldi}} + \underbrace{2nms^2}_{\text{QR}} + \underbrace{2m^2s^4}_{\text{Reflections}})\end{aligned} \tag{3.4}$$

Similarly, using the costs described in Tables 3.2 and 3.3, we obtain the following computational runtime models for $s \times$ GMRES and for GMRES-LI:

$$\begin{aligned}\text{Time}_{\text{G}}(n, nz, m, s) \approx &\overbrace{\alpha \cdot ms + \beta \cdot nz \cdot ms}^{\text{Data Movement}} \\ &+ \gamma \cdot (\underbrace{2nz \cdot ms + 2nm^2s}_{\text{Arnoldi}} + \underbrace{3nms}_{\text{Normalization}} + \underbrace{2m^3s}_{\text{Reflections}})\end{aligned} \tag{3.5}$$

$$\begin{aligned}\text{Time}_{\text{LI}}(n, nz, m, s) \approx &\overbrace{\alpha \cdot m + \beta \cdot nz \cdot m}^{\text{Data Movement}} \\ &+ \gamma \cdot (\underbrace{2nz \cdot ms + 2nm^2s}_{\text{Arnoldi}} + \underbrace{3nms}_{\text{Normalization}} + \underbrace{2m^3s}_{\text{Reflections}})\end{aligned} \tag{3.6}$$

Comparing these three approximate runtime models, we notice the following:

1. The GMRES algorithm requires s times as many data transfers as BGMRES and GMRES-LI. Clearly, this implies that if the cost of data movement is steep (i.e., if the latency α or inverse bandwidth β are far greater than the

floating point throughout γ), then BGMRES will be far faster than GMRES assuming the number of iterations m used in each algorithm is approximately the same.

2. While the BGMRES method requires a QR factorization at each iteration, the GMRES and GMRES-LI methods do not, and instead use a normalization procedure that is $O(s)$ times cheaper than the QR factorization. This is an additional cost of BGMRES over GMRES and GMRES-LI.

3. In most practical applications, n and nz are far greater than m, s, so the final terms in these two models (the $2m^2s^4$ and $2m^3s$ terms in BGMRES and GMRES/GMRES-LI, respectively) are insignificant compared to the other operation counts.

4. Purely in terms of floating point operations required, we see that the block Arnoldi procedure forces BGMRES to use $O(nm^2s^2)$ operations to construct the Arnoldi basis, while the single right hand side GMRES and GMRES-LI only use $O(nm^2s)$ operations to do so. In spite of this, since modern GPUs can execute BLAS-3 operations very efficiently [23, 46], the block Arnoldi procedure–and by extension the BGMRES method–is not weighed

down significantly by this extra floating point cost in practice, but we still consider the extra operations in our model.

To illustrate the models of the GMRES and BGMRES runtime, Figure 3.1 displays a plot of the "Block Speedup" quantified by $\frac{\text{Time}_\text{G}}{\text{Time}_\text{BG}}$ with $n = 147,000$, $nz = 3,489,300$ and experimentally determined values for m_G and m_BG. These are realistic parameters that will be seen in the next section for solving $AX = B$ using BGMRES and GMRES, where B is a random RHS matrix, and A is 'FEM_3D_thermal2' from the SuiteSparse Library [18]. We chose the parameters $\beta = 1.1 \times 10^{-12}$ and $\gamma = 1.3 \times 10^{-13}$ from hardware specifications on the GPU we used, the NVIDIA V100 SXM2 [52]. We used $\alpha = 3 \times 10^{-4}$ based on numerical results of applying the latency-bandwidth model with the aforementioned values for β and γ to a matrix-vector multiplication of A times a vector of ones.

Figure 3.1 indicates if we fix parameters for α, β, γ based on our computer's hardware and fix n and nz based on the 'FEM_3D_thermal2' coefficient matrix, and m_G and m_BG based on observed convergence behavior for this problem, then there is some s such that the Block Speedup peaks.

This phenomenon is consistent with our numerical experiments for the 'FEM_3D_thermal2' matrix and for every other problem we attempted that does not require restarts to converge within the GPU's storage constraints. Namely, we notice that the speed of non-restarted BGMRES relative to GMRES peaks at a certain number of right hand sides, and performance gains diminish for any number of right

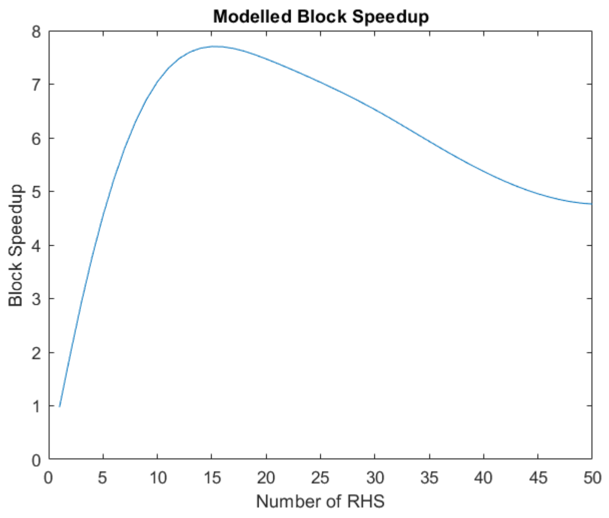


Figure 3.1: Block Speedup = $\text{Time}_{\text{G}}/\text{Time}_{\text{BG}}$ with $n = 147,000$, $nz = 3,489,300$, $\alpha = 3 \times 10^{-4}$, $\beta = 1.1 \times 10^{-12}$, $\gamma = 1.3 \times 10^{-13}$, using realistic iteration counts m_{BG} and m_{G}. The vertical axis is above 1 when BGMRES is faster than GMRES. The horizontal axis is the number of RHS s.

hand sides smaller or larger than this critical point. In general, this phenomenon does not always hold when the BGMRES method needs restarts to converge. In Section 3.5, we show that our model is still consistent with our observations when restarts are required.

3.4 Implementation Details

We wrote and tested parallel implementations of $s \times$ GMRES, BGMRES, and GMRES-LI on both the CPU and GPU using the Kokkos C++ Performance Portability Library in order to keep our CPU and GPU implementations similar enough to make a fair comparison [24]. Within Kokkos, our GPU implementations utilized NVIDIA's cuBLAS and cuSPARSE linear algebra libraries while CPU implementations used the Intel MKL library with OpenMP [11, 17, 38, 46, 49]. Our implementation of BGMRES was inspired by MATLAB code written by Soodhalter [66].

While we intended for the CPU and GPU implementations of our algorithms to be similar, we also kept performance considerations in mind. For instance, in order to edit many data types stored on the GPU directly (e.g., matrices and vectors used to solve the least squares problem in our GMRES and BGMRES algorithms), one must copy the data to the host first, then modify the data, and then copy it back to the GPU. In contrast, CPU code does not require these copies to be made, and so they were not included on the CPU implementations.

Additionally, we made an effort to be mindful of these memory transfer costs when creating the GPU implementations and avoided them when it was convenient to do so. While we were mindful of the communication overhead costs, we did not use more sophisticated Communication-Avoiding implementations of the subroutines, e.g., in [37, 78].

While many different implementations of the block Arnoldi process within the BGMRES algorithm can be considered, the block Arnoldi process within our BGMRES code is based on a block MGS procedure with a standard MGS QR factorization used to orthogonalize the next iterate's columns. Other options with better stability properties exist, however, this implementation was chosen as it is reasonably stable with respect to loss of orthogonality errors without incurring significantly more computational work, and it is simple to implement compared to sophisticated stable low-synch variants [13].

To solve the Frobenius norm minimization process for BGMRES, we computed the upper triangular part of the QR factorization of the banded Hessenberg H and the transformed right hand side using Householder reflections, employing the strategy suggested by Gutknecht and Schmelzer [33]. For efficiency, these Householder reflections were executed in parallel using batched linear algebra routines. All remaining parallel tasks were simple and were executed by parallel for-loops over vectors and matrices, which Kokkos mapped to OpenMP on the CPU and CUDA on the GPU.

In some applications, the columns of the RHS B may be closely related, or the columns of the block vectors produced by the block Arnoldi process may become linearly dependent. This issue is traditionally alleviated through deflation of the linearly dependent vectors via a Rank-Revealing QR (RRQR) factorization [32]. However, deflating nearly linearly dependent vectors can slow down the convergence of the BGMRES method as the deflated block Krylov Subspace is less rich than the non-deflated space [41]. To avoid losing information in the search space while maintaining linear independence of the columns of the Arnoldi vectors, it is possible to “deflate” by introducing random vectors in place of the nearly linearly dependent ones [66].

The exact cost of deflation depends on the strategy used, but in general, deflated BGMRES can incur more computational cost than using a non-deflated BGMRES. In our experiments, we used randomly generated RHS. Hence, the likelihood of our experiments needing deflation was low, so our BGMRES implementation did not use any deflation scheme. As a result, our code may incur less computational cost than a more robust implementation of BGMRES, which should be considered when we compare the runtime of BGMRES to s times GMRES.

We compared the performance of our CPU and GPU implementations of GMRES and BGMRES using 10 matrices of varying size and sparsity. We generated random RHS block-vectors B using a fixed random seed in order to have random RHS that could be reproducible and exactly the same when we compared the block and single RHS versions of our algorithms. All test problems used matrices from the SuiteSparse library and are listed in Table 3.4 [18]. All test problems were preconditioned with a Jacobi preconditioner on the left, and all residuals and relative residuals computed were the left-preconditioned residuals.

We required that all of our test problems satisfy the following criteria:

1. The coefficient matrix A fit in our GPU's available memory (16GB)

2. The problem converges to a solution in fewer than 100 restarts for both our block and single RHS methods

3. The coefficient matrix is too large to fit in our GPU's cache (16MB)

Matrix	Rows	nz	Sparsity
`shipsec8`	114,919	3,303,553	0.0250%
`Dubcova3`	146,689	3,636,643	0.0169%
`FEM_3D_thermal2`	147,900	3,489,300	0.0160%
`SiO2`	155,331	11,283,503	0.046%
`thermomech_dM`	204,316	1,423,116	0.0034%
`stomach`	213,360	3,021,648	0.0066%
`hood`	220,542	9,895,422	0.0221%
`CO`	221,119	7,666,057	0.0157%
`CoupCons3D`	416,800	17,277,420	0.0099%
`cage13`	445,315	7,479,343	0.0038%

Table 3.4: Coefficient Matrices from SuiteSparse Collection

In order to satisfy these criteria while analyzing problems with many right hand sides, we chose the number of inner iterations m for the BGMRES method to be as large as possible (without exceeding the GPU's total memory) for 50 right hand sides, and scaled the number of inner iterations m for different numbers of right hand sides so that the maximal Krylov subspace had the same size for all numbers of right hand sides. For example, if the GPU could execute a maximum of $m = 100$ inner iterations of a problem with $s = 50$ right hand sides before running out of memory, we allowed $m = 500$ inner iterations for $s = 10$ right hand sides, and $m = 5000$ inner iterations for $s = 1$ right hand side, allowing for a total of $m \cdot s = 5000$ columns in the basis of our Krylov subspace, regardless of the number of RHS s used.

For each of the s RHS $b_i = B_{1:n,i}$, our single RHS GMRES algorithm terminates when the relative residual in the 2-norm is at most 10^{-8} for each $i = 1, \ldots, s$. To keep a consistent stopping criteria, our BGMRES algorithm terminates when the relative residual in the Frobenius norm is at most $10^{-8}\sqrt{s}$.

GMRES-LI was implemented without restarts, as implementing an appropriate restarting strategy becomes tedious when the different RHS converge at different speeds. For simplicity, the GMRES-LI algorithm terminates when one of the RHS converges, since the experiments test random RHS which converge at approximately the same speed. It is worth noting that the BGMRES method is subject to far more stringent stopping criteria than GMRES-LI.

All test results were obtained using Kokkos 3.1.00, Cuda 10.0.13, and GCC 7.5.0. The C++ code was run on a 20-core Intel(R) Xeon(R) CPU E5-2698 v4 2.2GHz processor. All GPU implementations were performed on a single NVIDIA V100 SXM2 GPU in the DGX-1 server at Temple University.

3.5 Experimental Results

The numerical results for the unrestarted test problems are shown in Figure 3.2. Generally, the GPU methods perform very well compared to their CPU counterparts in terms of runtime, demonstrated in the bar graphs. Moreover, we notice several cases where the Block Speedup on the CPU is less than 1 while the Block Speedup on the GPU is above 1. This implies that there are many cases where using the single RHS GMRES is advantageous over BGMRES on the CPU, while the opposite is true for the same situation on the GPU.

While the `Dubcova3` results in Figure 3.2 seem to indicate that BGMRES performs worse on the GPU than on the CPU, one should note that GPU BGMRES is still far faster than the CPU implementation. The high CPU block speedup in this example is driven purely by the extremely slow CPU GMRES runtimes.

Additionally, when restarts are not necessary, our experiments show that we get a clear peak in the Block Speedup on the GPU, and diminished Block Speedups thereafter. In other words, there is an optimal number of RHS for each problem where the block GMRES method is most efficient in comparison to the single RHS GMRES. When we apply our theoretical runtime model to these unrestarted problems using the real number of iterations required for each number of RHS and the realistic GPU hardware parameters $\alpha = 3 \times 10^{-4}$, $\beta = 1.1 \times 10^{-12}$, and $\gamma = 1.3 \times 10^{-13}$, we see similar behavior in our model to what we see in the experiments, as shown in Figures 3.4 and 3.5. This supports our belief that when the number of RHS increases, the benefit of BGMRES over $s \times$ GMRES on the GPU will be determined by weighing computational costs against the cost of accessing the coefficient matrix.

Since our implementation of GMRES-LI does not allow for restarts, all reported experiments with GMRES-LI had to converge before the GPU ran out of memory. This is a significant limitation of GMRES-LI. Although the algorithm is extremely attractive from the standpoint of having the same data movement cost and lower computational complexity than BGMRES, GMRES-LI is severely hindered by the fact that it must store $m \cdot s$ vectors from all of the Krylov subspaces for each RHS simultaneously, but only uses one Krylov subspace per RHS (m vectors) to search for a solution. In contrast, BGMRES must also store $m \cdot s$ vectors in the block Krylov subspace, but gets to leverage all $m \cdot s$ vectors for each RHS to find its solution.

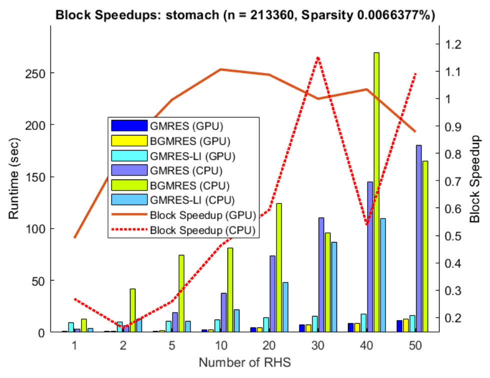
Results for GMRES-LI are shown for cases where the method converged in Figure 3.2. The method failed for experiments using `Dubcova3` with more than 10 RHS and for 50 RHS during experiments on `stomach`. The relative performance of GMRES-LI compared to BGMRES depends greatly on the problem. Our experiments here suggest GMRES-LI is not particularly advantageous for these problems, but this is not always true, as the `cage13` and `thermomech_dM` results indicate. GMRES-LI is most effective when GMRES converges in very few iterations. In this situation, BGMRES can offer little acceleration in terms of iteration counts, and therefore will incur higher computational cost than GMRES-LI while requiring a similar number of iterations to converge. This is exactly the case in the `cage13` and `thermomech_dM` examples, where GMRES requires only 16 and 17 iterations to converge respectively, as shown in Tables 3.9 - 3.12. For this problem, as we increase the number of RHS, BGMRES cannot accelerate convergence very much, so the increasing computational costs eventually force BGMRES to converge slower than GMRES-LI. While GMRES-LI can theoretically be advantageous over BGMRES, our results suggest that on GPUs, BGMRES is still at least as fast as

GMRES-LI, and is noticably faster for fewer than 20 RHS. Since most practical problems use fewer than 20 RHS, we can conclude BGMRES is still generally advantageous on the GPU over GMRES-LI.

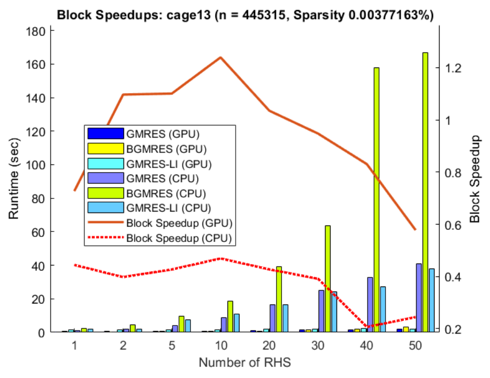
Since BGMRES and GMRES-LI both performed well and were competitive with each other in the `cage13` and `thermomech_dM` problems, we can conclude that the acceleration BGMRES provides in this problem is due mostly to accessing the coefficient matrix less frequently. For the other non-restarted problems this is not the case, indicating the acceleration of BGMRES due to the block Krylov subspace is significant. This is confirmed by the results throughout Section 3.6.

The numerical results for the restarted test problems are shown in Figure 3.3. While these restarted results report very high Block Speedups on the CPU for over 20 RHS, one should note that GPU BGMRES is still far faster than the CPU implementation. The high CPU Block Speedups is driven purely by the extremely slow CPU GMRES runtimes. Moreover, most practical applications will use far fewer than 20 RHS. Thus, BGMRES is still more advantageous on the GPU in the restarted case when restarts are required. Some restarted problems exhibit similar behavior to the non-restarted counterparts, but others exhibit multiple clear peaks in the Block Speedup on the GPU as the number of RHS increase, such as `CoupCons3D`.

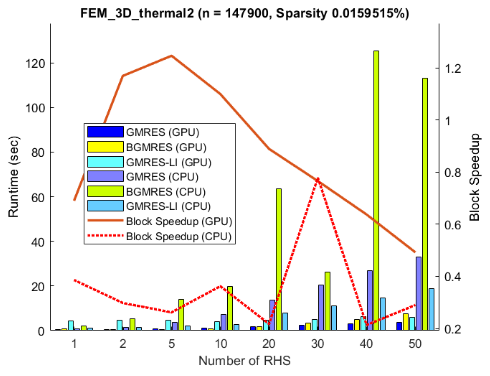
The reason that we get multiple peaks in the Block Speedups for these matrices and not for the unrestarted problems is relatively straightforward: when we do



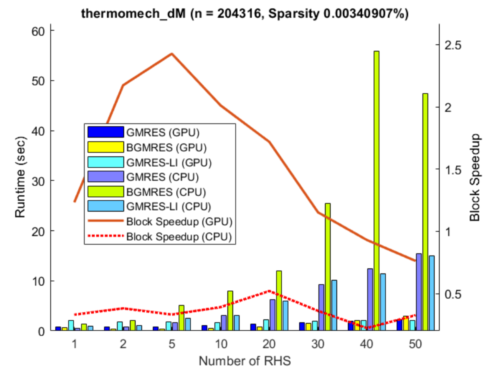
(a) stomach



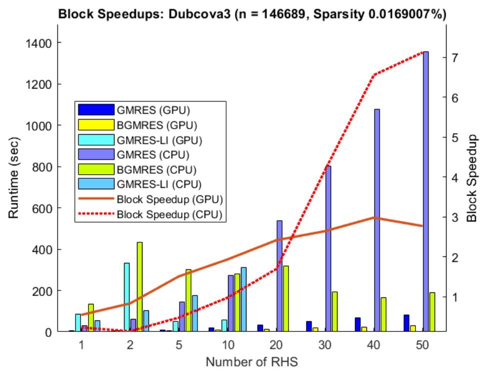
(b) cage13



(c) FEM_3D_thermal2

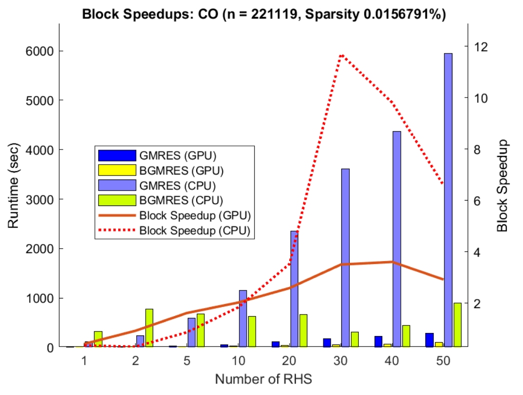


(d) thermomech_dM

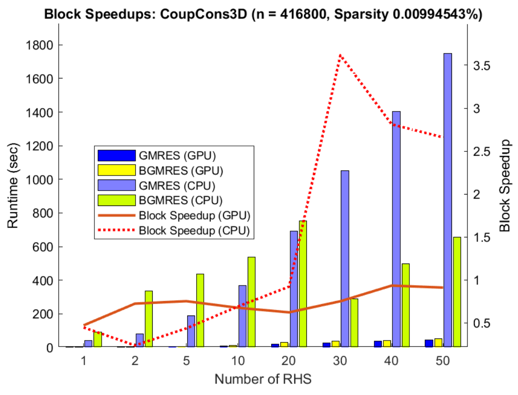


(e) Dubcova3

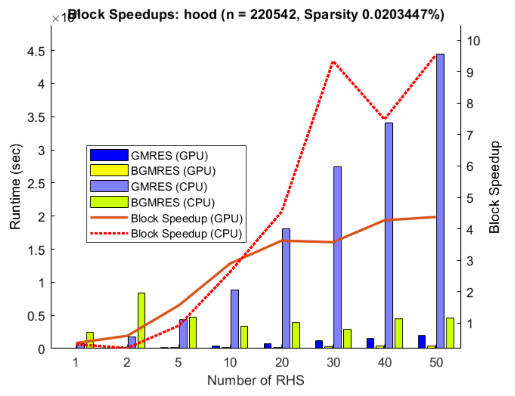
Figure 3.2: Runtimes of the $s \times$ GMRES, BGMRES, and GMRES-LI algorithms on GPUs and CPUs for test problems that do not require restarts, along with the Block Speedup on the CPU and GPU



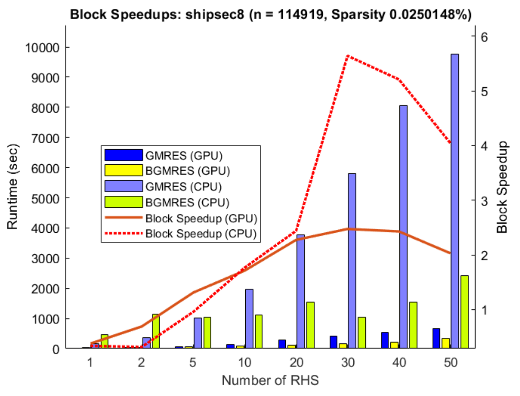
(a) CO



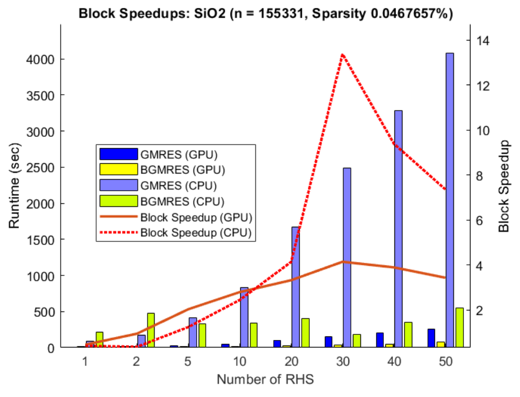
(b) CoupCons3D



(c) hood



(d) shipsec8



(e) SiO2

Figure 3.3: Runtimes of the GMRES and BGMRES algorithms on GPUs and CPUs for test problems requiring restarts, along with the Block Speedup on the CPU and GPU

not restart, increasing the number of RHS will increase the size of the block Krylov subspace, and thus the BGMRES method should converge in fewer iterations. However, each iteration becomes more computationally expensive as the number of RHS increases, and eventually outweighs the advantage of using fewer iterations. Hence, we observe a peak in the Block Speedups in practice for unrestarted problems.

In our experiments, we imposed that the size of the Krylov subspace remains constant in order to satisfy storage constraints on the GPU while offering an equally rich solution space for each number of RHS. Thus, when the number of RHS for the BGMRES method increases, the restart parameter must decrease. Therefore, when restarts are necessary, BGMRES requires more frequent restarting when we increase the number of RHS, making the convergence behavior of BGMRES less regular in terms of iteration counts as the number of RHS increases. For this reason, the iteration counts can increase when we increase the number of RHS for restarted BGMRES. This non-monotone convergence behavior with respect to changes in the number of RHS can lead to multiple peaks in the Block Speedup in the non-restarted case.

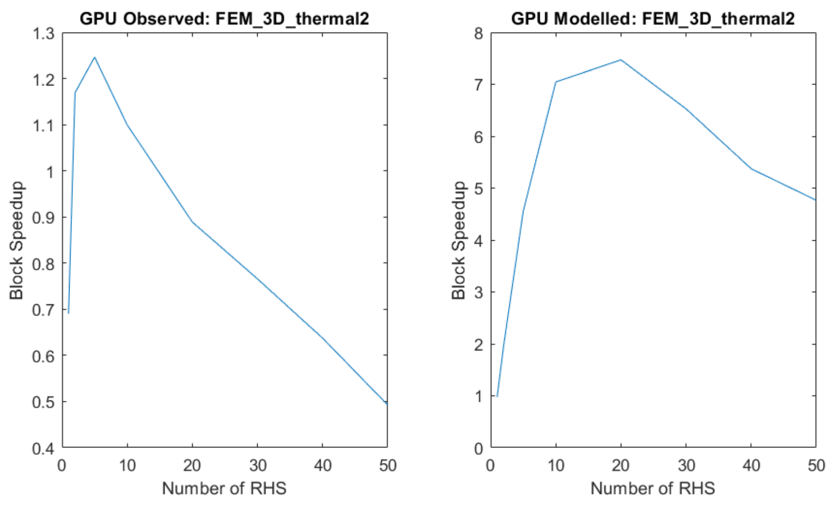
Our theoretical runtime model was not applied to these restarted cases, as it was not designed with restarts in mind, and will therefore not be expected to always capture the possible multiple peaks in the Block Speedups. Moreover, our restarted examples require significantly more iterations to converge than our non-restarted examples; in other words, our examples require many restarts, not just a few. Hence, the runtime model will be almost entirely determined by the cost of the (block) Arnoldi orthogonalization procedure. Since our model does not consider the parallelization benefit of using BLAS3 operations in the block Arnoldi procedure compared to BLAS2 operations in the standard Arnoldi algorithm, the model will severely over-estimate the cost of the block Arnoldi orthogonalization in cases where many restarts are required, such as the cases presented here.

For completeness, tables listing the iteration counts and runtimes for all of our experiments can be found in Section 3.6.

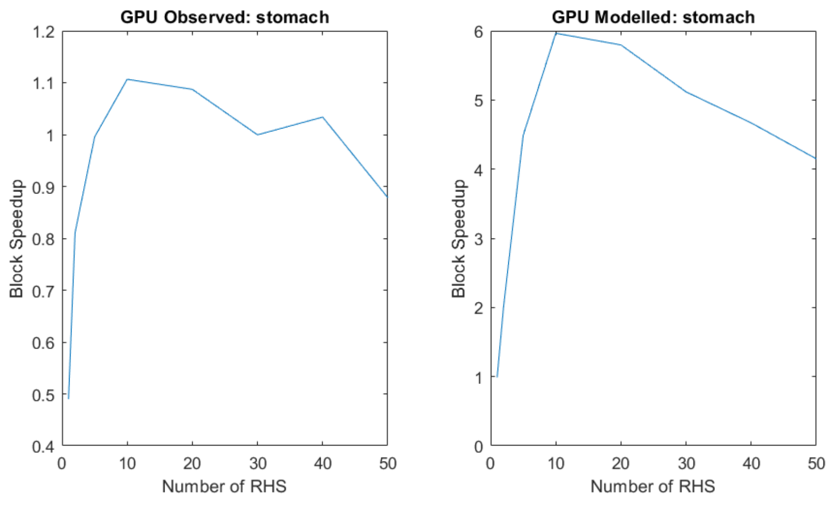
3.6 Conclusions and Outlook

The experiments in Section 3.5 proved our hypothesis that there are many cases where BGMRES converges faster (in runtime) than $s \times$ GMRES on the GPU while the opposite is true on the CPU. Further, our experiments suggest that when we use full BGMRES, i.e., without restarts, there is an optimal number of right hand sides where the BGMRES algorithm is most advantageous over the single RHS GMRES method on the GPU. Our theoretical runtime model described in Section 3.3, which can be used to qualitatively model the Block Speedups for any problem that cannot fit in the GPU's cache, confirms this phenomenon when we use observed iteration counts for unrestarted problems.

To better understand how much of an advantage the block Krylov subspace offers in BGMRES in comparison to the standard Krylov subspaces in GMRES, we also compared BGMRES to GMRES-LI because BGMRES can outperform stan-

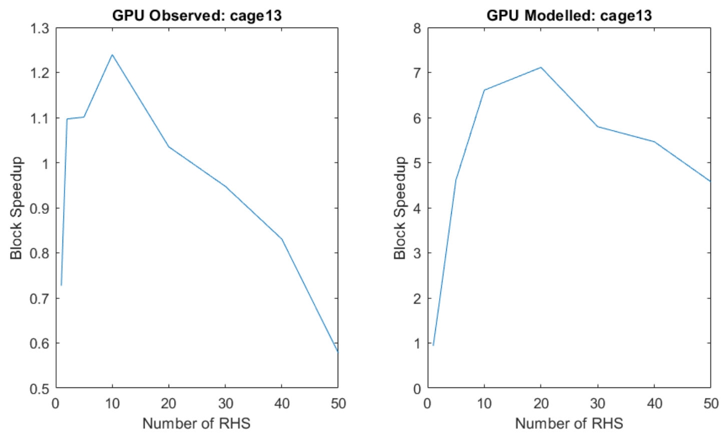


(a) `FEM_3D_thermal2`

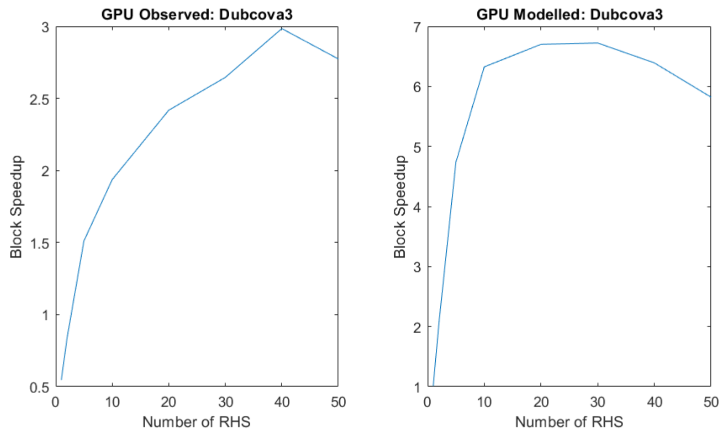


(b) `stomach`

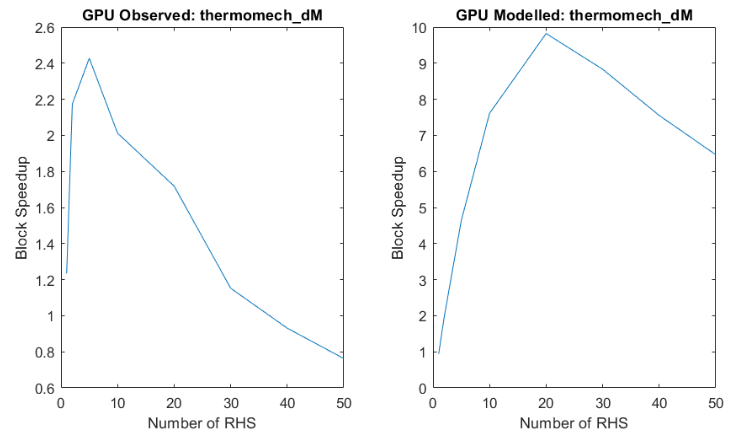
Figure 3.4: Observed vs. modelled Block Speedups on the GPU for the `FEM_3D_thermal2` and `stomach` problems



(a) `cage13`



(b) `Dubcova3`



(c) `thermomech_dM`

Figure 3.5: Observed vs. modelled Block Speedups on the GPU for the `cage13`, `Dubcova3`, and `thermomech_dM` problems

dard GMRES due to reduced iteration counts (and therefore reduced computational cost) from the block Krylov space and/or due to reduced communication costs from accessing the coefficient matrix fewer times.

Comparing BGMRES to GMRES-LI guarantees the dominant communication costs of BGMRES and GMRES are the same. This eliminates any benefits of BGMRES over GMRES due to communication costs, and therefore identifies when the speedups are due to the reduction in iteration counts only.

When GMRES converges slowly, BGMRES is expected to perform well, due to the reduction in iteration counts. This is confirmed on both the CPU and GPU in Figure 3.2 (a), (c), and (e) where BGMRES is noticeably faster than both GMRES-LI and $s \times$ GMRES.

When GMRES converges quickly, BGMRES is often slower than GMRES on CPUs due to higher computational cost and insignificant communication costs. Conversely, GPU memory latency is high, making communication costs very significant. On GPUs, we observed that BGMRES can outperform GMRES even when GMRES converges quickly due to reduced communication costs, which is confirmed by the fact that GMRES-LI also outperforms standard GMRES in these cases; see for example Figure 3.2 (b) and (d).

Figure 3.2 indicates that a direct comparison of BGMRES to GMRES-LI gives mixed results that are entirely dependent on the iteration counts of BGMRES and

GMRES. However, BGMRES on the GPU seems to perform noticeably better for fewer than 20 RHS (which is the case for most practical problems), and BGMRES is often more memory efficient than GMRES-LI in terms of required memory usage to converge. This is a significant upside of BGMRES on GPUs, because GPU memory is often far more limited than CPU memory. Therefore, on the GPU, BGMRES provides benefit over GMRES-LI as well.

Our results apply to the case of other preconditioners. In the computational model, the cost of the combined matrix-vector product with the coefficient matrix and the preconditioned has to be taken into account.

The results and ideas of this work could be used in the future to devise a strategy for partitioning the RHS of systems with many RHS. Specifically, if the optimal performance is estimated to be attained with 20 RHS, one may choose to partition a system of 40 RHS into two sets of 20 RHS.

3.7 BGMRES: Complete Results

In the tables, entries with a '-' indicate the method failed to converge. All times are taken in seconds, and k is the restart parameter. All iteration counts taken for $s \times$ GMRES are the average number of iterations required for each RHS until convergence. Data for GMRES-LI are only included for problems where the method converges for at least $s = 5$ RHS.

We note that BGMRES is often slower than $s \times$ GMRES for s small, e.g., $s = 1$. This is due to different implementations of solving the least squares problem among the two algorithms. BGMRES uses Householder reflections, which is advantageous for multiple RHS, while the Givens rotations in $s \times$ GMRES are designed with minimal effort for one RHS in mind.

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	4060	39	2.15	4130	39.00	0.83	4060	39	1.08
2	2030	38	5.20	4130	39.00	1.55	2030	39	1.35
5	826	37	13.98	4130	39.00	3.66	826	39	2.11
10	413	36	19.69	4130	39.00	7.13	413	39	2.86
20	207	35	63.42	4130	39.00	13.79	207	39	7.85
30	138	34	26.22	4130	39.00	20.41	138	39	11.12
40	103	34	125.21	4130	39.00	26.71	103	39	14.65
50	83	33	113.25	4130	39.00	32.85	83	39	18.70

Table 3.5: `FEM_3D_thermal2` CPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	4060	39	0.70	4130	39.00	0.48	4060	39	4.42
2	2030	38	0.46	4130	39.00	0.54	2030	39	4.60
5	826	37	0.58	4130	39.00	0.72	826	39	4.64
10	413	36	0.93	4130	39.00	1.02	413	39	3.89
20	207	35	1.83	4130	39.00	1.63	207	39	4.71
30	138	34	3.27	4130	39.00	2.50	138	39	5.10
40	103	34	4.84	4130	39.00	3.08	103	39	6.28
50	83	33	7.54	4130	39.00	3.71	83	39	6.08

Table 3.6: `FEM_3D_thermal2` GPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	3710	92	12.86	3780	92.00	3.45	3710	92	3.92
2	1855	85	41.94	3780	91.00	6.80	1855	92	12.54
5	756	75	74.72	3780	91.40	19.31	756	92	10.87
10	378	67	81.53	3780	91.20	37.70	378	92	21.58
20	189	58	124.06	3780	91.30	73.68	189	92	48.24
30	126	53	95.79	3780	91.37	110.50	126	92	86.80
40	95	49	269.24	3780	91.33	144.98	95	-	-
50	76	47	164.87	3780	91.38	180.17	76	-	-

Table 3.7: `stomach` CPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	3710	92	1.25	3780	92.00	0.61	3710	92	9.25
2	1855	85	0.99	3780	91.00	0.80	1855	92	10.12
5	756	75	1.39	3780	91.40	1.38	756	92	11.00
10	378	67	2.25	3780	91.20	2.49	378	92	11.87
20	189	58	4.35	3780	91.30	4.72	189	92	14.24
30	126	53	6.97	3780	91.37	6.97	126	92	15.67
40	95	49	8.64	3780	91.33	8.93	95	-	-
50	76	47	12.69	3780	91.38	11.15	76	-	-

Table 3.8: `stomach` GPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	2660	16	2.39	2660	16.00	1.06	2660	16	1.78
2	1330	16	4.57	2660	16.00	1.82	1330	16	1.96
5	532	14	9.59	2660	16.00	4.09	532	16	7.60
10	266	14	18.44	2660	16.00	8.65	266	16	10.84
20	133	13	38.97	2660	16.00	16.65	133	16	16.48
30	89	13	63.67	2660	16.00	24.91	89	16	24.25
40	67	12	157.97	2660	16.00	32.66	67	16	27.26
50	53	12	166.59	2660	16.00	40.71	53	16	38.08

Table 3.9: `cage13` CPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	2660	16	0.59	2660	16.00	0.43	2660	16	1.40
2	1330	16	0.41	2660	16.00	0.45	1330	16	1.62
5	532	14	0.47	2660	16.00	0.52	532	16	1.66
10	266	14	0.52	2660	16.00	0.64	266	16	1.41
20	133	13	0.84	2660	16.00	0.87	133	16	1.88
30	89	13	1.46	2660	16.00	1.38	89	16	1.89
40	67	12	1.96	2660	16.00	1.63	67	16	2.34
50	53	12	3.23	2660	16.00	1.87	53	16	1.87

Table 3.10: `cage13` GPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	3640	17	1.40	3640	17.00	0.46	3640	17	0.90
2	1820	16	2.04	3640	17.00	0.78	1820	17	1.04
5	728	16	5.05	3640	17.00	1.67	728	17	2.48
10	364	16	7.94	3640	17.00	3.11	364	17	3.07
20	182	15	11.99	3640	17.00	6.25	182	17	6.00
30	121	15	25.41	3640	17.00	9.18	121	17	10.15
40	91	15	55.79	3640	17.00	12.45	91	17	11.45
50	73	15	47.33	3640	17.00	15.44	73	17	15.01

Table 3.11: `thermomech_dM` CPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	3640	17	0.59	3640	17.00	0.72	3640	17	2.06
2	1820	16	0.35	3640	17.00	0.75	1820	17	1.83
5	728	16	0.35	3640	17.00	0.85	728	17	1.81
10	364	16	0.52	3640	17.00	1.04	364	17	1.72
20	182	15	0.79	3640	17.00	1.35	182	17	2.17
30	121	15	1.45	3640	17.00	1.67	121	17	1.97
40	91	15	2.13	3640	17.00	1.98	91	17	2.03
50	73	15	2.99	3640	17.00	2.28	73	17	2.02

Table 3.12: `thermomech_dM` GPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	4850	288	133.91	4850	292.00	30.78	4850	288	54.36
2	4925	257	433.52	4850	290.00	59.28	4925	286	103.69
5	970	192	301.50	4850	287.20	144.30	970	289	175.49
10	485	155	280.85	4850	287.40	274.97	485	288	312.49
20	243	121	317.71	4850	287.15	539.12	243	-	-
30	162	102	192.02	4850	287.13	802.20	162	-	-
40	121	92	164.28	4850	287.45	1077.20	121	-	-
50	97	87	190.16	4850	287.82	1354.82	97	-	-

Table 3.13: `Dubcova3` CPU Results

	BGMRES			$s \times$ GMRES			GMRES-LI		
s	k	Iter.	Time	k	Iter.	Time	k	Iter.	Time
1	4850	288	4.30	4850	292.00	2.35	4850	288	85.75
2	4925	257	5.11	4850	290.00	4.26	4925	286	331.43
5	970	192	6.11	4850	287.20	9.23	970	289	50.92
10	485	155	9.10	4850	287.40	17.62	485	288	59.00
20	243	121	13.90	4850	287.15	33.61	243	-	-
30	162	102	18.77	4850	287.13	49.65	162	-	-
40	121	92	22.20	4850	287.45	66.25	121	-	-
50	97	87	29.87	4850	287.82	82.83	97	-	-

Table 3.14: `Dubcova3` GPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	2700	180	93.41	2700	180.00	41.74
2	1350	174	336.04	2700	180.00	79.76
5	540	166	436.69	2700	180.00	190.50
10	270	160	535.80	2700	180.00	369.46
20	135	157	751.46	2700	180.05	692.62
30	90	154	291.04	2700	180.10	1052.58
40	68	157	499.65	2700	180.07	1404.51
50	54	154	657.33	2700	180.06	1748.62

Table 3.15: `CoupCons3D` CPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	2700	180	2.94	2700	180.00	1.38
2	1350	174	3.12	2700	180.00	2.26
5	540	166	6.51	2700	180.00	4.90
10	270	160	14.10	2700	180.00	9.54
20	135	157	30.35	2700	180.05	18.90
30	90	154	37.29	2700	180.10	28.04
40	68	157	39.69	2700	180.07	37.09
50	54	154	50.83	2700	180.06	46.30

Table 3.16: `CoupCons3D` GPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	4250	566	330.03	4250	564.00	117.02
2	2125	423	778.53	4250	564.00	239.10
5	850	324	677.86	4250	561.60	588.00
10	425	263	630.90	4250	559.10	1159.89
20	213	207	666.33	4250	559.50	2352.35
30	142	203	308.30	4250	559.70	3607.20
40	106	233	445.15	4250	559.40	4368.15
50	85	283	902.32	4250	560.02	5945.07

Table 3.17: `CO` CPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	4250	566	14.96	4250	564.00	6.19
2	2125	423	12.82	4250	564.00	11.88
5	850	324	17.85	4250	561.60	28.89
10	425	263	28.11	4250	559.10	56.97
20	213	207	44.76	4250	559.50	115.97
30	142	203	49.57	4250	559.70	173.79
40	106	233	64.28	4250	559.40	231.99
50	85	283	98.62	4250	560.02	288.14

Table 3.18: `CO` GPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	3640	1477	2491.04	3650	1479.00	798.09
2	1820	1300	8373.21	3650	1479.00	1714.77
5	728	1152	4749.30	3650	1481.60	4357.76
10	364	988	3378.33	3650	1481.10	8893.71
20	182	1155	3972.07	3650	1481.75	18051.77
30	121	1349	2937.16	3650	1481.87	27442.97
40	91	1385	4549.10	3650	1481.92	34054.49
50	73	1335	4655.02	3650	1482.04	44393.21

Table 3.19: `hood` CPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	3640	1477	102.73	3650	1479.00	37.36
2	1820	1300	127.96	3650	1479.00	76.00
5	728	1152	124.34	3650	1481.60	194.15
10	364	988	134.23	3650	1481.10	388.86
20	182	1155	214.43	3650	1481.75	776.50
30	121	1349	327.32	3650	1481.87	1170.12
40	91	1385	365.00	3650	1481.92	1559.29
50	73	1334	444.88	3650	1482.04	1946.72

Table 3.20: `hood` GPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	4850	933	466.12	4850	924.00	155.58
2	2425	774	1140.85	4850	923.50	361.28
5	970	602	1049.17	4850	928.20	1005.80
10	485	569	1117.53	4850	928.20	1974.79
20	243	666	1550.99	4850	930.30	3785.29
30	162	706	1029.54	4850	931.43	5808.20
40	121	825	1548.81	4850	931.65	8067.60
50	97	948	2411.37	4850	931.24	9754.14

Table 3.21: `shipsec8` CPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	4850	933	37.62	9	924.00	14.21
2	2425	774	40.29	9	923.50	27.85
5	970	602	53.16	9	928.20	69.68
10	485	569	81.65	9	928.20	139.84
20	243	666	121.57	9	930.30	276.56
30	162	706	165.54	9	931.43	409.57
40	121	825	223.18	9	931.65	542.04
50	97	948	332.30	9	931.24	674.39

Table 3.22: `shipsec8` GPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	4000	536	214.66	4000	547.00	86.07
2	2000	418	475.26	4000	545.50	169.42
5	800	273	333.45	4000	539.80	411.61
10	400	223	341.84	4000	542.20	834.58
20	200	182	403.50	4000	543.40	1667.57
30	133	178	185.68	4000	540.67	2483.79
40	100	203	349.97	4000	540.10	3284.45
50	80	231	553.80	4000	539.22	4074.85

Table 3.23: `SiO2` CPU Results

	BGMRES			$s \times$ GMRES		
s	k	Iter.	Time	k	Iter.	Time
1	4000	536	13.30	4000	547.00	6.04
2	2000	418	11.75	4000	545.50	11.05
5	800	273	12.80	4000	539.80	25.96
10	400	223	18.32	4000	542.20	51.12
20	200	182	31.01	4000	543.40	102.94
30	133	178	36.79	4000	540.67	152.24
40	100	203	52.07	4000	540.10	202.53
50	80	231	73.84	4000	539.22	253.11

Table 3.24: `SiO2` GPU Results

CHAPTER 4
s-STEP GMRES AND BLOCK ORTHOGONALIZATION SCHEMES

On modern heterogeneous computer architectures, GMRES' performance can be limited by its communication cost to orthonormalize Krylov basis vectors. To address this potential performance bottleneck, its s-step variant orthogonalizes a block of s basis vectors at a time, providing the potential to reduce the communication cost by a factor of s while simultaneously utilizing as many BLAS-3 (and therefore compute-bound) operations as possible. Thus, practical implementations of s-step GMRES give rise to a highly parallelizable GMRES implementation that is amenable to modern heterogeneous–and by extension exascale–machines. In this chapter, we provide the necessary background on s-step GMRES along with its practical challenges to motivate our work in Chapters 5 and 6, which take aim at specific subroutines to improve the s-step method's practicality.

4.1 Introduction

To orthogonalize the new basis vector at each iteration, GMRES (e.g., Algorithm 1 or 2) uses BLAS-1 and BLAS-2 operations, which limit data re-use and require global synchronizations across all parallel processes at each instance. Therefore, GMRES requires several communications per iteration for the basis orthogonalization alone. As mentioned in Chapter 2.4, on modern machines, these communications (e.g., the cost of moving data through the local memory hierarchy and between the processes) can take much longer than the required computation time and can limit the performance of the orthogonalization process. As a result, when efficient and scalable `spmv` and preconditioners are available, orthogonalization becomes a significant part of the iteration time and a performance bottleneck.

4.2 s-Step GMRES

To reduce this performance bottleneck, communication-avoiding (CA) variants of GMRES [12, 37], based on s-step methods [19, 39], were proposed. A highly simplified version of s-step GMRES is given in Algorithm 6. While standard GMRES generates 1 Krylov basis vector per iteration and then orthogonalizes this vector against the prior ones, the idea of s-step GMRES is effectively to form a new block of s Krylov vectors at each iteration, and then orthogonalize in a block-wise fashion to significantly reduce communication requirements and increase usage of BLAS-3 operations during the orthogonalization procedure. Conceptually, this is similar to BGMRES, except that instead of forming a new block vector via one application of a `spmv`, at least s calls of `spmv` are done to form the new block vector so that the algorithm is mathematically equivalent to standard GMRES.

Algorithm 6 s-step GMRES: m iter.

```
1: for $j = 1, \dots, m$ do
2:     $\boldsymbol{V}_j = \text{MPK}(A, q_{j-1}^{(s)}, s)$
3:     $[\boldsymbol{Q}_j, \boldsymbol{H}_{1:j,j}] = \text{BlkOrth}(\boldsymbol{Q}_{1:j-1}, \boldsymbol{V}_j)$
4: end for
5: $y_m = \arg\min \|b - A\boldsymbol{Q}y\|_2$
6: $x_m = x_0 + \boldsymbol{Q}y_m$
```

To generate the orthogonal basis vectors of the Krylov subspace, s-step GMRES utilizes two computational kernels:

1. a Matrix Powers Kernel (MPK) to generate s (non-orthogonal) Krylov vectors by applying `spmv` and preconditioner s times, followed by
2. a Block Orthogonalization Kernel (BlkOrth) that orthogonalizes a block of s basis vectors at once.

While sophisticated communication-avoiding MPK algorithms and preconditioning strategies exist [30, 44, 79], the majority of the communication and computational cost is incurred during the block orthogonalization step, and therefore optimizing the BlkOrth kernel will be the focus of this thesis rather than optimizing the MPK. Moreover, improvements to the BlkOrth kernel will not only affect the performance of s-step GMRES, but any algorithm using a block orthogonalization, like the previously mentioned BGMRES algorithm.

For simplicity, we will assume the MPK is a standard monomial matrix powers algorithm, given in Algorithm 7.

Algorithm 7 Monomial Matrix Powers Kernel: $V = \texttt{MPK}(A, b, s)$

```
1: $v_1 = Ab$
2: **for** $k = 2, \ldots, s$ **do**
3:     $v_k = Av_{k-1}$
4: **end for**
```

4.3 Block Orthogonalization Schemes

While many choices of the block orthogonalization kernel used to orthogonalize a new block $\boldsymbol{V}_j$ against previously orthogonalized blocks $\boldsymbol{Q}_{1:j-1}$ exist, they all consist of two primary components:

1. An "inter-block" orthogonalization, which we refer to as the *inter-ortho* step, which computes some new matrix $\widehat{\boldsymbol{Q}}_j$ that is the result of orthogonalizing $\boldsymbol{V}_j$ against $\boldsymbol{Q}_{1:j-1}$ in a block-wise fashion, and

2. an "intra-block" orthogonalization, which we refer to as the *intra-ortho* step, which orthogonalizes the columns of $\widehat{\boldsymbol{Q}}_j$ against eachother.

It is important to note that for any reasonable choice of the step size s used within s-step GMRES (or number of RHS used within BGMRES), that $\widehat{\boldsymbol{Q}}_j$ will be a *tall-and-skinny matrix*; that is, it has significantly more rows than it has columns. Therefore, the intra-ortho procedure is done using a tall-and-skinny QR factorization.

The inter-ortho procedure, on the other hand, is typically handled by some block Gram-Schmidt procedure, such as block classical Gram-Schmidt (BCGS), block modified Gram-Schmidt (BMGS), or some iterated version of these algorithms. Regardless of the type of block Gram-Schmidt algorithm chosen, all block Gram-Schmidt algorithms rely entirely on BLAS-3 operations and synchronize parallel processes approximately s times fewer times than analogous standard Gram-Schmidt procedures. Thus, s-step GMRES has the potential to reduce the communication cost of orthogonalizing the s basis vectors by a factor of s, provided that the intra-ortho (tall-and-skinny QR) can be done efficiently, making them ideal for modern heterogeneous machines.

4.4 Implementation Issues of s-step GMRES

If s-step GMRES can potentially reduce the communication cost of the Krylov basis orthogonalization by a factor of s, a natural question arises: is there a limit on how large s can be? In short, yes, but precisely how large s can be is usually related to the stability of the inter-ortho and intra-ortho algorithms used in the block orthogonalization.

In general, even if one had an unconditionally stable inter-ortho and intra-ortho, s-step GMRES should still use a step size s so that the MPK produces a new block that is full-rank, otherwise the algorithm performs additional work only to introduce extraneous information into the solution space and is no longer mathematically equivalent to standard GMRES. In exact arithmetic, as long as the coefficient matrix A is full rank, it follows from Proposition 2.4 that the MPK will produce a full rank block $\boldsymbol{V}_j$ as long as $s \leq \text{grade}(q_{j-1}^{(s)})$.

In finite-precision, one must be careful about the notion of “full-rank” matrices used, as there is a limitation on how close to singular a matrix can be before roundoff errors significantly affect the result of the algorithm, and therefore the convergence of s-step GMRES. In the same way a matrix having full-rank in exact arithmetic corresponds to a matrix with all non-zero singular values, in finite precision, "numerically full-rank" matrices are those whose singular values are numerically non-zero relative to each other. Precise definitions of the numerical rank of a matrix, along with equivalent statements of full numerical rank, are given in Definitions 4.1 and 4.2. For clarity, we reiterate that the finite precision unit round-off error is denoted by **u**, as indicated in Table 2.2.

Definition 4.1 (Numerical Rank)**.** *Let* $V \in \mathbb{C}^{n\times m}$*. The* numerical rank *of* V *is the largest integer* r *such that*

$$\sigma_k(V) \geq \max\{n, m\}\ \|V\|_2\, \boldsymbol{u}, \quad \forall k \leq r. \tag{4.1}$$

Definition 4.2 (Numerically Full-Rank)**.** *A matrix* $V \in \mathbb{C}^{n\times m}$ *is* numerically full rank *if the numerical rank of* V *is* $\min\{n, m\}$*. Equivalently,* V *is numerically full rank if,*

$$\max\{n, m\}\ \kappa(V) \leq \boldsymbol{u}^{-1}. \tag{4.2}$$

Thus in practice, even if we have an unconditionally stable inter-ortho and intra-ortho, the MPK within the s-step GMRES algorithm should terminate so that the new block $\boldsymbol{V}_j$ still satisfies $\max\{n, m\}\ \kappa(\boldsymbol{V}_j) \leq \mathbf{u}^{-1}$. However, because $\boldsymbol{V}_j$ is formed by applying s powers of a matrix A to a single vector, the condition number of $\boldsymbol{V}_j$ often becomes too large even when relatively small values of the step size s are used. In practice, a conservatively small step size, like $s \approx 5$ is often used in practice to circumvent this issue, however, reducing the step size s limits the reduction in communication and gains in parallelism that current block orthogonalization schemes can offer.

Additionally, practical tall-and-skinny QR factorizations used in the intra-ortho step are either stable or high-performance on modern machines, but generally not both. More specifically, many modern implementations of s-step GMRES either use a Cholesky-based QR factorization or a HouseholderQR intra-ortho. HouseholderQR is unconditionally stable, but is not highly parallelizable and is communication intensive, making it an impractical choice for modern large-scale machines. The current state-of-the-art Cholesky-based QR algorithms for tall-and-skinny matrices are the CholeskyQR2 and shifted CholeskyQR3 algorithms [27, 28], which are given in Algorithms 9 and 10, respectively. Both algorithms are based on the CholeskyQR algorithm [67], given in Algorithm 8. To simplify the overview of error and stability properties of existing Cholesky-based QR algorithms in this section, we will use the notation $f(V) \lessapprox \mathbf{u}$ and $f(V) \gtrapprox \mathbf{u}$ to mean that there is some function $g(\mathbf{u}) = O(\mathbf{u})$ where $f(V) \leq g(\mathbf{u})$ and $f(V) \geq g(\mathbf{u})$, respectively.

CholeskyQR is a popular building block for high performance tall-and-skinny QR factorizations, because each instance of it requires only one processor synchronization total (to compute the Gram matrix in step 1), along with the fact that it strictly exploits vendor provided highly-optimized dense linear algebra subroutines [2, 47, 48], thereby giving excellent performance on modern heterogeneous machines. It is not used as a standalone scheme for computing a QR factorization often for two reasons. The first is simply because it is numerically unstable, in that when the matrix V has condition number $\kappa(V) \gtrapprox \mathbf{u}^{-1/2}$, the method may fail[1] [77]. The second reason is that even when the algorithm succeeds, it does not compute an accurate orthogonal factor, giving an orthogonality error $\|I - Q^T Q\|_2 \lessapprox \mathbf{u}\, \kappa^2(V)$ [77].

Algorithm 8 Cholesky QR: $[Q, R] = \texttt{cholQR}(V)$

```
   Input:  Matrix $V \in \mathbb{R}^{n,m}$
   Output: Orthogonal factor $Q \in \mathbb{R}^{n,m}$, Triangular factor $R \in \mathbb{R}^{m,m}$ such that
           $QR = V$.
1: Compute Gram matrix $G = V^T V$
2: Perform Cholesky on $G$: $R = \texttt{chol}(G)$
3: Recover orthogonal matrix: $Q = VR^{-1}$
```

Algorithm 9 CholeskyQR2: $[Q, R] = \texttt{cholQR2}(V)$

```
   Input:  Full rank matrix $V \in \mathbb{R}^{n,m}$
   Output: Orthogonal factor $Q \in \mathbb{R}^{n,m}$, Triangular factor $R \in \mathbb{R}^{m,m}$
1: Perform Cholesky QR on $V$: $[Q_0, R_0] = \texttt{cholQR}(V)$
2: Perform Cholesky QR on $Q_0$: $[Q, R_1] = \texttt{cholQR}(Q_0)$
3: Return $R$: $R = R_1 R_0$
```

CholeskyQR2 (Algorithm 9) is designed to remedy the large orthogonality loss incurred by CholeskyQR by re-orthogonalizing the result with another pass of CholeskyQR, while simultaneously providing high performance because the method consists of two passes of CholeskyQR, implying the method only requires two synchroniza-

[1]The intuition behind why this occurs is due to the fact that in Algorithm 8, $\kappa(R) = \kappa(G) = \kappa(V^T V) = \kappa(V)^2$. Numerically, the triangular solve in step 3 is only reliable when R is numerically full-rank [36], or in other words, when $\kappa(V)^2 = \kappa(R) \le m\, \mathbf{u}$. Thus, the method cannot succeed when $\kappa(V) \gtrapprox \mathbf{u}^{-1/2}$. For a rigorous stability analysis explaining all of the details that takes roundoff errors at each step of the algorithm into consideration, see [77].

Algorithm 10 Shifted CholeskyQR3: $[Q, R] = \texttt{sCholQR3}(V)$

```
    Input:    Full rank matrix $V \in \mathbb{R}^{n,m}$, scalar shift $\omega \in \mathbb{R}$
    Output: Orthogonal factor $Q \in \mathbb{R}^{n,m}$, Triangular factor $R \in \mathbb{R}^{m,m}$
1: Compute shifted Gram matrix $G = V^T V + \omega I$
2: Perform Cholesky on $G$: $R_0 = \texttt{chol}(G)$
3: Recover orthogonal matrix $Q_0 = V R_0^{-1}$
4: Perform CholeskyQR2 on $Q_0$: $[Q, R_1] = \texttt{cholQR2}(Q_0)$
5: Return $R$: $R = R_1 R_0$
```

tions total (one per CholeskyQR) and consists only of highly optimized BLAS-3 routines. While CholeskyQR2 produces an accurate orthogonal term Q with $O(\mathbf{u})$ orthogonality error when it succeeds, CholeskyQR2 has the same stability requirements as CholeskyQR, and it therefore may fail when its input matrix V has condition number $\kappa(V) \gtrapprox \mathbf{u}^{-1/2}$ [77]. This is particularly problematic in the context of s-step GMRES, as CholeskyQR2's stability imposition that $\kappa(\boldsymbol{V}_j) \lessapprox \mathbf{u}^{-1/2}$ for each block $\boldsymbol{V}_j$ forces users to select the step size s extremely small, which hurts the overall performance of s-step GMRES.

To combat CholeskyQR2's instability while still maintaining relatively high performance, Shifted CholeskyQR3 (Algorithm 10) was developed. Essentially, Shifted CholeskyQR3 introduces a small diagonal shift ω to $V^T V$ in its first CholeskyQR pass to improve the conditioning of the resulting Gram matrix G, and by extension, the triangular factor R_0 formed in step 2. Shifted CholeskyQR3 provides a significant stability improvement over CholeskyQR2 for appropriately chosen shifts ω, maintaining its stability for input matrices V with $\kappa(V) \lessapprox \mathbf{u}^{-1}$ [27], which is preferable within the context of s-step GMRES. A practical choice of the scalar shift ω that is cheap to compute and guarantees the aforementioned theoretical stability results is $\omega = 11\,(nm + m(m+1))\ \mathbf{u}\ \|V\|_F$ [27].

In spite of the stability gains Shifted CholeskyQR3 introduces, it is still not ideal, as it requires over 50% more computational and communication cost than CholeskyQR2 [27], which can drastically reduce the performance of iterative schemes relying on it, like s-step GMRES. Although more stable communication-avoiding algorithms exist, such as TSQR [21], they rely on Householder QR factorizations, and are often significantly slower than CholeskyQR2 in practice [28].

4.5 Outlook and Motivation

Based on the issues and considerations posed in Sections 4.3 and 4.4, practical s-step GMRES implementations on modern heterogeneous machines need the following:

1. a high-performance, stable tall-and-skinny QR algorithm for the intra-ortho step, and
2. an intra-ortho that is high performance for relatively small step sizes s.

In Chapter 5, we address this first concern by developing and analyzing a novel randomized tall-and-skinny QR algorithm that combines the superior computational and communication costs as CholeskyQR2 with the attractive stability guarantees of Shifted CholeskyQR3. In Chapter 6, we address the second concern by developing and analyzing a novel two-stage block orthogonalization scheme designed to significantly reduce the communication cost of the block orthogonalization for small step sizes s, which translates to sizable performance gains of the s-step GMRES algorithm.

CHAPTER 5

RANDOMIZED HOUSEHOLDER-CHOLESKY QR WITH MULTISKETCHING: A MORE STABLE, HIGH PERFORMANCE TALL-AND-SKINNY QR ALGORITHM

As pointed out in Chapter 4, practical implementations of s-step GMRES, BGMRES, or any iterative numerical method dependent on a block orthogonalization scheme require a high performance tall-and-skinny QR algorithm to execute the intra-ortho step of the block orthogonalizaton process. Additionally, s-step GMRES often forms ill-conditioned blocks, imposing a requirement that the tall-and-skinny QR algorithm it relies on must also be very stable.

CholeskyQR2 and shifted CholeskyQR3 are two state-of-the-art algorithms for computing tall-and-skinny QR factorizations since they attain high performance on current computer architectures. As pointed out in Section 4.4, to guarantee stability, CholeskyQR2 faces a restriction on the condition number of the underlying matrix to factorize that can be prohibitive for many applications, including s-step GMRES. Shifted CholeskyQR3 relaxes this stability requirement but has 50% more computational and communication costs than CholeskyQR2. In this chapter, a randomized QR algorithm called Randomized Householder-Cholesky (`rand_cholQR`) is proposed and analyzed. Using one or two random sketch matrices, it is proved that with high probability, its orthogonality error is bounded by a constant of the order of unit roundoff for any numerically full-rank matrix, and hence it is as stable as shifted CholeskyQR3. An evaluation of the performance of `rand_cholQR` on a NVIDIA A100 GPU demonstrates that for tall-and-skinny matrices, `rand_cholQR` with multiple sketch matrices is nearly as fast as, or in some cases faster than, CholeskyQR2. Hence, compared to CholeskyQR2, `rand_cholQR` is more stable with almost no extra computational or memory cost, and therefore a superior algorithm both in theory and practice.

5.1 Introduction

Computing the QR factorization of tall-and-skinny matrices is a critical component of many scientific and engineering applications, including the solution of least squares problems, block orthogonalization kernels for solving linear systems and eigenvalue problems within block or s-step Krylov methods, dimensionality reduction methods for data analysis like Principal Component Analysis, and many others. Current state-of-the-art QR algorithms for tall-and-skinny matrices are the CholeskyQR2 and shifted CholeskyQR3 algorithms [27, 28], thanks to their communication-avoiding properties along with their exploitation of vendor-provided highly-optimized dense linear algebra subroutines [2, 47, 48]. However, CholeskyQR2 may fail to accurately factorize a matrix V when its condition number $\kappa(V) \gtrapprox \mathbf{u}^{-1/2}$, where $\mathbf{u}$ is unit roundoff [77]. Shifted CholeskyQR3 is numerically stable as long as $\kappa(V) \lessapprox \mathbf{u}^{-1}$, but it requires over 50% more computational and communication cost than CholeskyQR2 [27]. Although more stable communication-avoiding algorithms exist, such as TSQR [21], they rely on Householder QR factorizations, and are often significantly slower than CholeskyQR2 in practice [28].

In this chapter, we present and analyze a randomized algorithm called `randQR` for orthogonalizing the columns of a tall-and-skinny matrix with respect to a specific bilinear form. In order to reduce the cost of the computations, we propose to use "multisketching," i.e., the use of two consecutive sketch matrices, obtaining another algorithm called `rand_cholQR` for computing the QR factorization of a tall-and-skinny matrix V. Our approach is general in the sense that our analysis applies to any two ϵ-subspace embedding sketching matrices (see Section 5.3 for definitions), but what we have in mind is one sparse sketch and one dense sketch, such as a Gaussian or Radamacher sketch [1]. Our analysis applies in particular to Count-Gauss (one application of CountSketch followed by a Gaussian sketch), as described in [40, 63, 64].

We prove that with high probability, the orthogonality error of `rand_cholQR` is bounded by a constant of the order of unit roundoff for any numerically full-rank matrix V, and hence it is as stable as shifted CholeskyQR3 and it is significantly more numerically stable than CholeskyQR2. Our numerical experiments ilustrate the theoretical results. In addition, the `rand_cholQR` algorithm may be implemented using the same basic linear algebra kernels as CholeskyQR2. Therefore, it is simple to implement and has the same communication-avoiding properties. We perform a computational study on a state-of-the-art GPU to demonstrate that `rand_cholQR` can perform up to 4% faster than CholeskyQR2 and 56.6% faster than shifted CholeskyQR3, while significantly improving the robustness of CholeskyQR2.

In summary, our primary contribution consists of a new error analysis of a multisketched randomized QR algorithm, proving it can be safely used for matrices of larger condition number than CholeskyQR2 can handle. This analysis applies in particular to the case of one sketch, improving upon the existing results. Our implementation confirms and illustrates the theory developed in this chapter.

In Section 5.2, we begin by discussing prior work on similar topics. Then, in Section 5.3, we present some preliminary definitions and known results from randomized linear algebra relevant to this work. We follow with Section 5.4, where we present multisketching on a conceptual level, and how to incorporate it into a randomized QR factorization (`rand_cholQR`). We also discuss performance considerations for `rand_cholQR` compared to other high performance tall-skinny QR algorithms, leading to the motivation as to why multisketching is recommended. In Section 5.5, we present rigorous error bounds and their proofs for the proposed multisketched `rand_cholQR`. These bounds can also be applied to the case of a single sketch matrix, and we compare the new results to those available in the literature. Numerical experiments are presented in Section 5.6, followed by our conclusions.

5.2 Related Work

In the case of a single sketch matrix, the concept of sketching a tall-and-skinny matrix, computing its QR factorization, and then preconditioning the matrix with the resulting triangular factor like `randQR` is not new. The earliest appearance of such an algorithm was by Rokhlin and Tygert in 2008 [57] for solving overdetermined least squares problems, where they proposed a version of `randQR` with a column-pivoted QR factorization and a single subsampled randomized Hadamard transform sketch.

While this thesis was being written, Balabanov proposed the “RCholeskyQR” and “RCholeskyQR2” methods in an unpublished manuscript [6], which are identical to what we refer to as `randQR` and `rand_cholQR`, respectively, in the case of a single (ε, d, m) oblivious ℓ_2-subspace embedding, and gave stability results similar to Corollary 5.18 of this chapter. However, our results differ from Balabanov’s, as ours impose no assumptions on the level of accuracy performed by subroutines within the algorithm, meticulously deriving all bounds from existing roundoff error analysis of each subroutine. Additionally, Balabanov’s work imposes a far stricter limit on the subspace embedding parameter $\epsilon \leq \frac{1}{2}$, while ours provides analysis up to $\epsilon < \frac{616}{634}$ for a (ε, d, m) oblivious ℓ_2-subspace embedding, which is nearly the theoretical upper limit of $\epsilon < 1$ imposed by the theory in Section 5.3. This is significant, because stability guarantees for larger values of ϵ ensure high accuracy with smaller sketch matrices, resulting in a more computationally efficient algorithm.

Our results extend beyond a single (ε, d, m) oblivious ℓ_2-subspace embedding, and cover the more generalized case of two subspace embeddings (i.e., multisketch). Also, our work includes explicit analysis of the S_2S_1-orthogonality error of `randQR`, which is a specific notion of orthogonality with respect to a sketched inner product, and the loss of orthogonality error in the standard Euclidean inner product of `rand_cholQR`.

Our work is novel in several ways. To our knowledge, this work is the first to propose a randomized QR algorithm with multiple sketches. The stability results in this chapter improve upon and expand the existing stability analysis of `randQR` and `rand_cholQR`, and considers the multisketch case for the first time. Additionally, our experimental results are the first to demonstrate the performance of `rand_cholQR` in a parallel heterogeneous computing environment under any sketching framework, particularly in the multisketch case which allows the algorithm to sometimes run faster than the widely used high-performance `cholQR2` algorithm. This tangibly demonstrates the potential of the multisketch `rand_cholQR` in exascale applications.

5.3 Preliminaries on Random Sketching

Suppose one would like to compress $V \in \mathbb{R}^{n\times m}$ into a matrix with fewer rows with nearly the same norm. We denote the *sketch matrix* by $S \in \mathbb{R}^{p\times n}$ for $p \ll n$. The sketch matrix is typically chosen to be a *ϵ-subspace embedding*, or a linear map to a lower dimensional space that preserves ℓ_2-inner products and norms of all vectors within the subspace up to a factor of $\sqrt{1 \pm \varepsilon}$ for $\varepsilon \in [0, 1)$ [7, 45, 60].

Definition 5.1 (ε-subspace embedding)**.** *Given $\varepsilon \in [0, 1)$, the sketch matrix $S \in \mathbb{R}^{p \times n}$ is an ε-*subspace embedding *for the subspace $\mathcal{V} \subset \mathbb{R}^n$ if $\forall x, y \in \mathcal{V}$,*

$$|\langle x, y\rangle - \langle Sx, Sy\rangle| \leq \varepsilon \|x\|_2 \|y\|_2,$$

where $\langle \cdot, \cdot \rangle$ is the Euclidean inner product.

Proposition 5.2. *If the sketch matrix $S \in \mathbb{R}^{p \times n}$ is an ε-*subspace embedding *for the subspace $\mathcal{V} \subset \mathbb{R}^n$, then $\forall x \in \mathcal{V}$,*

$$\sqrt{1-\varepsilon}\, \|x\|_2 \leq \|Sx\|_2 \leq \sqrt{1+\varepsilon}\, \|x\|_2. \tag{5.1}$$

Corollary 5.3. *If the sketch matrix $S \in \mathbb{R}^{p \times n}$ is an ε-*subspace embedding *for the subspace $\mathcal{V} \subset \mathbb{R}^n$, and V is a matrix whose columns form a basis of $\mathcal{V}$, then*

$$\sqrt{1-\varepsilon}\|V\|_2 \leq \|SV\|_2 \leq \sqrt{1+\varepsilon}\|V\|_2, \tag{5.2}$$

$$\sqrt{1-\varepsilon}\|V\|_F \leq \|SV\|_F \leq \sqrt{1+\varepsilon}\|V\|_F. \tag{5.3}$$

Corollary 5.3 is a simple consequence of Proposition 5.2 using the definition of the ℓ_2 and Frobenius matrix norms that gives us a way to bound the norms of a sketched matrix.

Proposition 5.4. *If the sketch matrix $S \in \mathbb{R}^{p\times n}$ is an ε-*subspace embedding *for the subspace $\mathcal{V} \subset \mathbb{R}^n$, and V is a matrix whose columns form a basis of $\mathcal{V}$, then*

$$(1+\varepsilon)^{-1/2}\,\sigma_{min}(SV) \leq \sigma_{min}(V) \leq \sigma_{max}(V) \leq (1-\varepsilon)^{-1/2}\,\sigma_{max}(SV). \quad (5.4)$$

Thus,

$$\kappa(V) \leq \sqrt{\frac{1-\varepsilon}{1+\varepsilon}}\,\kappa(SV). \quad (5.5)$$

A proof of Proposition 5.4 is given in [7], and it implies that the singular values of V are bounded by those of SV. Hence if SV is well conditioned, then so is V.

While ε-subspace embeddings require knowledge of the subspace $\mathcal{V} \subset \mathbb{R}^n$ a priori, (ε, d, m) *oblivious ℓ_2-subspace embeddings* do not [7].

Definition 5.5 ((ε, d, m) oblivious ℓ_2-subspace embedding)**.** *$S \in \mathbb{R}^{p\times n}$ is an (ε, d, m)* oblivious ℓ_2-subspace embedding *if it is an ε-subspace embedding for any fixed m-dimensional subspace $\mathcal{V} \subset \mathbb{R}^n$ with probability at least $1-d$.*

An example of a (ε, d, m) oblivious ℓ_2-subspace embedding is $S = \frac{1}{\sqrt{s}}G$ for a fully dense Gaussian matrix $G \in \mathbb{R}^{p\times n}$ and

$$s = \Omega(\varepsilon^{-2}\log m \log(1/d));$$

see, e.g., [60]. Sparse (ε, d, m) oblivious ℓ_2-subspace embeddings exist, including CountSketch, which consists of a single ± 1 per column, where the row storing the entry and its sign are chosen uniformly at random [15, 76]. In order to be a (ε, d, m) oblivious ℓ_2-subspace embedding, the number of columns of the CountSketch matrix must satisfy

$$s \geq \frac{m^2 + m}{\varepsilon^2 d} \; ; \tag{5.6}$$

see [43]. Other popular (ε, d, m) oblivious ℓ_2-subspace embeddings include subsampled randomized Hadamard and Fourier transforms, and "sparse dimension reduction maps" [7, 45], though obtaining high performance with these is difficult, and the complexity of applying them is higher than CountSketch. We do not consider such embeddings in this chapter.

5.4 Multisketching

Next, we consider the case of applying two sketch matrices one after the other, which is what we refer to as "multisketching" in this chapter, generalizing the approach of [40, 64], where one application of a large sparse CountSketch is followed by a smaller dense Gaussian sketch. In these references though, there is no analysis of stability, as we do here. The main motivation for this approach is to be able to apply the dense Gaussian sketch to a smaller matrix, obtained after the application of a sparse sketch, thus obtaining similar good results at a fraction of the cost; see more details on this motivation in Section 5.4.2.

We first present the algorithm `randQR` using this multisketching approach, and then prove bounds similar to those in Proposition 5.4 for the case of two sketches.

Let $V \in \mathbb{R}^{n\times m}$, and suppose $S_1 \in \mathbb{R}^{p_1\times n}$ and $S_2 \in \mathbb{R}^{p_2\times p_1}$ are (ε_1, d_1, m) and (ε_2, d_2, m) oblivious ℓ_2-subspace embeddings, respectively. Let $d = d_1 + d_2 - d_1 d_2$, so that $1 - d = (1 - d_1)(1 - d_2)$. We define the Randomized Householder QR algorithm (`randQR`) in Algorithm 11, where we use MATLAB function call notation.

Algorithm 11 Randomized Householder QR: $[Q, R] =$ `randQR`(V, S_1, S_2)

```
    Input:   Matrix $V \in \mathbb{R}^{n\times m}$, sketch matrices $S_1 \in \mathbb{R}^{p_1\times n}$, $S_2 \in \mathbb{R}^{p_2\times p_1}$
    Output:  $S_2S_1$-Orthogonal factor $Q \in \mathbb{R}^{n\times m}$, Triangular factor $R \in \mathbb{R}^{m\times m}$
    such that $QR = V$.
1:  Apply sketches $W = S_2S_1V$
2:  Perform Householder QR: $[Q_{tmp}, R] = \texttt{hhqr}(W)$
3:  Recover $S_2S_1$-orthogonal matrix: $Q = VR^{-1}$
```

Remark 5.6. In exact arithmetic, provided that $V \in \mathbb{R}^{n\times m}$ is full rank, then `randQR` produces a matrix Q that is S_2S_1-orthogonal[1] with probability at least $1 - d$; i.e., it satisfies $(S_2S_1Q)^T(S_2S_1Q) = I$, because

$$S_2S_1Q = S_2S_1VR^{-1} = WR^{-1} = Q_{tmp},$$

where Q_{tmp} is the orthogonal factor produced by the Householder QR factorization of $W = S_2S_1V$. Observe that Q being S_2S_1-orthogonal is equivalent to being an orthonormal matrix with respect to the inner product[2] $\langle S_2S_1\cdot, S_2S_1\cdot\rangle$. Unlike traditional Householder QR, even in exact arithmetic V must have full rank, since step 3 of Algorithm 11 requires rank(V) = rank$(R) = m$. In finite precision, intuition

[1] In exact arithmetic, Q will only fail to be S_2S_1-orthogonal if $V \in$ null(S_2S_1), which by Proposition 5.4, should only happen with probability at most d.

[2] Although $\langle S_2S_1\cdot, S_2S_1\cdot\rangle$ is not an inner product over the traditional vector space $\mathbb{R}^{n\times m}$, it is an inner product over the complement of null(S_2S_1).

suggests that an inevitable requirement of `randQR` is that V must be numerically full rank.

Next, we introduce some convenient norm, singular value, and condition number inequalities when one uses the multisketching approach with two oblivious ℓ_2 subspace embeddings.

Proposition 5.7. *Let $S_1 \in \mathbb{R}^{p_1 \times n}$ be a (ε_1, d_1, m) oblivious ℓ_2-subspace embedding in $\mathbb{R}^n$, $S_2 \in \mathbb{R}^{p_2 \times p_1}$ be a (ε_2, d_2, m) oblivious ℓ_2-subspace embedding in $\mathbb{R}^{p_1}$, generated independently. Let $\varepsilon_L = \varepsilon_1 + \varepsilon_2 - \varepsilon_1\varepsilon_2$, $\varepsilon_H = \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2$, and $d = d_1 + d_2 - d_1 d_2$. Then for any m-dimensional subspace $\mathcal{V} \subset \mathbb{R}^n$ and $\forall x \in \mathcal{V}$,*

$$\sqrt{1-\varepsilon_L}\,\|x\|_2 \leq \|S_2 S_1 x\|_2 \leq \sqrt{1+\varepsilon_H}\,\|x\|_2, \tag{5.7}$$

with probability at least $1 - d$.

Proof. Let $x \in \mathcal{V}$. By assumption, S_2 is a (ε_2, d_2, m) oblivious ℓ_2-subspace embedding, and thus it is an ε_2-subspace embedding of $S_1\mathcal{V} \in \mathbb{R}^{p_1}$ with probability at least $1 - d_2$. Observe that $S_1 x \in S_1\mathcal{V}$. Therefore, by (5.1),

$$\sqrt{1-\varepsilon_2}\|S_1 x\|_2 \leq \|S_2 S_1 x\|_2 \leq \sqrt{1+\varepsilon_2}\|S_1 x\|_2,$$

with probability at least $1 - d_2$, because this is the probability at which (5.1) holds.

Again, by assumption, S_1 is a (ε_1, d_1, m) oblivious ℓ_2-subspace embedding, and thus it is an ε_2-subspace embedding of $\mathcal{V} \in \mathbb{R}^m$ with probability at least $1-d_1$. Now, using (5.1) again for S_1 and ϵ_1, we have

$$\sqrt{1-\varepsilon_1}\|x\|_2 \leq \|S_1 x\|_2 \leq \sqrt{1+\varepsilon_1}\|x\|_2,$$

with probability at least $1 - d_1$.

Combining these results, we find that

$$\begin{aligned}
\sqrt{1-(\varepsilon_1+\varepsilon_2-\varepsilon_1\varepsilon_2)}\|x\|_2 = \sqrt{(1-\varepsilon_2)(1-\varepsilon_1)}\|x\|_2 &\leq \sqrt{1-\varepsilon_2}\|S_1 x\|_2 \\
&\leq \|S_2 S_1 x\|_2 \leq \sqrt{1+\varepsilon_2}\|S_1 x\|_2 \\
&\leq \sqrt{(1+\varepsilon_2)(1+\varepsilon_1)}\|x\|_2 \\
&= \sqrt{1+(\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2)}\|x\|_2
\end{aligned}$$

with probability at least $(1-d_1)(1-d_2) = 1-(d_1+d_2-d_1 d_2)$.

Proving d and consequently $1-d$ are between $[0,1]$ is equivalent to showing $p(d_1, d_2) = d_1 + d_2 - d_1 d_2 \in [0,1]$ for any $(d_1, d_2) \in [0,1]^2$. This is straightforward, as on the boundaries, $p(0, d_2) = d_2 \in [0,1]$, $p(d_1, 0) = d_1 \in [0,1]$, $p(1, d_2) = p(d_1, 1) = 1 \in [0,1]$, and $\nabla p \geq 0$ on $[0,1]^2$, and therefore $p(d_1, d_2)$ cannot go below 0 or above 1. $\square$

If S_1, S_2 are $\varepsilon_1, \varepsilon_2$ embeddings respectively, then by Propositions 5.4 and 5.7 along with Corollary 5.3,

$$\sqrt{1-\varepsilon_L}\|V\|_2 \leq \|S_2S_1V\|_2 \leq \sqrt{1+\varepsilon_H}\|V\|_2, \tag{5.8}$$

$$\sqrt{1-\varepsilon_L}\|V\|_F \leq \|S_2S_1V\|_F \leq \sqrt{1+\varepsilon_H}\|V\|_F. \tag{5.9}$$

$$\begin{aligned}(1+\varepsilon_H)^{-1/2}\,\sigma_{min}(S_2S_1V) \leq \sigma_{min}(V) \leq \sigma_{max}(V)\\ \leq (1-\varepsilon_L)^{-1/2}\,\sigma_{max}(S_2S_1V),\end{aligned} \tag{5.10}$$

and so,

$$\kappa(V) \leq \sqrt{\frac{1-\varepsilon_L}{1+\varepsilon_H}}\,\kappa(S_2S_1V). \tag{5.11}$$

Remark 5.8. By Remark 5.6, in exact arithmetic, the Q factor computed by `randQR` is S_2S_1-orthogonal, and therefore, by (5.11),

$$\kappa(Q) \leq \sqrt{\frac{1-\varepsilon_L}{1+\varepsilon_H}}\,\kappa(S_2S_1Q) = \sqrt{\frac{1-\varepsilon_L}{1+\varepsilon_H}} = O(1)\ . \tag{5.12}$$

Thus, `randQR` serves well for applications where a well-conditioned set of vectors is sufficient, or as a pre-processing algorithm for less stable orthogonalization schemes.

5.4.1 Algorithms and Performance Considerations

We introduce the main algorithm of interest for this chapter, `rand_cholQR` (Algorithm 12). In this chapter, `randQR` is strictly used to precondition the tall-and-skinny matrix V as a pre-processing step for `rand_cholQR`, which is a true orthogonalization scheme. As a proof of concept for `rand_cholQR`, in step 1, the algorithm computes a S_2S_1-orthogonal factor Q_0 from `randQR` (Algorithm 11). By Remark 5.8, in exact arithmetic, $\kappa(Q_0) = O(1)$. In step 2 of `rand_cholQR`, Q is computed by re-orthogonalizing Q_0 using Cholesky QR (`cholQR`, Algorithm 8 given in Section 4.4). Since $\kappa(Q_0) = O(1)$, one can expect the resulting Q satisfies $\|Q^TQ - I\|_2 = O(\mathbf{u})$ using the roundoff error analysis of `cholQR` [77].

Algorithm 12 Rand. Householder-Cholesky: $[Q, R]$ = `rand_cholQR`(V, S_1, S_2)

```
   Input:  Matrix $V \in \mathbb{R}^{n\times m}$, sketch matrices $S_1 \in \mathbb{R}^{p_1\times n}$, $S_2 \in \mathbb{R}^{p_2\times p_1}$
   Output: Orthogonal factor $Q \in \mathbb{R}^{n\times m}$, Triangular factor $R \in \mathbb{R}^{m\times m}$ such that
   $QR = V$.
1: Recover $S_2S_1$-orthogonal matrix $Q_0$: $[Q_0, R_0]$ = randQR$(V, S_1, S_2)$
2: Perform Cholesky QR on $Q_0$: $[Q, R_1]$ = cholQR$(Q_0)$
3: Return $R$: $R = R_1R_0$
```

To examine the expected performance of `randQR` and `rand_cholQR`, we first discuss their communication costs compared to `cholQR` and `cholQR2` respectively, and then analyze their arithmetic costs. The most computationally intensive parts of `randQR` (steps 1 and 3) are nearly identical to those of `cholQR`, in the sense that both perform a product of tall and skinny matrices, followed by a triangular solve of a tall and skinny matrix. Similar to the way `cholQR` (Algorithm 8, given in Section 4.4) requires only one processor synchronization total to compute the Gram matrix in step 1, `randQR` only requires one synchronization total to compute W in step 1 provided $m \leq p_2 \leq p_1 \ll n$ and therefore the algorithms incur the same number of processor synchronizations[3]. Moreover, `rand_cholQR` and `cholQR2` simply build on these algorithms, adding passes of `cholQR` to matrices of the same size for both algorithms. Thus, like `cholQR2`, `rand_cholQR` only requires two synchronizations total.

The computational cost of step 2 of `randQR` (Algorithm 11) is negligible compared to steps 1 and 3, since $W \in \mathbb{R}^{p_2 \times m}$ with $p_2 \ll n$. The arithmetic cost of step 1 is dependent on the type of sketch matrices used. Suppose one replaces $S_2 S_1$ with a single dense Gaussian sketch matrix $S \in \mathbb{R}^{p \times n}$, which is conceptually simple, very efficient in parallel, but computationally expensive since it is fully dense. Then the arithmetic cost of `randQR` and `rand_cholQR` (in FLOPs) are:

$$\texttt{randQR}\text{ FLOPs: } \underbrace{pm(2n-1)}_{\text{Sketching}} + \underbrace{2pm^2 - \frac{2}{3}m^3}_{\text{Householder QR}} + \underbrace{nm^2}_{\text{Tri. solve}} \approx 2nmp + nm^2.$$

$$\texttt{rand\_cholQR}\text{ FLOPs: } \underbrace{2nmp + nm^2}_{\texttt{randQR}} + \underbrace{2nm^2}_{\texttt{cholQR}} + \underbrace{m^2(2m-1)}_{\text{Matrix mult.}} \approx 2nmp + 3nm^2.$$

Provided that $p = O(m)$, e.g., $p \approx 2m$, then `rand_cholQR` FLOPs $\approx 7nm^2$.

[3]Specifically, suppose one has p parallel processes and $m \leq p_2 \leq p_1 \ll n$ so that $S_2 \in \mathbb{R}^{p_2 \times p_1}$ can be stored locally on each process. One can distribute block row partitions of $V = [V_1^T, \ldots, V_p^T]^T$ and block column partitions of the larger sketch $S_1 = [(S_1)_1, \ldots, (S_1)_p]$ to each of the processes, along with the entire small sketch S_2 to each process. Then on process k, one computes $W_k = S_2(S_1)_k V_k$, and then one synchronizes the processes to compute $W = S_2 S_1 V = \sum_{k=1}^{p} W_k$ in a single reduction.

In contrast, CholeskyQR2 (`cholQR2`), which is explicitly shown in Algorithm 9 (Section 4.4), incurs a cost of

$$\texttt{cholQR2}\text{ FLOPs: } \underbrace{2nm^2}_{\texttt{cholQR}} + \underbrace{2nm^2}_{\texttt{cholQR}} + \underbrace{m^2(2m-1)}_{\text{Matrix mult.}} \approx 4nm^2.$$

Thus, `randQR` (using a dense Gaussian sketch) and `cholQR` have about the same asymptotic arithmetic costs. Because the two algorithms have the same communication costs and `rand_cholQR` has a slightly higher arithmetic cost, in a large scale parallel setting, one can expect `rand_cholQR` to run slightly slower but on the same order of runtime as `cholQR2` and scale in the same way. However, as we show in Section 5.5.3, `rand_cholQR` is significantly more stable with high probability.

5.4.2 Motivation for Multisketching

Using a single Gaussian sketch requires a dense matrix-matrix multiply with a sketch matrix S of dimension $p \times n$. In addition to performing $O(nmp)$ FLOPs to apply this sketch, we need to store and load this fully dense $p \times n$ sketch matrix. As shown in Section 5.4.1, the time to sketch the matrix with the dense Gaussian can dominate the total factorization time for `randQR` and consequently `rand_cholQR`.

One can reduce the sketching cost using a sparse sketch such as a CountSketch matrix [15]. Since the CountSketch matrix has only one non-zero per column, the cost of applying the CountSketch matrix to $V \in \mathbb{R}^{n\times m}$ is only $O(nm)$, and it only requires to store $O(n)$ numerical values. Additionally, CountSketch can be implemented using the sparse-matrix multiple-vector multiply (SpMM), whose optimized implementation is often available on specific architectures. A clever implementation can exploit the fact that applying the CountSketch matrix is equivalent to adding/subtracting subsets of rows of V, and can therefore be parallelized well using batched BLAS-1 kernels or a highly-optimized sparse linear algebra library. Hence, CountSketch could obtain high performance using only readily available linear algebra libraries. However, a CountSketch matrix requires $p = O(m^2)$ to maintain the ε-embedding properties, so one is left to factorize $W \in \mathbb{R}^{p\times m}$ with Householder QR, which incurs $O(m^4)$ FLOPs. In contrast, the Gaussian sketch ensures that S is an ε-subspace embedding with $p = O(m)$, meaning the cost of the Householder QR factorization is only $O(m^3)$ FLOPs. Householder QR imposes high communication costs and does not parallelize well [21]. As a result, on current computers, it obtains much lower performance than the BLAS-3 operations like the dense matrix product (`gemm`), and these $O(m^4)$ FLOPs for Householder QR become a performance bottleneck for sufficiently large m.

Ideally, we want an embedding that offers low computational and storage costs like CountSketch, while returning a sketched matrix $W \in \mathbb{R}^{p\times m}$ with $p = O(m)$ like the Gaussian sketch does, to avoid a performance bottleneck from Householder QR. This is possible by using the multisketching framework with first a sparse

CountSketch and then a Gaussian sketch. To see this, suppose $S_1 \in \mathbb{R}^{p_1 \times n}$ is a CountSketch matrix with $p_1 = \frac{m^2+m}{\varepsilon_1^2 d_1}$, cf. (5.6), and suppose $S_2 \in \mathbb{R}^{p_2 \times p_1}$ is a Gaussian sketch where $p_2 = 2m$.

We split the computation of $W = S_2 S_1 V$ into two steps: first computing $W_1 = S_1 V$, then $W = S_2 W_1$. Storing S_1 only requires $O(n)$ bytes of memory, and the sparse matrix product $W_1 = S_1 V$ costs $O(nm)$ FLOPs. The cost to compute $W = S_2 W_1$ costs $O(m^4)$ FLOPs, but since the dense matrix product (`gemm`) obtains much higher performance than the Householder QR, this cost became negligible in our performance studies with a GPU. The storage of S_2 only requires $O(m^3)$ bytes of memory, and the Householder QR factorization of the $O(m) \times m$ matrix W incurs negligble computational cost as well.

Moreover, the $O(nm + m^4)$ total FLOPs incurred using the multisketch framework can actually be lower than the $O(nm^2)$ FLOPs required to perform `cholQR`, making `rand_cholQR` sometimes cheaper than `cholQR2` under the multisketch framework while incurring the same number of communications (as discussed in Section 5.4.1). Thus, the multisketch framework provides an avenue for an extremely efficient, stable QR factorization that can potentially outperform `cholQR2` in terms of both stability and practical speed on modern parallel machines.

5.5 Error Analysis of `randQR` and `rand_cholQR`

Here, we present the main results of this work on theoretical properties (with high probability) of $\hat{Q}$ and $\hat{R}$ computed by `randQR` and `rand_cholQR`. The structure of this section is as follows. First, we highlight the sources of floating point error of the `randQR` algorithm in Section 5.5.1. Then, in Section 5.5.2, we introduce our assumptions for the proofs and some preparatory results for the error analysis.

We identify which results are probabilistic, and explicitly state the necessary assumptions and some useful initial consequences in Sections 5.5.2.1–5.5.2.2. In Section 5.5.2.3, we identify bounds on $\|E_1\|_2$, where E_1 (given in (5.14)) is the forward error in the matrix-matrix multiplication while sketching. In Sections 5.5.2.4–5.5.2.7, we formulate the remaining error analysis in terms of $\|E_1\|_2$.

Finally, in Section 5.5.3, we will provide the key theorems on the stability and accuracy of our `randQR` and `rand_cholQR` algorithms, and prove them primarily through the preparatory results from Section 5.5.2.

5.5.1 Sources of Floating Point Error in `randQR`

We use a hat to denote a computed version of each of the matrix in all algorithms. First, errors are incurred when performing the matrix products $W = S_2S_1V$ in step 1 of Algorithm 11. Specifically, there exist error terms $\Delta\hat{W}_1$, $\Delta\hat{W}$ such that

$$\begin{aligned} \hat{W}_1 &= S_1V + \Delta\hat{W}_1, \\ \hat{W} &= S_2\hat{W}_1 + \Delta\hat{W} \cdot \end{aligned} \tag{5.13}$$

We can group these error terms together so that the computed $\hat{W}$ satisfies

$$\hat{W} = S_2 S_1 V + E_1, \tag{5.14}$$

where $E_1 = S_2 \Delta\hat{W}_1 + \Delta\hat{W}$.

Applying Householder QR to $\hat{W}$ in step 2 incurs error E_2. Only the triangular factor $\hat{R}$ is needed, so some (exactly) orthogonal Q_{tmp} exists such that

$$Q_{tmp}\hat{R} = \hat{W} + E_2 = S_2 S_1 V + E_1 + E_2. \tag{5.15}$$

In step 3, solving the triangular system $Q\hat{R} = V$ also creates errors. These are analyzed in a row-wise fashion, taking the form

$$\hat{Q}_{i,:} = V_{i,:}(\hat{R} + \Delta\hat{R}_i)^{-1} \quad (i = 1, 2, \ldots m), \tag{5.16}$$

where $\hat{Q}_{i,:}$ and $V_{i,:}$ denote the i^{th} rows of $\hat{Q}$ and V, respectively, and $\Delta\hat{R}_i$ is an error term incurred during the solution of the triangular systems. Finally, we can recast the errors incurred in step 3 as $\hat{Q} = (V + \Delta\tilde{V})\hat{R}^{-1}$, which simplifies the analysis of the orthogonality of $\hat{Q}$.

5.5.2 Assumptions and Preparatory Results for our Proofs

Let $V \in \mathbb{R}^{n\times m}$, $n \gg m$, and suppose $S_1 \in \mathbb{R}^{p_1\times m}$ and $S_2 \in \mathbb{R}^{p_2\times p_1}$ are (ε_1, d_1, m) and (ε_2, d_2, m) oblivious ℓ_2-subspace embeddings, respectively, generated independently. Define $d = d_1 + d_2 - d_1 d_2$, $\varepsilon_L = \varepsilon_1 + \varepsilon_2 - \varepsilon_1\varepsilon_2$, $\varepsilon_H = \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2$.

5.5.2.1 Assumptions

For the sake of organization, we define a set of assumptions stating V is sufficiently numerically full rank (i.e., $\kappa(V) \lessapprox \mathbf{u}^{-1}$), $n \gg m$, and that the sketch matrices S_1, S_2 simultaneously satisfy the subspace embedding properties, ensuring equations (5.1)–(5.5), (5.7)–(5.11) hold with probability at least $1 - d$. We also impose an assumption that ϵ_L is sufficiently–but need not be too far–below 1, to obtain a positive lower bound on $\sigma_m(\hat{Q})$ while maintaining as general of a result as possible.

Assumption 5.9. *Suppose $S_1 \in \mathbb{R}^{p_1\times m}$ and $S_2 \in \mathbb{R}^{p_2\times p_1}$ are (ε_1, d_1, m) and (ε_2, d_2, m) oblivious ℓ_2-subspace embeddings respectively, generated independently. Define $d = d_1 + d_2 - d_1 d_2$, $\varepsilon_L = \varepsilon_1 + \varepsilon_2 - \varepsilon_1\varepsilon_2$, $\varepsilon_H = \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2$, where*

$$\varepsilon_L \in \left[0, \frac{616}{625} - \frac{9}{625}\varepsilon_H\right).$$

Further, suppose $V \in \mathbb{R}^{n\times m}$ has full rank and $1 < m \leq p_2 \leq p_1 \leq n$ where $nm\boldsymbol{u} \leq \frac{1}{12}$, and

$$\delta = \frac{12\left(1.1cp_2m^{3/2} + 1.21\sqrt{m}\|S_2\|_2(p_1\sqrt{p_2}\sqrt{1+\varepsilon_1} + 1.1n\|S_1\|_F)\right)}{\sqrt{1-\varepsilon_L}}\boldsymbol{u}\,\kappa(V) \leq 1, \tag{5.17}$$

for some small integer constant c.

The assumption that the integers $m, p_2, p_1,$ and n have the ordering

$$1 < m \leq p_2 \leq p_1 \leq n \tag{5.18}$$

is logical, otherwise the embeddings $S_2 \in \mathbb{R}^{p_2\times p_1}$ and $S_1 \in \mathbb{R}^{p_1\times n}$ project V into a larger space, defeating the purpose of sketching. Further, we assume

$$nm\mathbf{u} \leq \frac{1}{12}\,, \tag{5.19}$$

which is not directly implied by (5.17), but, depending of the values of p_1, p_2, and m, often is a consequence of it.

Define

$$\gamma_k := \frac{k\mathbf{u}}{1-k\mathbf{u}}\,.$$

Provided $k\mathbf{u} < \frac{1}{11}$, it follows that $\gamma_k \leq 1.1k\mathbf{u}$. In particular, since $\kappa(V) \geq 1$, we can deduce from (5.17) that

$$cp_2 m^{3/2}\mathbf{u} \leq \frac{1}{12} \quad \text{and} \quad p_1\sqrt{p_2}\mathbf{u} \leq \frac{1}{12}\,, \tag{5.20}$$

so (5.18)–(5.20) imply

$$\begin{aligned}\gamma_n \leq 1.1n\mathbf{u},\ \gamma_m \leq 1.1m\mathbf{u},\ \gamma_{p_1} \leq 1.1p_1\mathbf{u},\ \gamma_{p_2 m} \leq 1.1p_2 m\mathbf{u},\\ \text{and } \gamma_{cp_2 m} \leq 1.1cp_2 m\mathbf{u}\,,\end{aligned} \tag{5.21}$$

and

$$1+\gamma_n \leq 1.1 \quad \text{and} \quad 1+1.1p_2 m^{3/2}\mathbf{u} \leq 1+1.1cp_2 m^{3/2}\mathbf{u} \leq 1.1\,. \tag{5.22}$$

Finally, we will repeatedly use well-known bounds relating the ℓ_2 and Frobenius norms,

$$\|X\|_2 \leq \|X\|_F \leq \sqrt{m}\|X\|_2, \tag{5.23}$$

$$\|XY\|_F \leq \|X\|_2\|Y\|_F, \quad \text{for any } X \in \mathbb{R}^{n\times m}, Y \in \mathbb{R}^{m\times k}. \tag{5.24}$$

Remark 5.10. Note that instances of $\|S_1\|_F$ and $\|S_2\|_2$ in (5.17) do not dominate the bound on $\kappa(V)$ imposed by (5.17). If $S_1 \in \mathbb{R}^{p_1\times n}$ is an unscaled CountSketch and $S_2 \in \mathbb{R}^{p_2 \times p_1}$ is a scaled Gaussian sketch, there is the determinstic bound

$$\|S_1\|_2 \leq \|S_1\|_F \leq \sqrt{n},$$

and there is the probabilistic bound that there is some constant C such that

$$\|S_2\|_2 \leq C\left(1 + \sqrt{\frac{p_1}{p_2}} + \frac{3}{\sqrt{p_2}}\right),$$

with probability at least $1 - 2e^{-9} \approx 0.9998$ [72]. Thus, in the case of $p_1 = O(m^2)$ and $p_2 = O(m)$, it follows that $\|S_1\|_2 = O(\sqrt{n})$ and $\|S_2\|_2 = O(\sqrt{m})$ with very high probability. Therefore, condition (5.17) ultimately requires

$$\delta \ \leq \ g(n, m, p_1, p_2)\mathbf{u}\, \kappa(V) \leq 1,$$

where g is some low-degree polynomial for reasonable choices of sketches S_1, S_2.

5.5.2.2 Notes on Probabilistic Results

While some bounds constructed throughout the proof are deterministic, several are probabilistic. Here, we address specifically which equations are not deterministic, their prerequisite assumptions, and the probabilities with which they hold.

Throughout the proofs, it is assumed that S_1 embeds the column space of V and S_2 embeds the column space of S_1V simultaneously, which happens with probability at least $1 - d = (1 - d_1)(1 - d_2)$ because S_1 and S_2 are independently generated

(ε_1, d_1, m) and (ε_2, d_2, m) oblivious ℓ_2-subspace embeddings respectively. Therefore,

$$\sqrt{1-\varepsilon_1}\|V\|_2 \leq \|S_1 V\|_2 \leq \sqrt{1+\varepsilon_1}\|V\|_2 \tag{5.25}$$

$$\sqrt{1-\varepsilon_L}\|V\|_2 \leq \|S_2 S_1 V\|_2 \leq \sqrt{1+\varepsilon_H}\|V\|_2 \tag{5.26}$$

$$\sqrt{1-\varepsilon_1}\|V\|_F \leq \|S_1 V\|_F \leq \sqrt{1+\varepsilon_1}\|V\|_F \tag{5.27}$$

$$\sqrt{1-\varepsilon_L}\|V\|_F \leq \|S_2 S_1 V\|_F \leq \sqrt{1+\varepsilon_H}\|V\|_F, \tag{5.28}$$

and

$$\begin{aligned} (1+\varepsilon_H)^{-1/2}\,\sigma_{min}(S_2 S_1 V) \leq \sigma_{min}(V) &\leq \sigma_{max}(V) \\ &\leq (1-\varepsilon_L)^{-1/2}\,\sigma_{max}(S_2 S_1 V), \end{aligned} \tag{5.29}$$

along with the analogous statements for matrices whose column spaces are identical to V, will simultaneously hold with probability at least $1-d$. Specifically, this implies equations (5.33), (5.36), (5.43), (5.44), (5.51), and (5.58) simultaneously hold with probability at least $1-d$, which are used to build all of the results from (5.33)–(5.63) that are prerequisite to prove Theorems 5.12–5.15.

5.5.2.3 Forward Error in matrix-matrix multiplication S_2S_1V

By [36], for $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times k}$, $C = AB$ executed in floating point satisfies

$$\hat{C} = AB + \Delta C, \quad |\Delta C| < \gamma_n |A||B| \, .$$

Thus, in floating point, step 1 of Algorithm 11 becomes:

$$\hat{W}_1 = S_1V + \Delta\hat{W}_1, \quad |\Delta\hat{W}_1| < \gamma_n |S_1||V| \, , \tag{5.30}$$

$$\hat{W} = S_2\hat{W}_1 + \Delta\hat{W}, \quad |\Delta\hat{W}| < \gamma_{p_1}|S_2||\hat{W}_1| = \gamma_{p_1}|S_2||S_1V + \Delta\hat{W}_1| \cdot \tag{5.31}$$

In other words,

$$\hat{W} = S_2S_1V + E_1, \tag{5.32}$$

where the forward error of these matrix-matrix products $E_!$ is defined as:

$$E_1 = \hat{W} - W = \Delta\hat{W} + S_2\Delta\hat{W}_1 \cdot$$

By (5.23), (5.24), (5.27), (5.30), and (5.31),

$$\begin{aligned}
\|E_1\|_2 &\le \|\Delta\hat{W}\|_2 + \|S_2\|_2\|\Delta\hat{W}_1\|_2 \\
&\le \|\Delta\hat{W}\|_F + \|S_2\|_2\|\Delta\hat{W}_1\|_F \\
&\le \gamma_{p_1}\|S_2\|_F\|S_1V + \Delta\hat{W}_1\|_F + \|S_2\|_2\|\Delta\hat{W}_1\|_F \\
&\le \gamma_{p_1}\|S_2\|_F(\|S_1V\|_F + \|\Delta\hat{W}_1\|_F) + \|S_2\|_2\|\Delta\hat{W}_1\|_F \\
&\le \gamma_{p_1}\|S_2\|_F(\sqrt{1+\varepsilon_1}\|V\|_F + \|\Delta\hat{W}_1\|_F) + \|S_2\|_2\|\Delta\hat{W}_1\|_F \\
&\le \sqrt{p_2}\gamma_{p_1}\|S_2\|_2(\sqrt{1+\varepsilon_1}\|V\|_F + \|\Delta\hat{W}_1\|_F) + \|S_2\|_2\|\Delta\hat{W}_1\|_F \\
&= \sqrt{p_2}\gamma_{p_1}\|S_2\|_2\sqrt{1+\varepsilon_1}\|V\|_F + \|S_2\|_2(1+\sqrt{p_2}\gamma_{p_1})\|\Delta\hat{W}_1\|_F \\
&\le \sqrt{p_2}\gamma_{p_1}\|S_2\|_2\sqrt{1+\varepsilon_1}\|V\|_F + \|S_2\|_2(1+\sqrt{p_2}\gamma_{p_1})\gamma_n\|S_1\|_F\|V\|_F \\
&= \|S_2\|_2\left(\sqrt{p_2}\gamma_{p_1}\sqrt{1+\varepsilon_1} + \gamma_n(1+\sqrt{p_2}\gamma_{p_1})\|S_1\|_F\right)\|V\|_F \\
&\le \sqrt{m}\|S_2\|_2\left(\sqrt{p_2}\gamma_{p_1}\sqrt{1+\varepsilon_1} + \gamma_n(1+\sqrt{p_2}\gamma_{p_1})\|S_1\|_F\right)\|V\|_2.
\end{aligned}$$

Notice (5.20) and (5.21) imply $\gamma_n(1+\sqrt{p_2}\gamma_{p_1}) < 1.21n\mathbf{u}$. Hence,

$$\|E_1\|_2 \le \sqrt{m}\mathbf{u}\|S_2\|_2(1.1p_1\sqrt{p_2}\sqrt{1+\varepsilon_1} + 1.21n\|S_1\|_F)\|V\|_2. \tag{5.33}$$

Lemma 5.11. *If S_1 is a ε_1 embedding of the column space of V and S_2 is a ε_2 embedding of the column space of $S_1 V$, then*

$$\frac{12}{\sqrt{1-\varepsilon_L}}\left(1.1cp_2m^{3/2}\boldsymbol{u}\ \kappa(V) + 1.1\|E_1\|_2\,\sigma_m(V)^{-1}\right) \le \delta \le 1\,. \tag{5.34}$$

Proof. Follows directly from applying (5.33) to the definition of δ in (5.17). □

5.5.2.4 Backward Error of Householder QR of $\hat{W}$

By [36, Theorem 19.4], Householder QR of $\hat{W} \in \mathbb{R}^{p_2\times m}$ returns a triangular $\hat{R} \in \mathbb{R}^{m\times m}$ so that some orthogonal $Q_{tmp} \in \mathbb{R}^{p_2\times m}$ satisfies,

$$\hat{W} + E_2 = Q_{tmp}\hat{R}, \quad \|(E_2)_j\|_2 \le \gamma_{cp_2m}\|\hat{w}_j\|_2, \text{ for } j = 1,\ldots,m, \tag{5.35}$$

for some small integer constant c.

By (5.23), (5.32), and the embedding properties of S_2S_1 on V given in (5.28),

$$\begin{aligned}\|\hat{W}\|_F &\le \|S_2S_1V\|_F + \|E_1\|_F \le \sqrt{1+\epsilon_H}\|V\|_F + \|E_1\|_F \\ &\le \sqrt{m}\left(\sqrt{1+\epsilon_H}\|V\|_2 + \|E_1\|_2\right),\end{aligned} \tag{5.36}$$

and therefore

$$\|E_2\|_F \le \gamma_{cp_2m}\|\hat{W}\|_F \le \gamma_{cp_2m}\sqrt{m}\left(\sqrt{1+\epsilon_H}\|V\|_2 + \|E_1\|_2\right)\ , \tag{5.37}$$

with probability at least $1-d$. Finally, by (5.20) and (5.21),

$$\begin{aligned}\|E_2\|_2 \le \|E_2\|_F &\le 1.1cp_2m\mathbf{u}(\sqrt{1+\epsilon_H}\|V\|_F + \|E_1\|_F)\\ &\le 1.1cp_2m^{3/2}\mathbf{u}(\sqrt{1+\epsilon_H}\|V\|_2 + \|E_1\|_2)\\ &= 1.1cp_2m^{3/2}\mathbf{u}\sqrt{1+\epsilon_H}\|V\|_2 + 1.1cp_2m^{3/2}\mathbf{u}\|E_1\|_2\\ &\le 1.1cp_2m^{3/2}\mathbf{u}\sqrt{1+\epsilon_H}\|V\|_2 + 0.1\|E_1\|_2\,\cdot \qquad (5.38)\end{aligned}$$

Additionally, by (5.20) and (5.21),

$$\begin{aligned}\|E_2\|_F &\le 1.1cp_2m\mathbf{u}(\sqrt{1+\epsilon_H}\|V\|_F + \|E_1\|_F)\\ &\le 1.1cp_2m^{3/2}\mathbf{u}\sqrt{1+\epsilon_H}\|V\|_2 + 1.1cp_2m\mathbf{u}\|E_1\|_F\\ &\le 1.1cp_2m^{3/2}\mathbf{u}\sqrt{1+\epsilon_H}\|V\|_2 + 0.1\|E_1\|_F\,\cdot \qquad (5.39)\end{aligned}$$

5.5.2.5 Backward Error of the Forward Substitution

In Step 3 of `randQR`, we solve for Q via the triangular system $Q\hat{R} = V$. By [36, Theorem 8.5], in floating point, $\hat{Q}_{i,:}$ satisfies

$$\hat{Q}_{i,:}(\hat{R} + \Delta R_i) = V_{i,:}, \quad |\Delta R_i| < \gamma_m|\hat{R}| \text{ for } i = 1, \ldots n. \qquad (5.40)$$

While it would be convenient to simply write $\hat{Q}(R + \Delta R) = V$ for some ΔR, each ΔR_i error incurred depends on each right hand side of (5.40), and therefore each row must be accounted for separately. For each $i = 1, \ldots, n$,

$$\|\Delta\hat{R}_i\|_2 \le \|\Delta\hat{R}_i\|_F = \||\Delta\hat{R}_i|\|_F < \gamma_m \||\hat{R}|\|_F = \gamma_m \|\hat{R}\|_F \cdot \tag{5.41}$$

By (5.35), (5.37), and the orthogonality of Q_{tmp}, it follows that

$$\|\hat{R}\|_F = \|Q_{tmp}\hat{R}\|_F = \|\hat{W} + E_2\|_F \le (1 + \gamma_{cp_2m})\|\hat{W}\|_F, \tag{5.42}$$

$$\begin{aligned}\|\hat{R}\|_2 &= \|\hat{W} + E_2\|_2 = \|S_2S_1V + E_1 + E_2\|_2 \\ &\le \sqrt{1+\varepsilon_H}\|V\|_2 + \|E_1\|_2 + \|E_2\|_2.\end{aligned} \tag{5.43}$$

Therefore, by (5.20)–(5.22), (5.36), (5.41), and (5.42),

$$\begin{aligned}\|\Delta\hat{R}_i\|_2 &\le 1.1m^{3/2}\mathbf{u}(1 + 1.1cp_2m\mathbf{u})\left(\sqrt{1+\epsilon_H}\|V\|_2 + \|E_1\|_2\right) \\ &\le 1.21m^{3/2}\mathbf{u}\left(\sqrt{1+\epsilon_H}\|V\|_2 + \|E_1\|_2\right)\end{aligned}$$

5.5.2.6 Bounding the 2-norm of $\hat{R}^{-1}$ and $V\hat{R}^{-1}$

By (5.29), (5.14), and Weyl's inequality [75], with probability at least $1 - d$,

$$\begin{aligned}\sigma_m(\hat{W} + E_2) &\ge \sigma_m(\hat{W}) - \|E_2\|_2 \ge \sigma_m(S_2S_1V) - (\|E_1\|_2 + \|E_2\|_2) \\ &\ge \sqrt{1-\epsilon_L}\,\sigma_m(V) - (\|E_1\|_2 + \|E_2\|_2).\end{aligned} \tag{5.44}$$

By Lemma 5.11, and the fact that the fact that $\|V\|_2 = \kappa(V)\,\sigma_m(V)$,

$$1.1cp_2m^{3/2}\mathbf{u}\sqrt{1+\varepsilon_H}\|V\|_2 + 1.1\|E_1\|_2 \leq \frac{\sqrt{1-\epsilon_L}}{12}\sigma_m(V)\,\delta. \tag{5.45}$$

Combining (5.38) and (5.45) and the assumption that $\delta \leq 1$, results in:

$$\begin{aligned}\|E_1\|_2 + \|E_2\|_2 &\leq 1.1cp_2m^{3/2}\mathbf{u}\sqrt{1+\varepsilon_H}\|V\|_2 + 1.1\|E_1\|_2 \\ &\leq \frac{\sqrt{1-\epsilon_L}}{12}\sigma_m(V)\delta \leq \frac{\sqrt{1-\epsilon_L}}{12}\sigma_m(V),\end{aligned} \tag{5.46}$$

so by (5.35), (5.44), and (5.46),

$$\sigma_m(\hat{R}) = \sigma_m(Q_{tmp}\hat{R}) = \sigma_m(\hat{W} + E_2) \geq \frac{11\sqrt{1-\epsilon_L}}{12}\,\sigma_m(V). \tag{5.47}$$

Therefore, by (5.47)

$$\|\hat{R}^{-1}\|_2 \leq \frac{12}{11\sqrt{1-\epsilon_L}}\left(\sigma_m(V)\right)^{-1}. \tag{5.48}$$

By (5.15), we have that Step 2 of `randQR` satisfies

$$S_2S_1V\hat{R}^{-1} = Q_{tmp} - (E_1 + E_2)\hat{R}^{-1}. \tag{5.49}$$

Thus, by (5.46), (5.48), (5.49), the fact that Q_{tmp} is orthogonal,

$$\|S_2 S_1 V\hat{R}^{-1}\|_2 \leq \|Q_{tmp}\|_2 + (\|E_1\|_2 + \|E_2\|_2)\|\hat{R}^{-1}\|_2 \leq \frac{12}{11}\,, \tag{5.50}$$

with probability at least $1-d$. Observe that V and $V\hat{R}^{-1}$ have the same column space; therefore if S_1, S_2 embed the column space of V, they will also embed the column space of $V\hat{R}^{-1}$. Therefore, by (5.7),

$$\|V\hat{R}^{-1}\|_2 \leq \frac{1}{\sqrt{1-\epsilon_L}}\|S_2 S_1 V\hat{R}^{-1}\|_2 \leq \frac{12}{11\sqrt{1-\epsilon_L}}\,. \tag{5.51}$$

5.5.2.7 Evaluation of the Backward Error $\Delta\tilde{V} = \hat{Q}\hat{R} - V$

Instead of using backward errors ΔR_i for each triangular solve in equation (5.16), we capture the errors of each triangular solve in a matrix $\Delta\tilde{V}$, where

$$\hat{Q} = (V + \Delta\tilde{V})\hat{R}^{-1} \iff \hat{Q}\hat{R} = V + \Delta\tilde{V}. \tag{5.52}$$

We note that we use the notation $\Delta\tilde{V}$ to diferenciate it from ΔV from (5.30).

From (5.16), we have $\hat{Q}_{i,:}(R + \Delta R_i) = V_{i,:}$. Then $\Delta\tilde{V}$ can be defined row-wise,

$$\Delta\tilde{V}_{i,:} = -\hat{Q}_{i,:}\Delta R_i. \tag{5.53}$$

Thus, by (5.40), $|\Delta\tilde{V}_{i,:}| \leq 1.1m\mathbf{u}|\hat{Q}_{i,:}||\hat{R}|$, and so $|\Delta\tilde{V}| \leq 1.1m\mathbf{u}|\hat{Q}||\hat{R}|$, hence,

$$|\Delta\tilde{V}_{:,i}| \leq 1.1m\mathbf{u}|\hat{Q}||\hat{R}_{:,i}|.$$

From this, it follows that for each column $i = 1, \ldots, m$,

$$\begin{aligned}\|\Delta\tilde{V}_{:,i}\|_2 &\leq 1.1m\mathbf{u}\||\hat{Q}|\|_2\||\hat{R}_{:,i}|\|_2 \leq 1.1m\mathbf{u}\||\hat{Q}|\|_F\||\hat{R}_{:,i}|\|_2 \\ &= 1.1m\mathbf{u}\|\hat{Q}\|_F\|\hat{R}_{:,i}\|_2 \leq 1.1m^{3/2}\mathbf{u}\|\hat{Q}\|_2\|\hat{R}_{:,i}\|_2,\end{aligned}$$

and therefore by (5.20), (5.21), (5.36), and (5.42),

$$\begin{aligned}\|\Delta\tilde{V}\|_2 \leq \|\Delta\tilde{V}\|_F &\leq 1.1m^{3/2}\mathbf{u}\|\hat{Q}\|_2\|\hat{R}\|_F \\ &\leq 1.1m^{3/2}\mathbf{u}\|\hat{Q}\|_2(1+\gamma_{cp2m})\|\hat{W}\|_F \\ &\leq 1.1m^{3/2}\mathbf{u}\|\hat{Q}\|_2(1+\gamma_{cp2m})\sqrt{m}\left(\sqrt{1+\varepsilon_H}\|V\|_2 + \|E_1\|_2\right) \\ &\leq 1.21m^2\mathbf{u}\|\hat{Q}\|_2\left(\sqrt{1+\varepsilon_H}\|V\|_2 + \|E_1\|_2\right) \ . \qquad (5.54)\end{aligned}$$

By (5.18) it follows that $1.1\sqrt{m} \le p_2$, and so by (5.22), (5.46), and (5.54),

$$\begin{aligned}
\|\Delta\tilde{V}\|_F &\le 1.21m^2\mathbf{u}\|\hat{Q}\|_2\left(\sqrt{1+\varepsilon_H}\|V\|_2 + \|E_1\|_2\right) \\
&= \|\hat{Q}\|_2\left(1.21m^2\mathbf{u}\sqrt{1+\varepsilon_H}\|V\|_2 + 1.21m^2\mathbf{u}\|E_1\|_2\right) \\
&\le \|\hat{Q}\|_2\left(1.1p_2m^{3/2}\mathbf{u}\sqrt{1+\varepsilon_H}\|V\|_2 + 1.1p_2m^{3/2}\mathbf{u}\|E_1\|_2\right) \\
&\le \|\hat{Q}\|_2\left(1.1p_2m^{3/2}\mathbf{u}\sqrt{1+\varepsilon_H}\|V\|_2 + (1+1.1p_2m^{3/2}\mathbf{u})\|E_1\|_2\right) \\
&\le \|\hat{Q}\|_2\left(1.1p_2m^{3/2}\mathbf{u}\sqrt{1+\varepsilon_H}\|V\|_2 + 1.1\|E_1\|_2\right) \\
&\le \|\hat{Q}\|_2\left(1.1cp_2m^{3/2}\mathbf{u}\sqrt{1+\varepsilon_H}\|V\|_2 + 1.1\|E_1\|_2\right) \\
&\le \|\hat{Q}\|_2\frac{\sqrt{1-\epsilon_L}}{12}\sigma_m(V)\,\delta\,\cdot \qquad (5.55)
\end{aligned}$$

The remaining issue to resolve is that the bound on $\|\Delta\tilde{V}\|_F$ in (5.55) requires knowledge of $\|\hat{Q}\|_2$, which we have not yet found. Combining (5.17), (5.48), and (5.55) gives,

$$\begin{aligned}
\|\hat{Q} - V\hat{R}^{-1}\|_F = \|\Delta\tilde{V}\hat{R}^{-1}\|_F &\le \|\Delta\tilde{V}\|_F\|\hat{R}^{-1}\|_2 \\
&\le \|\hat{Q}\|_2\frac{\sqrt{1-\epsilon_L}}{12}\sigma_m(V)\delta\frac{12}{11\sqrt{1-\epsilon_L}}\left(\sigma_m(V)\right)^{-1} \\
&= \frac{\delta}{11}\|\hat{Q}\|_2 \le \frac{1}{11}\|\hat{Q}\|_2. \qquad (5.56)
\end{aligned}$$

Now, by (5.46), (5.48), and (5.49),

$$\begin{aligned}\|S_2S_1V\hat{R}^{-1} - Q_{tmp}\|_2 \le \|S_2S_1V\hat{R}^{-1} - Q_{tmp}\|_F = \|(E_1 + E_2)\hat{R}^{-1}\|_F \\ \le (\|E_1\|_F + \|E_2\|_F)\,\|\hat{R}^{-1}\|_2 \le \frac{\delta}{11}\,\cdot \qquad (5.57)\end{aligned}$$

Applying Weyl's inequality to (5.57) and the fact that Q_{tmp} is orthogonal yields,

$$1 - \frac{\delta}{11} \le \sigma_m(S_2S_1V\hat{R}^{-1}) \le \sigma_1(S_2S_1V\hat{R}^{-1}) \le 1 + \frac{\delta}{11}\,\cdot$$

Since V and $V\hat{R}^{-1}$ have identical column spaces and S_1,S_2 embed the column space of V, the embedding properties in (5.29) also apply to $V\hat{R}^{-1}$, and so

$$\frac{1-\frac{\delta}{11}}{\sqrt{1+\epsilon_H}} \le \sigma_m(V\hat{R}^{-1}) \le \sigma_1(V\hat{R}^{-1}) \le \frac{1+\frac{\delta}{11}}{\sqrt{1-\epsilon_L}} \le \frac{12}{11\sqrt{1-\epsilon_L}}\,\cdot \qquad (5.58)$$

Then, we can use Weyl's inequality again on $\hat{Q} - V\hat{R}^{-1}$. In particular,

$$\sigma_m(V\hat{R}^{-1}) - \|\hat{Q} - V\hat{R}^{-1}\|_2 \le \sigma_m(\hat{Q}) \le \sigma_1(\hat{Q}) \le \sigma_1(V\hat{R}^{-1}) + \|\hat{Q} - V\hat{R}^{-1}\|_2\,\cdot \qquad (5.59)$$

Then, by (5.56), (5.58), and (5.59),

$$\|\hat{Q}\|_2 = \sigma_1(\hat{Q}) \le \sigma_1(V\hat{R}^{-1}) + \|\hat{Q} - V\hat{R}^{-1}\|_2 \le \frac{12}{11\sqrt{1-\epsilon_L}} + \frac{1}{11}\|\hat{Q}\|_2, \qquad (5.60)$$

and so,

$$\|\hat{Q}\|_2 \leq \frac{6}{5\sqrt{1-\epsilon_L}} \cdot \tag{5.61}$$

Then, we obtain from (5.56),

$$\|\hat{Q} - V\hat{R}^{-1}\|_2 = \|\Delta\tilde{V}\hat{R}^{-1}\|_2 \leq \|\Delta\tilde{V}\hat{R}^{-1}\|_F \leq \frac{\delta}{11}\|\hat{Q}\|_2 \leq \frac{6\delta}{55\sqrt{1-\epsilon_L}}. \tag{5.62}$$

5.5.2.8 Bounding $\|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2$

If no additional assumptions on the embedding of S_1, S_2 are made, clearly it follows that

$$\|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2 \leq \|S_2\|_2\|S_1\|_2\|\Delta\tilde{V}\hat{R}^{-1}\|_2 \leq \frac{6\|S_2\|_2\|S_1\|_2}{55\sqrt{1-\epsilon_L}}\,\delta. \tag{5.63}$$

Alternatively, if we assume S_1, S_2 embed $\Delta\tilde{V}\hat{R}^{-1}$, by (5.7),

$$\|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2 \leq \sqrt{1+\epsilon_H}\|\Delta\tilde{V}\hat{R}^{-1}\|_2 \leq \frac{6\sqrt{1+\epsilon_H}}{55\sqrt{1-\epsilon_L}}\,\delta. \tag{5.64}$$

5.5.3 Key Theoretical Results

Remark 5.6 indicates that in exact arithmetic, `randQR` yields a matrix Q that is orthogonal with respect to $\langle S_2S_1\cdot, S_2S_1\cdot\rangle$. We show next that provided V has full numerical rank, then in floating point arithmetic, the orthogonality error of the matrix $\hat{Q}$ generated by `randQR` measured in $\langle S_2S_1\cdot, S_2S_1\cdot\rangle$ is $O(\mathbf{u})\kappa(V)$, and the factorization error is $O(\mathbf{u})\|V\|_2$ with high probability.

Theorem 5.12 (`randQR` Errors)**.** *Suppose Assumptions 5.9 are satisfied. Then the* $\hat{Q}, \hat{R}$ *factors obtained with Algorithm 11 (*`randQR`*) satisfy*

$$\|V - \hat{Q}\hat{R}\|_2 \leq \frac{\delta}{10}\sigma_m(V), \tag{5.65}$$

and

$$\begin{aligned}&\|(S_2S_1\hat{Q})^T(S_2S_1\hat{Q}) - I\|_2 \\ &\qquad \leq 2\frac{\delta}{11} + \left(\frac{\delta}{11}\right)^2 + \frac{24}{11}\frac{6\|S_2\|_2\|S_1\|_2}{55\sqrt{1-\epsilon_L}}\,\delta + \left(\frac{6\|S_2\|_2\|S_1\|_2}{55\sqrt{1-\epsilon_L}}\,\delta\right)^2.\end{aligned} \tag{5.66}$$

with probability at least $1 - d$*, where* δ *is defined as in* (5.17)*. Furthermore,*

$$\|(S_2S_1\hat{Q})^T(S_2S_1\hat{Q}) - I\|_2 \leq 3\delta \tag{5.67}$$

with probability at least $(1-d)^2$*.*

Proof. Equation (5.65) follows by combining (5.55) and (5.61), since $\Delta\tilde{V} = \hat{Q}\hat{R} - V$, and this holds with probability at least $1-d$ because (5.55) and (5.61) hold with this probability, as discussed in Section 5.5.2.2.

Observe that by (5.15), we have $S_2S_1V = Q_{tmp}\hat{R} - (E_1 + E_2)$, and so

$$(S_2S_1V)^T(S_2S_1V) =$$
$$\hat{R}^T\hat{R} - (E_1 + E_2)^TQ_{tmp}\hat{R} - \hat{R}^TQ_{tmp}^T(E_1 + E_2) + (E_1 + E_2)^T(E_1 + E_2).$$

Using (5.52) to expand $S_2S_1\hat{Q} = (S_2S_1V + S_2S_1\Delta\tilde{V})\hat{R}^{-1}$, we get

$$\begin{aligned}(S_2S_1\hat{Q})^T(S_2S_1\hat{Q}) &= I - \hat{R}^{-T}(E_1 + E_2)^TQ_{tmp} - Q_{tmp}(E_1 + E_2)\hat{R}^{-1}\\ &+ \hat{R}^{-T}(E_1 + E_2)^T(E_1 + E_2)\hat{R}^{-1} + (S_2S_1\Delta\tilde{V}\hat{R}^{-1})^T(S_2S_1V\hat{R}^{-1})\\ &+ (S_2S_1V\hat{R}^{-1})^TS_2S_1\Delta\tilde{V}\hat{R}^{-1} + (S_2S_1\Delta\tilde{V}\hat{R}^{-1})^T(S_2S_1\Delta\tilde{V}\hat{R}^{-1}).\end{aligned}$$

Therefore, by (5.46), (5.48), and (5.50),

$$\begin{aligned}&\|(S_2S_1\hat{Q})^T(S_2S_1\hat{Q}) - I\|_2\\ &\le 2(\|E_1\|_2 + \|E_2\|_2)\|\hat{R}^{-1}\|_2 + (\|E_1\|_2 + \|E_2\|_2)^2\|\hat{R}^{-1}\|_2^2 \quad (5.68)\\ &+ 2\|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2\|S_2S_1V\hat{R}^{-1}\|_2 + \|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2^2\\ &\le 2\frac{\delta}{11} + \left(\frac{\delta}{11}\right)^2 + \frac{24}{11}\|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2 + \|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2^2. \quad (5.69)\end{aligned}$$

Observe that (5.46), (5.48), and (5.50), simultaneously hold with probability at least $1 - d$, because they rely on V and S_1V being simultaneously embedded by S_1 and S_2 respectively, which with this probability occurs, as discussed in Section

5.5.2.2. Thus, (5.69) holds with probability at least $1 - d$. Observe that (5.63) requires no further assumptions on the embedding properties of S_1, S_2 and so applying (5.63) to (5.69) gives

$$\begin{aligned}
&\|(S_2S_1\hat{Q})^T(S_2S_1\hat{Q}) - I\|_2 \\
&\qquad \le 2\frac{\delta}{11} + \left(\frac{\delta}{11}\right)^2 + \frac{24}{11}\|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2 + \|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2^2 \\
&\qquad \le 2\frac{\delta}{11} + \left(\frac{\delta}{11}\right)^2 + \frac{24}{11}\frac{6\|S_2\|_2\|S_1\|_2}{55\sqrt{1-\epsilon_L}}\,\delta + \left(\frac{6\|S_2\|_2\|S_1\|_2}{55\sqrt{1-\epsilon_L}}\,\delta\right)^2,
\end{aligned}$$

with probability at least $1 - d$, producing result (5.66).

On the other hand, observe that (5.64) requires not only the assumption that S_1, S_2 simultaneously embed V and S_1V respectively, but also that the sketch matrices embed $\Delta\tilde{V}\hat{R}^{-1}$ and $S_1\Delta\tilde{V}\hat{R}^{-1}$ respectively. Thus, (5.64) and (5.69) simultaneously hold with probability at least $(1-d)^2$, and the result of applying both of these results together yields,

$$\begin{aligned}
&\|(S_2S_1\hat{Q})^T(S_2S_1\hat{Q}) - I\|_2 \\
&\qquad \le 2\frac{\delta}{11} + \left(\frac{\delta}{11}\right)^2 + \frac{24}{11}\|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2 + \|S_2S_1\Delta\tilde{V}\hat{R}^{-1}\|_2^2 \\
&\qquad \le 2\frac{\delta}{11} + \left(\frac{\delta}{11}\right)^2 + \frac{24}{11}\frac{6\sqrt{1+\varepsilon_H}}{55\sqrt{1-\epsilon_L}}\,\delta + \left(\frac{6\sqrt{1+\varepsilon_H}}{55\sqrt{1-\epsilon_L}}\,\delta\right)^2,
\end{aligned} \tag{5.70}$$

with probability at least $(1-d)^2$. A useful consequence of Assumptions 5.9 is that

$$1-\varepsilon_L > \frac{9}{625}(1+\varepsilon_H),$$

and thus

$$\frac{1+\varepsilon_H}{1-\varepsilon_L} < \frac{625}{9} \Rightarrow \sqrt{\frac{1+\varepsilon_H}{1-\varepsilon_L}} < \frac{25}{3}\,. \tag{5.71}$$

Applying (5.71) to (5.70) (which holds with probability at least $(1-d)^2$) along with the fact that $\delta \leq 1$ from (5.17) implies $\delta^2 \leq \delta$, and so,

$$\begin{aligned}
&\|(S_2S_1\hat{Q})^T(S_2S_1\hat{Q}) - I\|_2 \\
&\qquad \leq 2\frac{\delta}{11} + \left(\frac{\delta}{11}\right)^2 + \frac{24}{11}\frac{6\sqrt{1+\varepsilon_H}}{55\sqrt{1-\epsilon_L}}\,\delta + \left(\frac{6\sqrt{1+\varepsilon_H}}{55\sqrt{1-\epsilon_L}}\,\delta\right)^2 \\
&\qquad \leq \frac{2}{11}\delta + \left(\frac{1}{11}\right)^2\delta + \frac{24}{11}\frac{6\cdot 25}{55\cdot 3}\,\delta + \left(\frac{6\cdot 25}{55\cdot 3}\right)^2\delta \\
&\qquad = \frac{2\cdot 11\cdot 55^2\cdot 3^2 + 55^2\cdot 3^2 + 24\cdot 6\cdot 25\cdot 11\cdot 55\cdot 3 + 6^2\cdot 25^2\cdot 11^2}{11^2\cdot 55^2\cdot 3^2}\delta \\
&\qquad = 3\delta\,,
\end{aligned}$$

with probability at least $(1-d)^2$, and thus result (5.67) follows.

□

Similar to the analysis of the condition number of Q generated by `randQR` in exact arithmetic in Section 5.4, we show next that provided that V has full numerical rank, then $\hat{Q}$ generated by `randQR` in floating point arithmetic also has $\kappa(\hat{Q}) = O(1)$.

Theorem 5.13 (Conditioning of `randQR`)**.** *Suppose Assumptions 5.9 are satisfied. Then with probability at least* $1 - d$*, the* $\hat{Q}$ *matrix obtained with Algorithm 11 (*`randQR`*) has condition number* $\kappa(\hat{Q}) = O(1)$*. In fact,*

$$\kappa(\hat{Q}) \le \frac{33}{25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3}. \tag{5.72}$$

Proof. As a direct consequence of (5.58), (5.59), (5.62), and the fact that $\delta \le 1$,

$$\begin{aligned}\sigma_m(\hat{Q}) &\ge \sigma_m(V\hat{R}^{-1}) - \|\hat{Q} - V\hat{R}^{-1}\|_2 \ge \frac{1 - \frac{\delta}{11}}{\sqrt{1+\epsilon_H}} - \frac{6\delta}{55\sqrt{1-\epsilon_L}} \\ &\ge \frac{10}{11\sqrt{1+\epsilon_H}} - \frac{6}{55\sqrt{1-\epsilon_L}} .\end{aligned}$$

Additionally, we found in (5.61) that

$$\sigma_1(\hat{Q}) = \|\hat{Q}\|_2 \le \frac{6}{5\sqrt{1-\epsilon_L}} .$$

Thus,

$$\kappa(\hat{Q}) = \frac{\sigma_1(\hat{Q})}{\sigma_m(\hat{Q})} \le \frac{33}{25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3} .$$

Since the intermediate results (5.58), (5.59), (5.61), and (5.62) simultaneously hold with probability at least $1-d$, as discussed in Section 5.5.2.2, the final result (5.13) holds with this probability as well. $\square$

In the following result we show that `rand_cholQR`(V) (Algorithm 12) produces a factor $\hat{Q}$ that is orthogonal in the Euclidean inner product up to a factor of $O(\mathbf{u})$ and has a factorization error of $O(\mathbf{u})\|V\|_2$ for any numerically full rank V.

Theorem 5.14 (`rand_cholQR` Errors)**.** *Suppose Assumptions 5.9 are satisfied. Then with probability at least* $1-d$*, the* $\hat{Q}, \hat{R}$ *factors obtained with Algorithm 12 (*`rand_cholQR`*) has* $O(\boldsymbol{u})$ *orthogonality error and* $O(\boldsymbol{u})\|V\|_2$ *factorization error. In fact,*

$$\|\hat{Q}^T\hat{Q} - I\|_2 \leq \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u}, \tag{5.73}$$

$$\begin{aligned}\|V - \hat{Q}\hat{R}\|_2 \leq &\left(\frac{56}{25\frac{1-\varepsilon_L}{\sqrt{1+\epsilon_H}} - 3\sqrt{1-\varepsilon_L}} + \frac{1.5}{\sqrt{1-\varepsilon_L}}\sqrt{1 + \frac{5445(nm+m(m+1))\boldsymbol{u}}{\left(25\sqrt{\frac{1-\varepsilon_L}{1+\varepsilon_H}} - 3\right)^2}}\right) \\ &\left(\sqrt{1+\varepsilon_H}\|V\|_2 + \frac{\sqrt{1-\varepsilon_L}}{12}\sigma_m(V)\delta\right) m^2\boldsymbol{u} + \frac{\delta}{10}\sigma_m(V),\end{aligned} \tag{5.74}$$

where δ *is bounded as in* (5.17)*.*

Proof. In Algorithm 12, we obtain $\hat{Q}_0, \hat{R}_0$ from `randQR` (so that the results in Section 5.5.2 apply to $\hat{Q}_0, \hat{R}_0$), and then obtain $\hat{Q}, \hat{R}$ where $\hat{R} = \mathrm{fl}(\hat{R}_1\hat{R}_0)$ and $\hat{Q}, \hat{R}_1$ are the outputs of Cholesky QR applied to $\hat{Q}_0$. As a direct consequence of

Theorem 5.13, $\hat{Q}_0$ arising from Step 1 of Algorithm 12 satisfies

$$\kappa(\hat{Q}_0) \le \frac{33}{25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3}.$$

By [77, Lemma 3.1], it follows that Step 2 of Algorithm 12 gives $\hat{Q}$ satisfying

$$\begin{aligned} \|\hat{Q}^T\hat{Q} - I\|_2 &\le \frac{5}{64} 64\kappa(\hat{Q}_0)^2 \left(nm + m(m+1)\right) \mathbf{u} \\ &\le \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3\right)^2} \left(nm + m(m+1)\right) \mathbf{u}, \end{aligned}$$

and so (5.73) follows.

Now, notice that by (5.19)–(5.21),

$$\sqrt{\frac{1+\gamma_n m}{1-\gamma_{m+1} m}} \le \sqrt{\frac{1+1.1nm\mathbf{u}}{1-1.1(m+1)m\mathbf{u}}} \le \sqrt{\frac{1.1}{0.9}} \le 1.11. \tag{5.75}$$

Observe $\hat{R} = \hat{R}_1\hat{R}_0 + \Delta\hat{R}$ where $\hat{R}_1$ is the Cholesky factor of $\hat{Q}_0^T\hat{Q}_0$, where $\hat{Q}_0$ results from `randQR`, and $|\Delta R| < \gamma_m|\hat{R}_1||\hat{R}_0|$ [36, Eq. (3.13)]. Then, it follows by [77, Eq. (3.16)], (5.21), (5.43). (5.46), (5.61), and (5.75), that

$$\begin{aligned}
\|\Delta\hat{R}\|_2 &\leq \|\Delta\hat{R}\|_F \leq \gamma_m \|\hat{R}_1\|_F \|\hat{R}_0\|_F \leq m\gamma_m \|\hat{R}_1\|_2 \|\hat{R}_0\|_2 \\
&\leq m\gamma_m \sqrt{\frac{1+\gamma_n m}{1-\gamma_{m+1} m}} \|\hat{Q}_0\|_2 \|\hat{R}_0\|_2 \\
&\leq 1.23 m^2 \mathbf{u} \frac{6}{5\sqrt{1-\varepsilon_L}} (\sqrt{1+\varepsilon_H}\|V\|_2 + \frac{\sqrt{1-\varepsilon_L}}{12}\sigma_m(V)\delta) \\
&\leq m^2 \mathbf{u} \frac{1.5}{\sqrt{1-\varepsilon_L}} (\sqrt{1+\varepsilon_H}\|V\|_2 + \frac{\sqrt{1-\varepsilon_L}}{12}\sigma_m(V)\delta) \cdot
\end{aligned} \tag{5.76}$$

Next, observe that by (5.52) we have that $\hat{Q}_0\hat{R}_0 = V + \Delta\tilde{V}$ from `randQR`. Using this and [77, Eq. (3.24)] to bound $\|\hat{Q}_0 - \hat{Q}\hat{R}_1\|_2$,

$$\begin{aligned}
-\|\Delta\tilde{V}\|_2 + \|V - \hat{Q}\hat{R}\|_2 \leq \|V + \Delta\tilde{V} - \hat{Q}\hat{R}\|_2 &= \|\hat{Q}_0\hat{R}_0 - \hat{Q}\hat{R}_1\hat{R}_0 - \hat{Q}\Delta\hat{R}\|_2 \\
&\leq \|\hat{R}_0\|\|\hat{Q}_0 - \hat{Q}\hat{R}_1\|_2 + \|\hat{Q}\|_2\|\Delta\hat{R}\|_2 \\
&\leq 1.4\|\hat{R}_0\|_2 \kappa(\hat{Q}_0)\|\hat{Q}_0\|_2 m^2 \mathbf{u} + \|\hat{Q}\|_2\|\Delta\hat{R}\|_2 \cdot
\end{aligned} \tag{5.77}$$

Observe that (5.73) implies

$$\|\hat{Q}\|_2 \leq \sqrt{1 + \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3\right)^2} \left(nm + m(m+1)\right)\mathbf{u}} \,.$$

Additionally, by (5.65) in Theorem 5.12,

$$\|\Delta\tilde{V}\|_2 = \|V - \hat{Q}_0\hat{R}_0\|_2 \leq \frac{\delta}{10}\sigma_m(V). \tag{5.78}$$

Now, starting from (5.43), we can use (5.46) to obtain

$$\begin{aligned}\|\hat{R}_0\|_2 &\le \sqrt{1+\varepsilon_H}\|V\|_2 + \|E_1\|_2 + \|E_2\|_2 \\ &\le \sqrt{1+\varepsilon_H}\|V\|_2 + \frac{\sqrt{1-\varepsilon_L}}{12}\sigma_m(V)\delta. \qquad (5.79)\end{aligned}$$

Using (5.72) and (5.61) to bound $\kappa(\hat{Q}_0)$ and $\|\hat{Q}_0\|_2$, (5.76) to bound $\|\Delta\hat{R}\|_2$, (5.79) to bound $\|\hat{R}_0\|_2$, adding $\|\Delta\tilde{V}\|_2$ to both sides of (5.77) and then bounding $\|\Delta\tilde{V}\|_2$ using (5.78), we finally obtain

$$\|V-\hat{Q}\hat{R}\|_2 \le \left(\frac{56}{25\frac{1-\varepsilon_L}{\sqrt{1+\epsilon_H}} - 3\sqrt{1-\varepsilon_L}} + \frac{1.5}{\sqrt{1-\varepsilon_L}}\sqrt{1+\frac{5445(nm+m(m+1))\mathbf{u}}{\left(25\sqrt{\frac{1-\varepsilon_L}{1+\varepsilon_H}}-3\right)^2}}\right) \left(\sqrt{1+\varepsilon_H}\|V\|_2 + \frac{\sqrt{1-\varepsilon_L}}{12}\sigma_m(V)\delta\right) m^2\mathbf{u} + \frac{\delta}{10}\sigma_m(V),$$

which does indeed satisfy $\|V-\hat{Q}\hat{R}\|_2 = O(\mathbf{u})\|V\|_2$, since $\sigma_m(V)\delta = O(\mathbf{u})\|V\|_2$.

Finally, observe that the probabilistic results used in this proof, namely (5.33)–(5.63) and Theorems 5.12–5.13, simultaneously hold with probability at least $1-d$ (see Section 5.5.2.2 for details), and hence (5.73) and (5.74) hold with this probability as well. $\square$

Theorem 5.13 guarantees `randQR`(V) produces a well-conditioned $\hat{Q}$. We show next that `rand_cholQR`(V) produces a factor $\hat{Q}$ with $\kappa(\hat{Q}) \approx 1$ (up to unit roundoff) for any numerically full rank V.

Theorem 5.15 (Conditioning of `rand_cholQR`)**.** *Suppose Assumptions 5.9 are satisfied. Then with probability at least* $1-d$*, the matrix* $\hat{Q}$ *obtained with Algorithm 11 satisfies* $\kappa(\hat{Q}) \approx 1$*. In fact,*

$$\kappa(\hat{Q}) < \sqrt{\frac{1 + \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}}-3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u}}{1 - \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}}-3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u}}}. \tag{5.80}$$

Furthermore, if $\frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}}-3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u} < \frac{1}{2}$*, then*

$$\kappa(\hat{Q}) < 1 + \frac{10890}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u}. \tag{5.81}$$

Proof. It follows from (5.73) that the i^{th} eigenvalue of $\hat{Q}^T Q$ satisfies

$$\lambda_i(\hat{Q}^T\hat{Q}) \geq 1 - \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3\right)^2}\left(nm + m(m+1)\right)\mathbf{u},$$

$$\lambda_i(\hat{Q}^T\hat{Q}) \leq 1 + \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3\right)^2}\left(nm + m(m+1)\right)\mathbf{u}.$$

Thus, the i^{th} singular value of $\hat{Q}$ satisfies

$$\sigma_i(\hat{Q}) \geq \sqrt{1 - \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3\right)^2} \left(nm + m(m+1)\right) \mathbf{u}},$$

$$\sigma_i(\hat{Q}) \leq \sqrt{1 + \frac{5445}{\left(25\sqrt{\frac{1-\epsilon_L}{1+\epsilon_H}} - 3\right)^2} \left(nm + m(m+1)\right) \mathbf{u}},$$

which gives (5.80). Further, for any $x < \frac{1}{2}$, $\sqrt{\frac{1+x}{1-x}} < 1 + 2x$, which gives (5.81). Since (5.73) holds with probability at least $1 - d$, (5.80) and (5.81) hold with this probability as well. □

Theorems 5.12–5.15 correspond to multisketchings, that is, to the application of one sketch matrix after another. In the rest of the section, we recast our error bounds for a single sketch matrix in Corollaries 5.17–5.20. The results apply for a single (ε, d, m) oblivious ℓ_2-subspace embedding for any $\varepsilon \in [0, \frac{616}{634})$, covering nearly the entire range of possible $\varepsilon \in [0, 1)$ for such embeddings.

We prove all the Corollaries simultaneously, as they are direct consequences of Theorems 5.12–5.15 by exploiting the fact that a single sketch can be recast as a product of two sketches, one of which is the identity, which is by definition a $(0, 0, m)$ oblivious ℓ_2-subspace embedding.

Assumption 5.16. *Suppose $\varepsilon \in [0, \frac{616}{634})$ and $S \in \mathbb{R}^{p \times m}$ is a (ε, d, m) oblivious ℓ_2-subspace embedding. Further, suppose $V \in \mathbb{R}^{n \times m}$ has full rank and*

$1 < m \leq s \leq n$ where $nm\boldsymbol{u} \leq \frac{1}{12}$ and

$$\delta = \frac{12\left(1.1cpm^{3/2} + 1.21\sqrt{m}(p^{3/2}\sqrt{1+\varepsilon} + 1.1n\|S\|_F)\right)}{\sqrt{1-\varepsilon}}\boldsymbol{u}\,\kappa(V) \leq 1. \qquad (5.82)$$

Corollary 5.17 (`randQR` Errors)**.** *Suppose Assumptions 5.16 are satisfied. Then the $\hat{Q}$, $\hat{R}$ factors obtained with Algorithm 11 (`randQR`) satisfy*

$$\|V - \hat{Q}\hat{R}\|_2 \leq \frac{\delta}{10}\sigma_m(V), \qquad (5.83)$$

and

$$\|(S\hat{Q})^T(S\hat{Q}) - I\|_2 \leq \frac{2\delta}{11} + \left(\frac{\delta}{11}\right)^2 + \frac{24}{11}\frac{6\|S\|_2}{55\sqrt{1-\epsilon}}\delta + \left(\frac{6\|S\|_2}{55\sqrt{1-\epsilon}}\delta\right)^2 \qquad (5.84)$$

with probability at least $1-d$, where δ is defined as in (5.82)*. Furthermore,*

$$\|(S\hat{Q})^T(S\hat{Q}) - I\|_2 \leq 3\delta \qquad (5.85)$$

with probability at least $(1-d)^2$.

Corollary 5.18 (Conditioning of `randQR`)**.** *Suppose Assumptions 5.16 are satisfied. Then with probability at least $1-d$, the $\hat{Q}$ matrix obtained with Algorithm 11* *(`randQR`) has condition number $\kappa(\hat{Q}) = O(1)$. In fact,*

$$\kappa(\hat{Q}) \leq \frac{33}{25\sqrt{\frac{1-\epsilon}{1+\epsilon}} - 3}. \tag{5.86}$$

Therefore, if $\varepsilon \leq 0.9$,

$$\kappa(\hat{Q}) \leq 12.07.$$

Corollary 5.19 (`rand_cholQR` Errors)**.** *Suppose Assumptions 5.16 are satisfied. Then with probability at least $1-d$, the $\hat{Q}, \hat{R}$ factors obtained with Algorithm 12 (`rand_cholQR`) has $O(\boldsymbol{u})$ orthogonality error and $O(\boldsymbol{u})\|V\|_2$ factorization error. In fact,*

$$\|\hat{Q}^T\hat{Q} - I\|_2 \leq \frac{5445}{\left(25\sqrt{\frac{1-\epsilon}{1+\epsilon}} - 3\right)^2} \left(nm + m(m+1)\right)\boldsymbol{u}, \tag{5.87}$$

$$\|V - \hat{Q}\hat{R}\|_2 \leq \left(\frac{56}{25\frac{1-\varepsilon}{\sqrt{1+\epsilon}} - 3\sqrt{1-\varepsilon}} + \frac{1.5}{\sqrt{1-\varepsilon}}\sqrt{1 + \frac{5445(nm + m(m+1))\boldsymbol{u}}{\left(25\sqrt{\frac{1-\varepsilon}{1+\varepsilon}} - 3\right)^2}}\right)$$
$$\left(\sqrt{1+\varepsilon}\|V\|_2 + \frac{\sqrt{1-\varepsilon}}{12}\sigma_m(V)\delta\right) m^2\boldsymbol{u} + \frac{\delta}{10}\sigma_m(V), \tag{5.88}$$

where δ is bounded as in (5.82).

Corollary 5.20 (Conditioning of `rand_cholQR`)**.** *Suppose Assumptions 5.16 are satisfied. Then with probability at least $1-d$, the matrix $\hat{Q}$ obtained with Algorithm*

11 satisfies $\kappa(\hat{Q}) \approx 1$. In fact,

$$\kappa(\hat{Q}) < \sqrt{\frac{1 + \frac{5445}{\left(25\sqrt{\frac{1-\epsilon}{1+\epsilon}}-3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u}}{1 - \frac{5445}{\left(25\sqrt{\frac{1-\epsilon}{1+\epsilon}}-3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u}}}. \tag{5.89}$$

Furthermore, if $\frac{5445}{\left(25\sqrt{\frac{1-\epsilon}{1+\epsilon}}-3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u} < \frac{1}{2}$, then

$$\kappa(\hat{Q}) < 1 + \frac{10890}{\left(25\sqrt{\frac{1-\epsilon}{1+\epsilon}}-3\right)^2}\left(nm + m(m+1)\right)\boldsymbol{u}. \tag{5.90}$$

Proof. We prove Corollaries 5.17–5.20 simultaneously by considering one subspace embedding $S = S_1$ is equivalent to two subspace embeddings S_2S_1 simply by interpreting $S_2 = I_{p,p}$ as the $p \times p$ identity, which is by definition a $(0, 0, m)$ oblivious ℓ_2-subspace embedding, therefore giving $\varepsilon_1 = \varepsilon_H = \varepsilon_L = \varepsilon$, $d = d_1$, $p = p_2 = p_1$, $\varepsilon_2 = 0$ and $d_2 = 0$.

We show next that if $\varepsilon = \varepsilon_L = \varepsilon_H \in [0, \frac{616}{634})$, then $\varepsilon_L \in [0, \frac{616}{625} - \frac{9}{625}\varepsilon_H)$. Indeed, $\varepsilon_L \in [0, \frac{616}{625} - \frac{9}{625}\varepsilon_H)$ is equivalent in this case to $0 \le \varepsilon < \frac{616}{625} - \frac{9}{625}\varepsilon$, or $0 \le \frac{634}{625}\varepsilon < \frac{616}{625}$, or what is the same, $\varepsilon \in [0, \frac{616}{634})$. This means that when $\varepsilon_H = \varepsilon_L = \varepsilon \in [0, \frac{616}{634})$, Assumptions 5.9 imply Assumptions 5.16. Thus, Assumptions 5.9 are satisfied and Corollaries 5.17–5.20 are direct consequences of Theorems 5.12–5.15. $\square$

Remark 5.21. Observe that (5.82) in Assumptions 5.16 is identical to (5.17) in Assumptions 5.9 using $S_1 = S$ and $S_2 = I_{p,p}$, where $p_1 = p_2 = p$. However, the analysis in Section 5.5.2.3 of $\|E_1\|_2$ takes into account roundoff errors for two matrix multiplications for two sketches to compute $\hat{W}$, while in the case of Corollaries 5.17–5.20, only one sketch and therefore one matrix multiplication to compute $\hat{W}$ is necessary. Therefore, we are over-estimating the error $\|E_1\|_2$ in the single sketch case, and if the analysis in Sections 5.5.2 were carefully performed again, we could tighten the bound on δ in (5.82), thereby loosening the requirements on $\kappa(V)$ in Assumptions 5.16. However, asymptotically, the requirement on $\kappa(V)$ would ultimately still be that $\delta \leq g(n, m, p_1, p_2)\mathbf{u}\,\kappa(V) \leq 1$ for some low-degree polynomial g.

5.6 Numerical Experiments

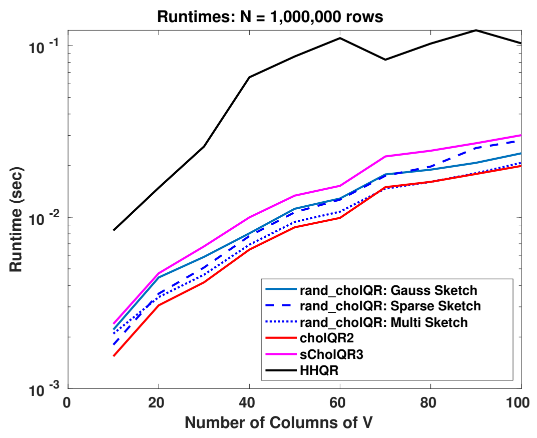
We conducted numerical experiments with two goals in mind. First, we compare the performance of `rand_cholQR` with the performance of `cholQR2`, `sCholQR3`, and Householder QR on a latest GPU leveraging vendor-optimized libraries. Second, we empirically validate the bounds given in Section 5.5.3, and more generally, compare the stability of `rand_cholQR` to the stability of `cholQR2`, `sCholQR3`, and Householder QR.

5.6.1 Implementation Details

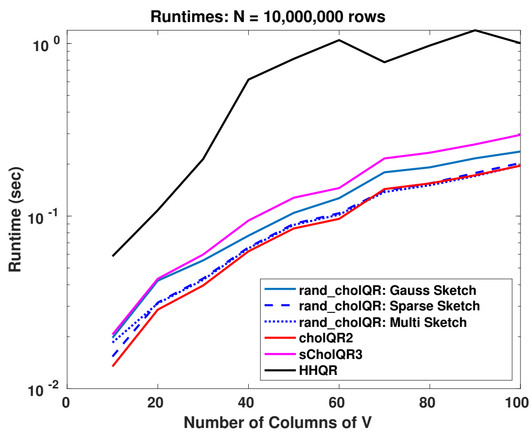
We implemented `rand_cholQR`, `cholQR2`, `sCholQR3` (Algorithm 10 given in Section 4.4), and Householder QR in C++. To be portable to a GPU, we used the Kokkos Performance Portability Library [24] and Kokkos Kernels [55]. For our experiments on an NVIDIA GPU, we configured and built our code such that Kokkos Kernels calls NVIDIA's cuBLAS and cuSPARSE linear algebra libraries for optimized dense and sparse basic linear algebra routines [46, 49]. To perform LAPACK routines that are not currently available natively within Kokkos Kernels (i.e., `dgeqrf` and `dorgqr` for computing the Householder QR factorization, and `dpotrf` for the Cholesky factorization), we directly called NVIDIA's cuSOLVER linear algebra library [2, 47, 48]. Test results were obtained using Kokkos 3.7.01, Cuda 11.7.99, and GCC 7.2.0 on an AMD EPYC 7742 64-Core 2.25GHz CPU with a NVIDIA A100-SXM4 40GB GPU. All computations were done in double precision, so $\mathbf{u} = 2^{-52} \approx 10^{-16}$.

We tested a variety of sketching strategies. The simplest was the case of a Gaussian sketch $S = \frac{1}{\sqrt{p}} G \in \mathbb{R}^{p \times n}$, which were generated within a parallel for loop. The sketch size chosen for a Gaussian to embed $V \in \mathbb{R}^{n \times m}$ was $p = \lceil 74.3 \log(m) \rceil$, which can be shown to produce a $(0.49, 1/m, m)$ oblivious ℓ_2-subspace embedding [1, Lemma 4.1]. To test using CountSketch, we explicitly constructed a sparse matrix and applied the sketch using a sparse-matrix vector product. The sketch size used to embed $V \in \mathbb{R}^{n \times m}$ with a $S \in \mathbb{R}^{p \times n}$ CountSketch matrix was $p = \lceil 8.24(m^2 + m) \rceil$, which can be shown to be a $(0.9, 0.15, m)$ oblivious ℓ_2-subspace embedding [43, Theorem 1].

In our implementation of multisketching, we chose $S_1 \in \mathbb{R}^{p_1 \times n}$ as a CountSketch described above with $\varepsilon_1 = 0.9$, and $S_2 \in \mathbb{R}^{p_2 \times p_1}$ a Gaussian sketch with



(a) $n = 1,000,000$ rows



(b) $n = 10,000,000$ rows

Figure 5.1: Runtimes (in seconds) of QR factorizations of V with $\kappa(V) = 10^6$ for a fixed number of rows as the number of columns vary.

$p_2 = \lceil 74.3 \log(p_1) \rceil$ giving $\varepsilon_2 = 0.49$. Thus, S_2S_1 produced an embedding with $\varepsilon_L \approx 0.9490$, $\varepsilon_H \approx 1.8310$, and $d \approx 0.15$. It is easily verified that S_2S_1 is in line with Assumptions 5.9, and that both of S_1 and S_2 satisfy Assumptions 5.16, ensuring the analysis in Section 5.5.3 is relevant to the experiments. Runtimes of `rand_cholQR` did not include the time to generate the sketch, as this was assumed to be a fixed overhead time.

5.6.2 Performance and Numerical Results

Figure 5.1 shows the runtimes of each QR method for test problems with $n = 10^6$ and $n = 10^7$ rows, and $m = 10$–100 columns. Since `cholQR2` is typically expected to be the fastest algorithm, Table 5.1 shows the relative slowdown of each QR method compared to `cholQR2` averaged across each data point from Figure 5.1. Table 5.1 and Figure 5.1 indicate that `cholQR2` is indeed the fastest method in general, while multisketch `rand_cholQR` performs the closest to

Average Slowdown compared to `cholQR2`

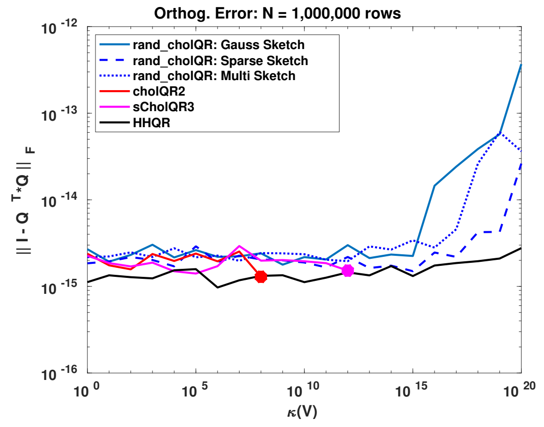
	1,000,000 rows	10,000,000 rows
`rand_cholQR`: Gauss Sketch	21.5%	23.1%
`rand_cholQR`: CountSketch	19.5%	5.3%
`rand_cholQR`: multisketch	7.1%	4.9%
`sCholQR3`	35.9%	33.8%
Householder	84.8%	83.4%

Table 5.1: Average slowdowns of each QR algorithm compared to `cholQR2`, taken from experiments shown in Figure 5.1. Smaller values indicate faster runtimes. Slowdowns for each QR algorithm are measured as (QR algorithm runtime$-$ `cholQR2` runtime)/(`cholQR2` runtime) $\times$ 100%

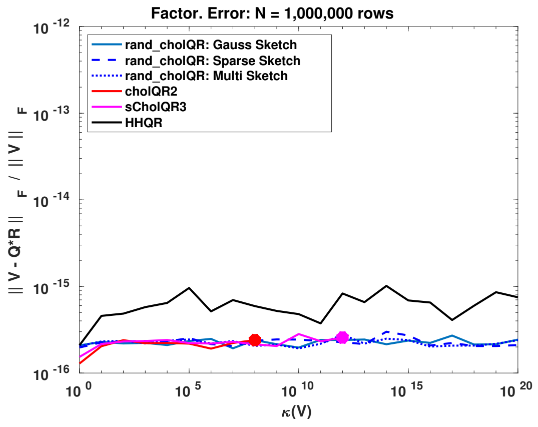
`cholQR2`, averaging only a 4.9–7.1% slowdown. Additionally, Figure 5.1 shows that in some cases, multisketch `rand_cholQR` actually outperforms `cholQR2`, specifically for $n = 10^6$ rows and $m = 70$ columns, and for $n = 10^7$ rows and $m =$ 70–80 columns. The most notable result is that for $n = 10^7$ rows and $m = 70$ columns, multisketch `rand_cholQR` is 4% faster than `cholQR2`. Multisketch `rand_cholQR` is significantly faster than `sCholQR3`, as evidenced by Figure 5.1, and both algorithms have the same $O(\mathbf{u})\kappa(V) < 1$ stability requirement.

Figure 5.2 shows the orthogonalization error $\|I - \hat{Q}^T\hat{Q}\|_F$ and the relative factorization error $\|V - \hat{Q}\hat{R}\|_F/\|V\|_F$ for condition number $\kappa(V) \in [1, 10^{16}]$. The results demonstrate that `rand_cholQR` maintains $O(\mathbf{u})$ orthogonality error and $O(\mathbf{u})\|V\|_2$ factorization error[4] while $\kappa(V) < O(\mathbf{u}^{-1})$, as predicted by Theorem 5.14, and is more robust than `cholQR2` and `sCholQR3`. In practice, it appears that `rand_cholQR` is stable even when V is numerically rank-deficient. In summary, Figures 5.1 and 5.2 demonstrate that multisketch `rand_cholQR` sig-

[4]This follows because $\|V - \hat{Q}\hat{R}\|_F/\|V\|_F = O(\mathbf{u})$.



(a) Orthogonality Error



(b) Relative Factorization Error

Figure 5.2: Orthogonality (left) and relative factorization error (right) of the QR factorization of a matrix V with varying condition number. To explicitly control $\kappa(V)$, $V := L\Sigma R^T \in \mathbb{R}^{n\times m}$ using random orthogonal matrices L, R, and a diagonal Σ with log-equispaced entries in the range $[\kappa^{-\frac{1}{2}}(V), \kappa^{\frac{1}{2}}(V)]$. Indicated by a large dot, lines for `cholQR2` and `sCholQR3` end at $\kappa(V) = 10^8$ and $\kappa(V) = 10^{12}$ respectively, as the methods fail beyond these points.

nificantly improves the robustness of `cholQR2` and `sCholQR3` at little to no cost, therefore making `rand_cholQR` a superior high-performance QR algorithm.

5.7 Conclusions

The results in Section 5.5.3 indicate that `rand_cholQR` using one or two sketch matrices orthogonalizes any numerically full-rank matrix V up to $O(\mathbf{u})$ error. This is a significant improvement over CholeskyQR2, which requires $\kappa(V) \lessapprox \mathbf{u}^{-1/2}$ to ensure a stable factorization. Our results for a single sketch apply for any ε-embedding with $\varepsilon \in [0, \frac{616}{634})$, covering nearly the entire possible range for ε-embeddings.

Our performance results in Section 5.6.2 indicate that the significantly better stability properties of `rand_cholQR` over `cholQR2` come at virtually no increase in the factorization time on a modern GPU. Additionally, `rand_cholQR` is theoretically just as stable and in practice more stable than `sCholQR3`, while being substantially faster. This is due to the fact that `rand_cholQR` and `cholQR2` incur the same number of processor synchronizations, while leveraging mostly BLAS-3 or optimized sparse matrix-vector routines for most of the required computation. In fact, `rand_cholQR` can perform better than `cholQR2` when using the multisketch framework. Of the sketching strategies considered, the multisketch framework is the most advantageous, likely because it requires little additional storage compared to `cholQR2`, and applying the sketches in this framework is extremely cheap.

Future work includes applying `rand_cholQR` to Krylov subspace methods that require tall-and-skinny QR factorizations, particularly block [32, 53], s-step [16, 37, 74], and enlarged Krylov methods [31], and further investigations into efficient multisketching implementations on a GPU, as our analysis is amenable to any multisketching strategy (not just a CountSketch followed by a dense Gaussian). In particular, applying the CountSketch matrix could potentially be optimized better than using a sparse-matrix vector multiplication by using a custom routine to add/subtract subsets of randomly selected rows in parallel using batched BLAS-1 routines, which should be investigated. Additionally, the performance of `randQR` and `rand_cholQR` using dense Rademacher sketch matrices in place of dense Gaussian sketches as in [1] should be investigated, as Rademacher sketches impose far lower storage requirements than a Gaussian sketch and can be generated much more efficiently.

CHAPTER 6

TWO-STAGE BLOCK ORTHOGONALIZATION TO IMPROVE PERFORMANCE OF s-STEP GMRES

As mentioned in Chapter 2.5, on modern heterogeneous computer architectures, GMRES' performance is often limited by its communication cost to generate the orthonormal basis vectors of the Krylov subspace. As described in Chapter 4, to address this potential performance bottleneck, its s-step variant orthogonalizes a block of s basis vectors at a time, potentially reducing the communication cost by a factor of s. Unfortunately, for a large step size s, the solver typically generates extremely ill-conditioned basis vectors, and to maintain stability in practice, a conservatively small step size is used, which limits the performance of the s-step solver.

To enhance the solver performance using a small step size, in this chapter, we introduce a two-stage block orthogonalization scheme. Similar to existing block orthogonalization schemes, the first stage of the proposed method operates on one block of s basis vectors at a time, but its objective is to maintain the well-conditioning of the generated basis vectors with a lower cost. The full orthogonalization of the basis vectors is delayed until the second stage when enough basis vectors are generated and thus obtains higher performance.

Our analysis shows the stability of the proposed two-stage scheme. The two-stage scheme offers performance improvements because while it still requires the same amount of computation as the original single-stage schemes, the proposed scheme performs the majority of the communication at the second stage, reducing the overall communication requirements. Our performance results with up to 192 NVIDIA V100 GPUs on the Summit supercomputer demonstrate that when solving a 2D Laplace problem, the two-stage approach can reduce the orthogonalization time and the total time-to-solution by the respective factors of up to $2.6\times$ and $1.6\times$ over the original s-step GMRES, which had already obtained the respective speedups of $2.1\times$ and $1.8\times$ over the standard GMRES. Similar speedups were obtained for 3D problems and for matrices from SuiteSparse [18].

6.1 Introduction

At each iteration, GMRES (Algorithm 1 or 2) generates a new basis vector for the Krylov subspace using a sparse-matrix vector multiply (`spmv`), typically combined with a preconditioner to accelerate its solution convergence rate. The basis vector is then orthonormalized to maintain the numerical stability of generating the Krylov subspace in finite precision and to compute the approximate solution that minimizes the ℓ_2 residual norm in the projection subspace, as described in detail in Section 2.3. As the subspace dimension grows, it becomes expensive to generate the orthonormal basis vectors in terms of both computation and storage. To reduce the costs of computing a large subspace, the iteration is restarted after a fixed number $m + 1$ of basis vectors are computed.

To orthogonalize the new basis vector at each iteration, GMRES uses BLAS-1 and BLAS-2 operations, which have limited potential for data reuse, and requires global synchronizations among all parallel processes. On modern heterogeneous computers, these communications (e.g., the cost of moving data through the local memory hierarchy and between parallel processes) can take much longer than the required computation time and can limit the performance of the orthogonalization process. As a result, when efficient and scalable `spmv` and preconditioners are available, the Krylov basis orthogonalization becomes a significant part of the iteration time and a performance bottleneck.

To reduce this potential performance bottleneck, communication-avoiding (CA) variants of GMRES [12, 37], based on s-step methods [19, 39], were proposed. To generate the orthogonal basis vectors of the Krylov projection subspace, the s-step GMRES utilizes two computational kernels:

1. the matrix powers kernel (MPK), described in detail in Section 4.3, used to generate the $s+1$ Krylov vectors by applying `spmv` and the preconditioner(s) s times, followed by

2. the block orthogonalization kernel that orthogonalizes a block of $s + 1$ basis vectors at once.

Since the block orthogonalization kernel performs most of its local computation using BLAS-3 operations and synchronizes only every s steps, compared to the standard GMRES, the s-step variant has the potential to reduce the communication cost of orthogonalizing the s basis vectors by a factor of s. This is a very attractive feature, especially on modern high-performance GPU clusters, where the communication can be significantly more expensive compared to computation.

Unfortunately, as discussed in Section 4.4, for large step sizes, MPK can generate extremely ill-conditioned s-step basis vectors. Hence, in practice, in order to maintain the stability of MPK, a conservatively small step size is used, which limits the performance advantages of s-step GMRES over the standard GMRES. In this chapter, we introduce a two-stage orthogonalization scheme to improve the performance of the s-step GMRES while still using a small step size s to maintain the stability of MPK.

There are two main contributions in this chapter. First, we analyze the current state-of-the-art block orthogonalization algorithms for s-step GMRES (Section 6.4). This motivates a new combination of recently developed block orthogonalization algorithms, which we call BCGS-PIP2. Though this new variant improves the performance of the original algorithms, it still has two synchronizations every s steps. Second, to further enhance the performance, we propose and extend the study to the two-stage approach, which delays one of the synchronizations until a large enough number of basis vectors, $\widehat{s}$, are generated to obtain higher performance (Section 6.5). In other words, though the two-stage approach performs about the same amount of computation as the original algorithms, it performs half of the local computation using the larger block size $\widehat{s}$ instead of the original step size s, hence increasing the potential for the data reuse. In addition, the two-stage approach performs only one synchronization at the first stage (every s steps), while delaying the other synchronization until the second stage (every $\widehat{s}$ steps). In particular, if we set the second step size same as the Krylov subspace projection dimension (i.e., $\widehat{s} = m$), the two-stage approach provides the potential to reduce the communication cost by a factor of two.

We demonstrate the potential of the new variant (BCGS-PIP2) and of the two-stage approach through numerical and performance experiments (Sections 6.6 and 6.8):

- We study the numerical stability of BCGS-PIP2 and that of the two-stage approach. We clarify the conditions that each of the algorithms requires to maintain its stability, and present numerical experiments to demonstrate the numerical properties of the algorithms.

- We implement the two-stage approach in Trilinos [70], which is a collection of open-source software packages for developing large-scale scientific and engineering simulation codes. Trilinos software stack allows the solvers, like s-step GMRES, to be portable to different computer architectures, using a single code base.

- We present GPU performance of s-step GMRES, combined with the two-stage approach. Our performance results on the Summit supercomputer demonstrate that when solving a 2D Laplace problem on 192 NVIDIA V100 GPUs, our two-stage approach can obtain speedups of $2.6\times$ and $1.6\times$ for orthogonalization and for the total time-to-solution, respectively, over the original s-step GMRES, which had already obtained the respective speedups of $2.1\times$ and $1.8\times$ over the standard GMRES. Similar speedups were observed for 3D model problems and for matrices from the SuiteSparse Matrix Collection.

The two-stage approach also alleviates the need of fine-tuning the step size for each problem on a specific hardware since a conservatively small step-size may be used for numerical stability while relying on the two-stage approach to obtain the performance improvement.

In addition to the notation established in Tables 2.1 and 2.2 consistent with the rest of the thesis, Table 6.1 lists additional notation convenient for this chapter and how it relates to s-step GMRES and our two-stage block orthogonalization algorithm. We reiterate that $\boldsymbol{Q}_{\ell:t}$ denotes the blocks column vectors of $\boldsymbol{Q}$ with the block column indexes ℓ to t, while we will use $q_{k:s}$ as the set of vectors with column indices k to s. Finally, $[Q, V]$ is the column concatenation of Q and V.

6.2 Related Work

Block orthogonalization is a critical component in many applications including linear or eigen solvers, and is an active research area. There are several combinations

notation	description
n	problem size
m	subspace dimension
s	step size (for the first stage)
$\widehat{s}$	second step size (for the second stage and $s \leq \widehat{s} \leq m$)
$v_k^{(j)}$	kth basis vector within block j
$\boldsymbol{V}_j$	jth block of s-step basis vectors including the starting vector, i.e., a set of $s+1$ vectors generated by MPK. Specifically, $\boldsymbol{V}_j = [v_{s(j-1)+1}, v_{s(j-1)+2}, \dots, v_{sj+1}]$ and $\boldsymbol{V}_0 = [v_0]$
$\underline{\boldsymbol{V}}_j$	same as $\boldsymbol{V}_j$ except excluding the last vector, which is the first vector of $\boldsymbol{V}_{j+1}$, i.e., a set of s vectors $\underline{\boldsymbol{V}}_j = [v_{s(j-1)+1}, v_{s(j-1)+2}, \dots, v_{sj}]$
$\widehat{\boldsymbol{V}}_j$	$\boldsymbol{V}_j$ after the first inter-block orthogonalization
$\widehat{\boldsymbol{Q}}_j$	$\boldsymbol{V}_j$ after the pre-processing stage
$\boldsymbol{Q}_j$	orthogonal form of $\boldsymbol{V}_j$

Table 6.1: Notation used in this chapter, and its specific relationship to s-step GMRES.

of block orthogonalization schemes [14], but, especially in terms of performance on current computer architectures, Block Classical Gram-Schmidt (BCGS) combined with some variants of Cholesky QR (CholQR) [67], is considered the state-of-the-art scheme. This chapter builds and extends on this combination. Some techniques that are relevant to this chapter include:

- CholQR computes the QR factorization of a tall and skinny matrix. Unfortunately, CholQR can fail when the condition number of the input matrix is greater than the reciprocal of the square-root of the machine precision (it computes the Cholesky factorization of the Gram matrix of the input basis vectors to be orthogonalized, and the Gram matrix has the condition number which is the square of the input vectors' condition number). Nonetheless, it performs well on current computer architectures because most of its local computation is based on BLAS-3 and it requires just one global synchronization. Hence, it is still used in practice but requires some remedies to maintain its stability. In addition, to maintain the orthogonality of the column vectors, it is often applied with reorthogonalization (referred to as CholQR twice, or equivalently CholQR2 for short).

- Shifted Cholesky QR [27] is introduced to avoid this numerical instability of CholQR and CholQR2. Though it may require one additional round of the orthogonalization, increasing the computational and communication costs of

CholQR2 by a factor of $1.5\times$, it has the stability guarantee as long as the input vectors are numerically full-rank.

- A mixed-precision variant of CholQR [81], which has similar stability properties as the shifted CholQR, was proposed. To ensure stability, the Gram matrix is accumulated in double the working precision. When working in double precision, it requires quadruple precision, which can be software-emulated through double-double precision arithmetic if quadruple precision is not supported by the hardware [35]. Though double-double arithmetic has high computational overhead compared to double precision, the mixed-precision CholQR does not increase the communication cost significantly. When the performance of CholQR is dominated by communication, it may obtain performance similar to the standard CholQR. Its application to the block orthogonalization has also been studied [82].

- There are low-synchronous variants of block orthogonalization algorithms that reduce the number of synchronizations and improve the performance of orthogonalization [14, 80]. These techniques require an efficient low-synchronous intra-block orthogonalization algorithm. Though there is a CA tall and skinny QR factorization algorithm that is unconditionally stable [20], its local computation is based on Householder QR (HHQR) factorization, which is mainly based on BLAS-1 or BLAS-2 and may obtain much lower performance than BLAS-3 based CholQR. Hence, in practice, these low-synchronous techniques rely on some variant of CholQR factorization for orthogonalizing each block.

Though some of the techniques mentioned above have improved stability, the s-step basis vectors, generated by MPK, can be extremely ill-conditioned for a large step size s, and in order to ensure the stability in practice, s-step GMRES still needs to use a small step-size. Since the performance of the orthogonalization may be limited by the multiple synchronizations required at every s steps, in this chapter, we look at avoiding or delaying some of the synchronizations, while using a small step size s to maintain stability. Moreover, the proposed two-stage approach may be combined with these previous approaches. In particular, random-sketching techniques have been recently integrated into CholQR [7]. We are investigating the potential of `rand_cholQR` to improve the stability of our block orthogonalization process.

6.3 s-Step GMRES

Algorithm 13 shows the pseudocode of s-step GMRES for solving a linear system $Ax = b$, which has been also implemented in the Trilinos software framework. The dominant costs of this algorithm occur in the inner loop in lines 5–12, which include both the MPK and the block orthogonalization procedure. Solving the least squares problem in lines 13–17 is much less costly, but the details of the least squares problem are outlined in this algorithm for completeness.

Algorithm 13 s-step GMRES

```
Input: coefficient matrix $A$, right-hand-side vector $b$, initial vector $x$, and appropriately-chosen "change-of-basis-matrix" $T$ (see [37, Section 3.2.3] for details)
Output: approximate solution $x$
1: $r = b - Ax$
2: $\gamma = \|r\|_2$
3: while not converged do
4:     $v_1 = r/\gamma$ and $h_{1,1} = 0$
5:     for $j = 1 : m/s$ do
6:         // Matrix Powers Kernel to generate new $s$ vectors
7:         for $k = 1 : s$ do
8:             $v_{k+1}^{(j)} = Av_k^{(j)}$
9:         end for
10:        // Block orthogonalization of $s + 1$ basis vectors
11:        $[\boldsymbol{Q}_j, \boldsymbol{R}_{1:j,j}] :=$ BlkOrth$(\underline{\boldsymbol{Q}}_{1:(j-1)}, \boldsymbol{V}_j)$
12:    end for
13:    // Generate the Hessenberg matrix $H$ such that $A\boldsymbol{Q}_{1:m/s} = \boldsymbol{Q}_{1:m/s+1}H$
14:    $H_{1:m+1,1:m} = \boldsymbol{R}_{1:m/s+1,1:m/s+1}T\boldsymbol{R}^{-1}_{1:m/s,1:m/s}$
15:    // Compute approximate solution with minimum residual
16:    $\hat{y} = \arg\min_{y \in \boldsymbol{Q}_{1:m/s+1}} \|\gamma e_1 - H_{1:m+1,1:m}y\|_2$
17:    $x = x + \boldsymbol{Q}_{1:m/s}\hat{y}$
18:    $r = b - Ax$
19:    $\gamma = \|r\|_2$
20: end while
```

Compared to the standard GMRES, this s-step variant has the potential of reducing the communication cost of generating the s orthonormal basis vectors by a factor of s, where the standard GMRES is essentially s-step GMRES with the step size of one. For instance, to apply `spmv` s times (Lines 7 to 9 of the pseudocode), several CA variants of the "matrix-powers kernel" (MPK) have been proposed [44]. In practice, `spmv` is typically applied together with a preconditioner to accelerate the convergence rate of GMRES. Although a few CA preconditioners of specific types have been proposed [30, 79], avoiding communication for other types of preconditioners is still an open research problem. To support a wide range of application needs, instead of CA MPK, Trilinos s-step GMRES uses a standard MPK (applying each `spmv` and preconditioner in sequence), and focuses on improving the performance of block orthogonalization by reducing its communication costs. Also, avoiding the global communication in orthogonalization could lead to a greater performance gain than CA MPK does, when scalable implementations of `spmv` and preconditioner are available. This motivates our study of the block orthogonalization in this chapter.

6.4 Block Orthogonalization

Algorithm 14 Block Classical Gram-Schmidt (BCGS) inter-ortho.

```
   Input: $\boldsymbol{Q}_{1:j-1}$ and $\boldsymbol{V}_j$
   Output: $\widehat{\boldsymbol{V}}_j$ and $\boldsymbol{R}_{1:j-1,j}$
1: // Orthogonalize $\boldsymbol{V}_j$ against $\boldsymbol{Q}_{1:j-1}$
2: $\boldsymbol{R}_{1:j-1,j} := \boldsymbol{Q}_{1:j-1}^T \boldsymbol{V}_j$          (GEMM for dot-products)
3: $\widehat{\boldsymbol{V}}_j := \boldsymbol{V}_j - \boldsymbol{Q}_{1:j-1}\boldsymbol{R}_{1:j-1,j}$      (GEMM for vector-update)
```

Algorithm 15 Block Classical Gram-Schmidt twice (BCGS2) for inter-ortho, combined with HHQR or CholQR2 intra-ortho.

```
    Input: $\boldsymbol{Q}_{1:j-1}$ and $\boldsymbol{V}_j$
    Output: $\boldsymbol{Q}_j$ and $\boldsymbol{R}_{1:j,j}$
 1: if $j > 1$ then
 2:     // First inter-block BCGS orthogonalization
 3:     $[\widehat{\boldsymbol{V}}_j, \boldsymbol{R}_{1:j-1,j}]$ := BCGS($\boldsymbol{Q}_{1:j-1}, \boldsymbol{V}_j$)
 4: else
 5:     $\widehat{\boldsymbol{V}}_j := \boldsymbol{V}_j$
 6: end if
 7: // First intra-block orthogonalization
 8: $[\widehat{\boldsymbol{Q}}_j, \boldsymbol{R}_{j,j}]$ := HHQR($\widehat{\boldsymbol{V}}_j$)    or    $[\widehat{\boldsymbol{Q}}_j, \boldsymbol{R}_{j,j}]$ := CholQR2($\widehat{\boldsymbol{V}}_j$)
 9: if $j > 1$ then
10:     // Second inter-block BCGS orthogonalization
11:     $[\widetilde{\boldsymbol{Q}}_j, \boldsymbol{T}_{1:j-1,j}]$ := BCGS($\boldsymbol{Q}_{1:j-1}, \widehat{\boldsymbol{Q}}_j$)
12:     // Second intra-block orthogonalization
13:     $[\boldsymbol{Q}_j, \boldsymbol{T}_{j,j}]$ := CholQR($\widetilde{\boldsymbol{Q}}_j$)
14:     $\boldsymbol{R}_{1:j-1,j} := \boldsymbol{T}_{1:j-1,j} + \boldsymbol{R}_{1:j-1,j}$
15:     $\boldsymbol{R}_{j,j} := \boldsymbol{T}_{j,j}\boldsymbol{R}_{j,j}$
16: end if
```

The block orthogonalization algorithm in s-step GMRES consists of two algorithms: the *inter* and *intra* block orthogonalization to orthogonalize the new block of $s + 1$ basis vectors against the already-orthogonalized previous blocks of vectors and among the vectors within the new block, respectively. There are several combinations of the inter- and intra-block orthogonalization algorithms [14], but the state-of-the-art inter-block algorithm is based on Block Classical Gram-Schmidt (BCGS), which is entirely based on BLAS-3 operations and requires only one global synchronization. As a result, BCGS (Algorithm 14) obtains superior performance on current computers.

To maintain orthogonality, in practice, BCGS is applied with re-orthogonalization (BCGS twice, or BCGS2). Algorithm 14 shows pseudocode of BCGS2, which has two algorithmic options for the first intra-block orthogonalization, while CholQR is used for the second intra-block orthogonalization. As we discuss in more detail below, with these combinations of the inter and intra block-orthogonalization algorithms, the orthogonality errors of the computed basis vectors $\boldsymbol{Q}_j$ can be bounded by $\mathcal{O}(\mathbf{u})$, where $\mathbf{u}$ is the machine precision. For our discussion of BCGS2 using different algorithms such as HHQR or CholQR2 for the first intra-block orthogonalization, we refer them as "BCGS2 with HHQR" or "BCGS2 with CholQR2", respectively.

6.4.1 BCGS2 with HHQR

When the column vectors of the input block-structured matrix $\boldsymbol{V}$ are numerically full-rank (i.e., $\kappa(\boldsymbol{V}) \max\{n, s\}\, \mathbf{u} < 1$), BCGS2 with HHQR in Algorithm 15 generates the orthonormal basis vectors $\boldsymbol{Q}$ with orthogonality error on the order of machine precision, i.e., $\|I - \boldsymbol{Q}^T\boldsymbol{Q}\|_2 = \mathcal{O}(\mathbf{u})$ [8, 9]. Unfortunately, for the small step size that we typically use within s-step GMRES (e.g., $s = 5$ is the default step size in Trilinos), the HHQR of $\widehat{\boldsymbol{V}}_j$ is based on BLAS-1 or BLAS-2 and requires $\mathcal{O}(s)$ global synchronizations, which often lead to the performance of HHQR and overall BCGS2 that is far below peak performance of modern high-performance clusters (e.g., based on the memory bandwidth).

There have been significant advances in the theoretical understanding of s-step Krylov methods [12]. However, though the orthogonality error bound to obtain the backward stability of GMRES has been established [29], to the authors knowledge, there are no known theoretical bounds on the orthogonality errors required to obtain the maximum attainable accuracy of s-step GMRES. Hence, in this chapter, we focus on block orthogonalization schemes that can maintain the $\mathcal{O}(\mathbf{u})$ orthogonality error, like BCGS2 with HHQR does (though this might not be needed to obtain the maximum accuracy of s-step GMRES), while improving the performance of the block orthogonalization.

6.4.2 BCGS2 with CholQR2

To generate the orthonormal basis vectors of $\widehat{\boldsymbol{V}}_j$ in step 8 of Algorithm 15, HHQR and CholQR2 (Algorithm 9) require about the same amount of computation. However, as the pseudocode in Algorithm 8 shows, in contrast to HHQR, CholQR is mainly based on BLAS-3 and requires only one synchronization. As a result, on current computer architectures, CholQR often obtains much higher performance than HHQR, and BCGS2 with CholQR2 is considered to be one of the state-of-the-art block-orthogonalization algorithms in terms of performance. Hence, in this chapter, we focus on BCGS2 with CholQR2 and discuss when it obtains the same stability as BCGS2 with HHQR, which is highly stable but lower performance.

In [67, 77], it was shown that when the condition number of the input vectors $\widehat{\boldsymbol{V}}_j$ is bounded as

$$c_1(\mathbf{u}, n, s)\kappa(\widehat{\boldsymbol{V}}_j)^2 < 1/2, \tag{6.1}$$

the orthogonality error of $\widetilde{\boldsymbol{V}}_j$ computed by the first pass of CholQR on line 1 of Algorithm 9 is bounded by

$$\|I - \widetilde{\boldsymbol{V}}_j^T \widetilde{\boldsymbol{V}}_j\|_2 \leq c_1(\mathbf{u}, n, s)\kappa(\widehat{\boldsymbol{V}}_j)^2, \tag{6.2}$$

where the scalar term $c_1(\mathbf{u}, n, s)$ is

$$c_1(\mathbf{u}, n, s) = 5\left(ns + s(s+1)\right)\mathbf{u}. \tag{6.3}$$

Condition (6.1) implies that the Cholesky factorization of the Gram matrix of $\widehat{\boldsymbol{V}}_j$ is numerically stable, and also that all the Krylov basis vectors generated by MPK are numerically full-rank (otherwise GMRES has converged).

When condition (6.1), and hence the orthogonality error bound (6.2), hold, we have the following theorem showing that CholQR2 is as stable as HHQR.

Theorem 6.1. *With the bound* (6.2) *and assumption* (6.1)*, the condition number of* $\widetilde{\boldsymbol{V}}_j$ *computed by the first CholQR (on Line 2 in Algorithm 9) is bounded by*

$$\kappa(\widetilde{\boldsymbol{V}}_j) < \sqrt{3}.$$

and hence, the orthogonality error of $\widehat{\boldsymbol{Q}}_j$ computed by CholQR2 satisfies

$$\|I - \widehat{\boldsymbol{Q}}_j^T \widehat{\boldsymbol{Q}}_j\|_2 = \mathcal{O}(\boldsymbol{u}).$$

Proof. Let $\sigma_1(G) \geq \cdots \geq \sigma_s(G)$ be the singular values of the Gram matrix of $\widetilde{\boldsymbol{V}}_j$, i.e., $G = \widetilde{\boldsymbol{V}}_j^T \widetilde{\boldsymbol{V}}_j$, and hence $\sigma_k(G) = \sigma_k(\widetilde{\boldsymbol{V}}_j)^2$ for $k = 1, \ldots, s$. Then, with the upper-bound (6.2) and assumption (6.1), along with Weyl's inequality [75], we have

$$\begin{cases} \sigma_1(\widetilde{\boldsymbol{V}}_j)^2 \leq 1 + \|\widetilde{\boldsymbol{V}}_j^T \widetilde{\boldsymbol{V}}_j - I\| \leq 1 + c_1(\mathbf{u}, n, s)\kappa(\widehat{\boldsymbol{V}}_j)^2 < 3/2 \\ \sigma_s(\widetilde{\boldsymbol{V}}_j)^2 \geq 1 - \|I - \widetilde{\boldsymbol{V}}_j^T \widetilde{\boldsymbol{V}}_j\| \geq 1 - c_1(\mathbf{u}, n, s)\kappa(\widehat{\boldsymbol{V}}_j)^2 > 1/2 \end{cases}$$

giving the above upper-bound on the condition number of $\widetilde{\boldsymbol{V}}_j$ and the orthogonality error of $\widehat{\boldsymbol{Q}}_j$. □

Hence, overall, when condition (6.1) is satisfied, BCGS2 with CholQR2 generates the basis vectors $\boldsymbol{Q}$ with orthogonality error on the order of the machine precision.

6.4.3 BCGS-PIP2

Algorithm 16 BCGS with Pythagorean Inner Product (BCGS-PIP).

```
    Input: $\boldsymbol{Q}_{1:j-1}$ and $\boldsymbol{V}_j$
    Output: $\widehat{\boldsymbol{Q}}_j$ and $\boldsymbol{R}_{1:j,j}$
1:  $\boldsymbol{R}_{1:j,j} := [\boldsymbol{Q}_{1:j-1}, \boldsymbol{V}_j]^T \boldsymbol{V}_j$
2:  $\boldsymbol{R}_{j,j} :=$ Chol$(\boldsymbol{R}_{j,j} - \boldsymbol{R}_{1:j-1,j}^T \boldsymbol{R}_{1:j-1,j})$
3:  $\widehat{\boldsymbol{V}}_j := \boldsymbol{V}_j - \boldsymbol{Q}_{1:j-1}\boldsymbol{R}_{1:j-1,j}$
4:  $\widehat{\boldsymbol{Q}}_j := \widehat{\boldsymbol{V}}_j \boldsymbol{R}_{j,j}^{-1}$
```

Algorithm 17 BCGS-PIP twice (BCGS-PIP2).

```
    Input: $\boldsymbol{Q}_{1:j-1}$ and $\boldsymbol{V}_j$
    Output: $\boldsymbol{Q}_j$ and $\boldsymbol{R}_{1:j,j}$
1: // First orthogonalization
2: $[\widehat{\boldsymbol{Q}}_j, \boldsymbol{R}_{1:j,j}]$ := BCGS-PIP($\boldsymbol{Q}_{1:j-1}, \boldsymbol{V}_j$)
3: // Second orthogonalization
4: $[\boldsymbol{Q}_j, \boldsymbol{T}_{1:j,j}]$ := BCGS-PIP($\boldsymbol{Q}_{1:j-1}, \widehat{\boldsymbol{Q}}_j$)
5: $\boldsymbol{R}_{1:j-1,j} := \boldsymbol{T}_{1:j-1,j}\boldsymbol{R}_{j,j} + \boldsymbol{R}_{1:j-1,j}$
6: $\boldsymbol{R}_{j,j} := \boldsymbol{T}_{j,j}\boldsymbol{R}_{j,j}$
```

Recently, a “single-synchronization” variant of BCGS with CholQR based on the Pythagorean Inner Product (BCGS-PIP) was proposed [14, 80]. The pseudocode of BCGS-PIP is shown in Algorithm 16, which orthogonalizes a new block of basis vectors $\boldsymbol{V}_j$ against the previously-orthonormalized blocks $\boldsymbol{Q}_{1:j-1}$. Instead of explicitly computing the Gram matrix of $\widehat{\boldsymbol{V}}_j$ for CholQR, BCGS-PIP computes it by updating the Gram matrix of $\boldsymbol{V}_j$ based on the block generalization of the Pythagorean theorem, allowing to orthonormalize the block vector $\boldsymbol{V}_j$ with a single synchronization.

Moreover, it was shown [14, Theorem 3.4] that if the previous basis vectors have been orthogonalized to satisfy[1]

$$\|I - \boldsymbol{Q}_{1:j-1}^T\boldsymbol{Q}_{1:j-1}\| = \mathcal{O}(\mathbf{u}), \tag{6.4}$$

[1]This is a much stronger condition than that required by [14, Theorem 3.4], but is satisfied by the algorithm discussed in this chapter.

and the MPK generates the next set of the block vector $\boldsymbol{V}_j$ such that

$$c_2(\mathbf{u})\kappa([\boldsymbol{Q}_{1:j-1}, \boldsymbol{V}_j])^2 < 1/2, \tag{6.5}$$

then the orthogonality error of $\widehat{\boldsymbol{Q}}_j$ computed by BCGS-PIP satisfies

$$\|I - [\boldsymbol{Q}_{1:j-1}, \widehat{\boldsymbol{Q}}_j]^T[\boldsymbol{Q}_{1:j-1}, \widehat{\boldsymbol{Q}}_j]\| \leq c_3(\mathbf{u})\kappa([\boldsymbol{Q}_{1:j-1}, \boldsymbol{V}_j])^2, \tag{6.6}$$

where $c_2(\mathbf{u})$ and $c_3(\mathbf{u})$ are two functions that behave asymptotically like $\mathcal{O}(\mathbf{u})$, similar to that given by (6.3).

Here, in order to generate the orthogonal basis vectors $\boldsymbol{Q}_j$ with orthogonality error of the order of the machine precision, we apply BCGS-PIP twice (BCGS-PIP2). The pseudocode of the resulting algorithm is shown in Algorithm 17. If conditions (6.4) and (6.5) are satisfied, we have the following theorem showing the stability of BCGS-PIP2.

Theorem 6.2. *With the bound* (6.6) *and the assumptions* (6.4) *and* (6.5)*, BCGS-PIP computes $\widehat{\boldsymbol{Q}}_j$ such that the condition number of the accumulated basis vectors $[\boldsymbol{Q}_{1:j-1}, \widehat{\boldsymbol{Q}}_j]$ satisfies,*

$$\kappa([\boldsymbol{Q}_{1:j-1}, \widehat{\boldsymbol{Q}}_j]) = \mathcal{O}(1), \tag{6.7}$$

and the resulting $\boldsymbol{Q}_j$ from BCGS-PIP2$(\boldsymbol{Q}_{1:j-1}, \boldsymbol{V}_j)$ satisfies

$$\|I - \boldsymbol{Q}_{1:j}^T\boldsymbol{Q}_{1:j}\| = \mathcal{O}(\boldsymbol{u}). \tag{6.8}$$

Proof. The proof is based on Weyl's inequality similar to that for Theorem 6.1. □

While BCGS2 with CholQR2 needs five synchronizations every s steps, this new variant BCGS-PIP2 needs just two synchronizations and reduces the total computational cost of the intra-block orthogonalization by a factor of $1.5\times$.

We note that when there are no previous blocks (i.e., $j = 1$), BCGS-PIP2 is CholQR2, which satisfies the condition (6.4) due to Theorem 6.1. Theorems 6.1 and 6.2 imply that when the required assumptions hold, BCGS followed by CholQR and BCGS-PIP are both stable pre-processing algorithms for BCGS2 with CholQR2 and BCGS-PIP2, respectively, and obtain $\mathcal{O}(1)$ condition number of the pre-processed block vectors.

Nevertheless, to obtain the best performance of the block orthogonalization, the step size s needs to be carefully chosen for each problem on a different hardware. Unfortunately, it is often infeasible to fine-tune the step size in practice, and for a large step size s, MPK can generate ill-conditioned basis vectors $\boldsymbol{V}_j$ with a large condition number. Hence, instead of fine-tuning the step size, in practice, a conservatively small step size is used to satisfy the conditions discussed in this section and avoid numerical instability (e.g., $s = 5$). Though BCGS-PIP2 improves the orthogonalization performance by reducing the number of the synchronizations, the small step size may still limit the performance gain that s-step methods can bring.

Algorithm 18 Two-stage BCGS-PIP2 block orthogonalization.

```
    Input: $\boldsymbol{Q}_{1:\ell-1}$, $s$, $\widehat{s}$
    Output: $\boldsymbol{Q}_{\ell:t}$ and $\boldsymbol{R}_{1:t,k:t}$
 1: // $t$ is last block column ID of the next big panel
 2: $t := \ell + \widehat{s}/s - 1$
 3: for $j = \ell, \ell+1, \dots, t$ do
 4:     // Matrix-Powers Kernel
 5:     if $j == \ell$ then
 6:         $v_1^{(j)} := q_{s+1}^{(j-1)}$
 7:     else
 8:         $v_1^{(j)} := \widehat{q}_{s+1}^{(j-1)}$
 9:     end if
10:     for $k = 1, 2, \dots, s$ do
11:         $v_{k+1}^{(j)} := Av_k^{(j)}$
12:     end for
13:     // First Stage (Preprocessing of each panel)
14:     [$\widehat{\boldsymbol{Q}}_j$, $\boldsymbol{R}_{1:j,j}$] := BCGS-PIP([$\boldsymbol{Q}_{1:\ell-1}$, $\widehat{\boldsymbol{Q}}_{\ell:j-1}$], $\boldsymbol{V}_j$)
15: end for
16: // First Stage (Block-orthogonalization of big panel)
17: [$\boldsymbol{Q}_{\ell:t}, \boldsymbol{T}_{1:t,\ell:t}$] = BCGS-PIP($\boldsymbol{Q}_{1:\ell-1}$, $\widehat{\boldsymbol{Q}}_{\ell:t}$)
18: $\boldsymbol{R}_{1:\ell-1,\ell:t} := \boldsymbol{T}_{1:\ell-1,\ell:t}\boldsymbol{R}_{\ell:t,\ell:t} + \boldsymbol{R}_{1:\ell-1,\ell:t}$
19: $\boldsymbol{R}_{\ell:t,\ell:t} := \boldsymbol{T}_{\ell:t,\ell:t}\boldsymbol{R}_{\ell:t,\ell:t}$
```

6.5 Two-Stage Block Orthogonalization

In order to improve the performance of block orthogonalization in s-step GMRES while using a small step size s, we propose a "two-stage" block orthogonalization process, shown in Algorithm 18. Instead of performing BCGS-PIP2 at every s steps to generate the orthonormal basis vectors, we call BCGS-PIP only once on the new $s+1$ basis vectors generated by MPK. The objective of this first stage is to pre-process the s-step basis vectors to maintain a small condition number of the generated basis vectors. In particular, since the s-step basis vectors $\widehat{\boldsymbol{Q}}_{1:j-1}$ are being "roughly" orthogonalized, when MPK generates the next s-step basis vector

$\boldsymbol{V}_j$ using the last vector of the block $\widehat{\boldsymbol{Q}}_{j-1}$ as the starting vecctor, the condition number of the accumulated basis vectors $[\widehat{\boldsymbol{Q}}_{\ell:j-1}, \boldsymbol{V}_j]$ is hoped to be roughly the same as that of $\boldsymbol{V}_j$ (we will show the numerical results in Section 6.6). Then once a sufficient number of basis vectors, $\widehat{s}$, are generated to obtain higher performance, we orthogonalize the $\widehat{s}$ basis vectors at once by calling BCGS-PIP for the second time, but now on a larger block size $\widehat{s}$ instead of the original step size s.

This two-stage approach in Algorithm 18 is similar to BCGS-PIP2 in Algorithm 17. To compare these two approaches, we distinguish between the blocks of two different block sizes s and $\widehat{s}$ for the first and second stages by referring to them as the "panels" and "big panels", respectively. Hence, the two-stage approach preprocesses the panels of s columns at a time, followed by BCGS-PIP on the big panel of $\widehat{s}$ columns. For the two extreme cases, with $\widehat{s} = s$ and $\widehat{s} = m$ the two-stage approach becomes the standard one-stage BCGS-PIP2 and BCGS-PIP on each panel followed by CholQR on the entire $m + 1$ basis vectors, respectively.

Compared to the original one-stage BCGS-PIP2, the two-stage approach performs about the same number of floating point operations, but it reduces the number of global synchronizations and performs the orthogonalization using a larger block size. In particular, BCGS2 with CholQR2 in Algorithm 15 or BCGS-PIP2 in Algorithm 17 performs five or two synchronizations at every s steps, respectively. In contrast, the two-stage approach in Algorithm 18 has only one synchronization every s steps, and one more synchronization every $\widehat{s}$ steps. Hence, with $\widehat{s} = m$, the two-stage approach reduces the number of synchronizations by a factor of $2\times$ (and could also reduce the amount of the required data movement through the local memory hierarchy). As a result, the two-stage approach often obtains much higher performance as we show in Section 6.8.

Next, we provide intuition behind the orthogonality errors of the two-stage approach. Since the two-stage approach uses BCGS-PIP on each panel and then on the big panel, we can apply error analysis similar to Section 6.4 but require the assumption (6.5) on the big panel. In particular, if the following condition on the big panel $\boldsymbol{V}_{\ell:t}$ is satisfied:

$$c_2(\mathbf{u})\kappa([\boldsymbol{Q}_{1:\ell-1}, \boldsymbol{V}_{\ell:t}])^2 < 1/2 \tag{6.9}$$

$$\text{with } \|I - \boldsymbol{Q}_{1:\ell-1}^T \boldsymbol{Q}_{1:\ell-1}\| = \mathcal{O}(\mathbf{u}),$$

then by [14, Theorem 3.4], the first BCGS-PIP pre-processes the big panel such that the generated basis vectors satisfy

$$\|I - [\boldsymbol{Q}_{1:\ell-1}, \widehat{\boldsymbol{Q}}_{\ell:t}]^T [\boldsymbol{Q}_{1:\ell-1}, \widehat{\boldsymbol{Q}}_{\ell:t}]\| \leq c_3(\mathbf{u})\kappa([\boldsymbol{Q}_{1:\ell-1}, \boldsymbol{V}_{\ell:t}])^2. \tag{6.10}$$

Hence, similar to Theorem 6.2, under condition (6.9), after the first stage, we expect the condition number of the accumulated big panels $[\boldsymbol{Q}_{1:\ell-1}, \widehat{\boldsymbol{Q}}_{\ell:t}]$ to be $\mathcal{O}(1)$, and when the big panel $\widehat{\boldsymbol{Q}}_{\ell:t}$ is orthogonalized by BCGS-PIP at the second stage, we expect an $\mathcal{O}(\mathbf{u})$ orthogonality error of $\boldsymbol{Q}_{1:t}$ by combining (6.10) with the $\mathcal{O}(1)$ condition number.

Specifically, by applying Weyl's inequality, we have the following theorem.

Theorem 6.3. *With the condition* (6.9) *and the bound* (6.10)*, the condition number of the big panel after the pre-processing is given by*

$$\kappa([\boldsymbol{Q}_{1:\ell-1}, \widehat{\boldsymbol{Q}}_{\ell:t}]) = O(1). \tag{6.11}$$

As a result, when the second stage calls BCGS-PIP on $[\boldsymbol{Q}_{1:\ell-1}, \widehat{\boldsymbol{Q}}_{\ell:t}]$*, the orthogonality error of the generated basis is bounded as*

$$\|I - \boldsymbol{Q}_{1:t}^T \boldsymbol{Q}_{1:t}\| = \mathcal{O}(\boldsymbol{u}). \tag{6.12}$$

The new condition (6.9) is on the condition number of the big panel, while the condition (6.5) for BCGS-PIP2 only required the condition number of each panel to be less than $\mathcal{O}(\mathbf{u}^{-1/2})$. The key feature of the two-stage approach is that the starting vector for MPK is the last column of $\widehat{\boldsymbol{Q}}_{j-1}$, which has been pre-processed by BCGS-PIP, whose objective is to maintain the small enough condition number of the big panel that satisfies (6.9).

In the next section, we study how the condition number of the pre-processed big panel grows, and compare the orthogonality errors of the two-stage approach with the standard algorithms. In particular, the numerical results show that

$$\kappa([\boldsymbol{Q}_{1:\ell-1}, \boldsymbol{V}_{\ell:j}]) \approx \kappa([\boldsymbol{Q}_{1:\ell-1}, \widehat{\boldsymbol{Q}}_{\ell:j-1}, \boldsymbol{V}_j]),$$

making assumption (6.9) required for the two-stage BCGS-PIP2 a similar stability requirement to assumption (6.5) required for the one-stage BCGS-PIP2.

6.6 Numerical Results

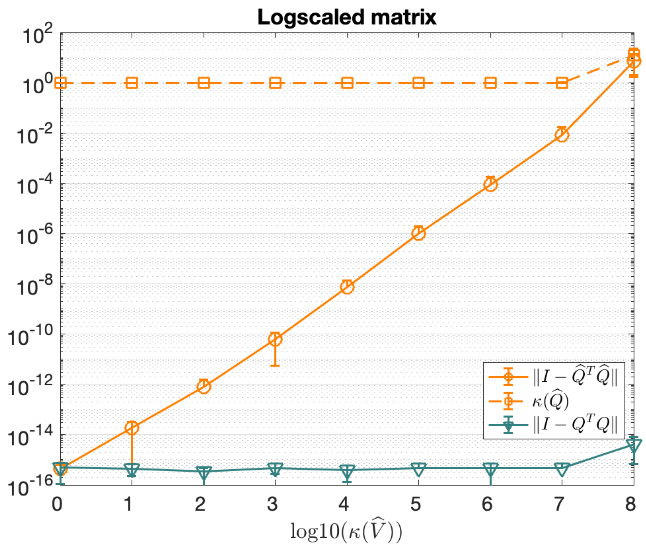
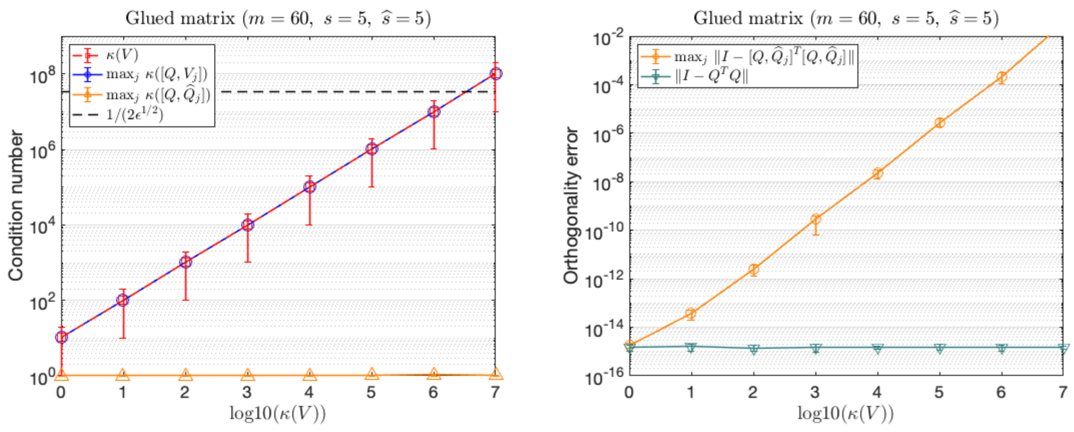


Figure 6.1: Orthogonality error and condition number with CholQR2 on 10^5-by-5 `Logscaled` matrix.

We compared the orthogonality errors of the proposed block orthogonalization schemes using the default double precision in MATLAB. We first study how the orthogonality errors grow with the condition number of the input vectors. For these studies, instead of studying the numerical properties of the proposed methods within s-step GMRES, we treat them as a general block orthogonalization scheme and use synthetic matrices as the input vectors such that we can control the condition number of the matrix easily. We start by showing that both CholQR2 and one-stage BCGS-PIP2 obtain $\mathcal{O}(\mathbf{u})$ orthogonality error given that conditions (6.1) and (6.5) are satisfied, respectively (Figures 6.1 and 6.2). We then show that the orthogonality errors of the two-stage approach are also $\mathcal{O}(\mathbf{u})$ when the condition (6.9) is satisfied (Figure 6.3). Since these synthetic matrices are generated using random numbers, we show the minimum, average, and maximum errors using ten different random seeds. Finally, we study how the condition numbers grow for the basis vectors generated by MPK using various positive indefinite matrices of dimension between 200,000 and 300,000 from the SuiteSparse Matrix Collection [18] in Figure 6.4. In order to maintain the stability of the original s-step method, we scaled the columns and then rows of the matrices by the maximum nonzero entries in the columns and rows (hence, all the resulting matrices are non-symmetric). For all of our experiments with MPK and s-step GMRES, we used monomial basis, even though using more stable bases, like Newton or Chebyshev bases, could reduce the condition number and improve the applicability of our approaches to a wider class of problems.

Figure 6.1 shows the condition numbers and orthogonality errors when CholQR2 is used to orthogonalize the 10^5-by-5 panel $\widehat{\boldsymbol{V}}_j$ of varying condition numbers (i.e., $\widehat{\boldsymbol{V}}_j := X\Sigma Y^T$ with random orthonormal X and Y, and diagonal matrix Σ with logspace singular values). It shows that as indicated by the bound (6.2), the or-



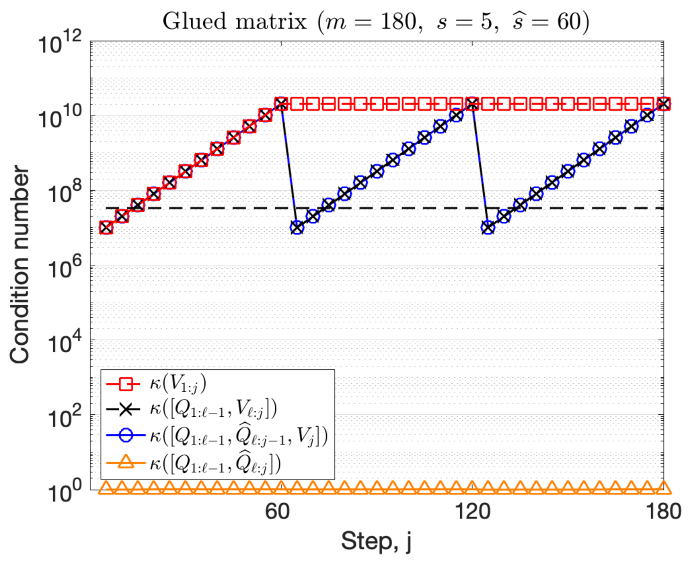
(a) Condition number. (b) Orthogonality errors.

Figure 6.2: Condition number and orthogonality error with one-step BCGS-PIP2 on `glued` matrix.

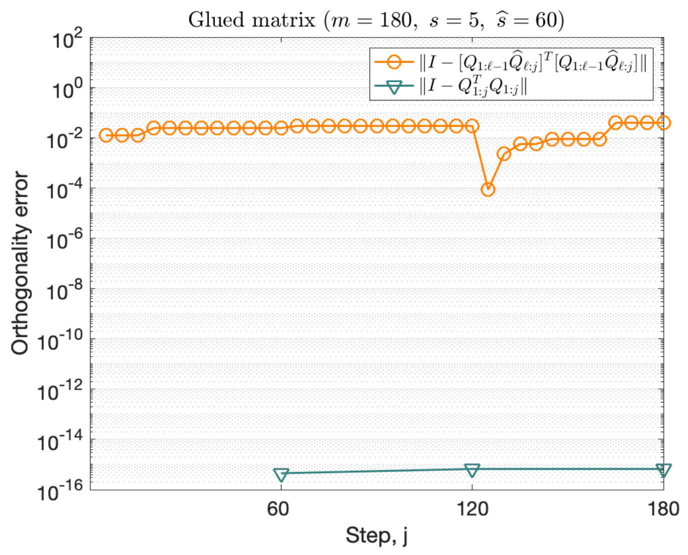
thogonality error of $\widehat{\boldsymbol{Q}}_j$ after the first CholQR grows as $\kappa(\widehat{\boldsymbol{V}}_j)^2\mathcal{O}(\mathbf{u})$. Hence, when $\kappa(\widehat{\boldsymbol{V}}_j) \lessapprox \mathbf{u}^{1/2}$, the condition number of $\widehat{\boldsymbol{Q}}_j$ stays $\mathcal{O}(1)$, and we obtain the $\mathcal{O}(\mathbf{u})$ orthogonality error of $\boldsymbol{Q}_j$ as indicated by Theorem 6.1.

Figure 6.2 shows the condition number and the orthogonality error when BCGS-PIP2 is used to orthogonalize the `glued` matrix that has the same specified order of the condition number for each panel and for the overall matrix. As expected, when the condition number of the input matrix is smaller than $\mathbf{u}^{-1/2}$, the orthogonality error of the basis vectors $\widehat{\boldsymbol{Q}}$ after the first BCGS-PIP is bounded by $\kappa(\boldsymbol{V})^2\mathcal{O}(\mathbf{u})$, and as a result, their condition number remained to be $\mathcal{O}(1)$. Consequently, after the second BCGS-PIP, the orthogonality error of the basis vector $\boldsymbol{Q}$ was $\mathcal{O}(\mathbf{u})$, which was the same error obtained by BCGS2 with CholQR2.

Figure 6.3 shows the orthogonality errors using the two-stage approach. The test matrix is the `glued` matrix, where each panel $\boldsymbol{V}_j$ has the condition number

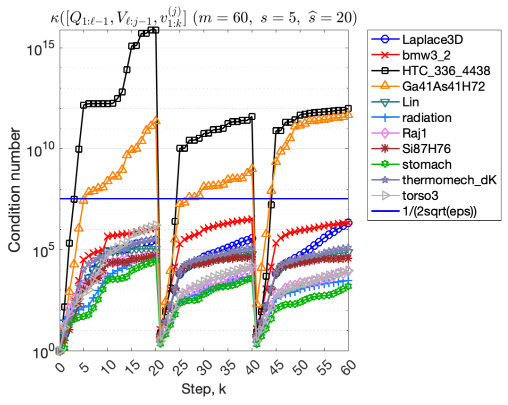


(a) Condition number (marker at every s steps).

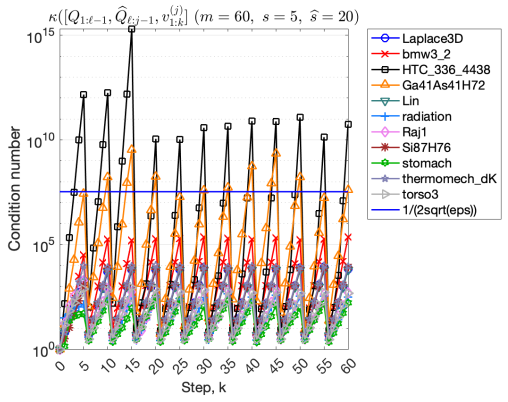


(b) Orthogonality errors (orange circle marker at every s steps, while green triangle marker at every $\widehat{s}$ steps).

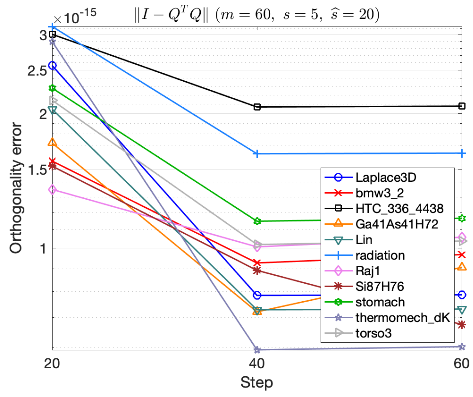
Figure 6.3: Condition number and orthogonality error using two-stage approach on `glued` matrix with $(n, m, \hat{s}, s) = (100000, 180, 60, 5)$.



(a) Condition number of $[\boldsymbol{Q}_{1:\ell-1}, \boldsymbol{V}_{\ell:j-1}, v^{(j)}_{1:k}]$ (marker at every step).



(b) Condition number of $[\boldsymbol{Q}_{1:\ell-1}, \widehat{\boldsymbol{Q}}_{\ell:j-1}, v^{(j)}_{1:k}]$ (marker at every step).



(c) Orthogonality error of $\boldsymbol{Q}$ (marker at every $\widehat{s}$ steps).

Figure 6.4: Condition number and orthogonality error with Krylov vectors generated by MPK.

$\mathcal{O}(10^7)$ but the condition number of $\boldsymbol{V}_{1:j}$ grows as $2^{j-1}\mathcal{O}(10^7)$. For this synthetic matrix with the pre-generated panels, after the first stage, the accumulated condition number of the panels $[\boldsymbol{Q}_{1:\ell-1}, \boldsymbol{V}_{\ell:j}]$ was still about the same as the condition number of the original big panel $\boldsymbol{V}_{\ell:j}$. Even though this synthetic matrix breaks the required condition (6.9), the pre-processing step managed to keep the $\mathcal{O}(1)$ condition number of the big panel $[\boldsymbol{Q}_{1:\ell-1}, \widehat{\boldsymbol{Q}}_{\ell:t}]$, and the overall orthogonality error of Q was $\mathcal{O}(\mathbf{u})$.

Finally, Figure 6.4 shows the condition number of the basis vectors that are generated by MPK combined with the two-stage block orthogonalization scheme. Unlike the pre-generated panels of the synthetic matrix in Figure 6.3, the s-step basis vectors of the big panel $\boldsymbol{V}_{\ell:t}$ are now generated by MPK, being interleaved with the pre-processing by BCGS-PIP. As a result, unlike what we have observed in Figure 6.3a, the accumulated condition number of $[\boldsymbol{Q}_{1:\ell-1}, \boldsymbol{V}_{\ell:j}]$ in Figure 6.4 did not increase significantly as more panels were appended, and except for the two matrices “HTC_336_4438” and “Ga41As41H72”, the condition number of the big panel satisfied the required condition (6.9). The condition number of the basis vectors are managed likely because BCGS-PIP now pre-processes the basis vectors $\widehat{\boldsymbol{Q}}_{j-1}$ before MPK generates the next set of the s-step basis vectors $\boldsymbol{V}_j$ such that the starting vector $v_1^{(j)}$ is roughly orthogonal to the space spanned by the previous panels $\underline{\boldsymbol{V}}_{\ell:j-1}$. Without pre-processing the basis vectors, the condition number will continue to increase, preventing us from using a large step size. Overall, after the second BCGS-PIP on big panel, the orthogonality errors of $\boldsymbol{Q}$ was $\mathcal{O}(\mathbf{u})$ for all the matrices tested.

6.7 Implementation

We have implemented all the block orthogonalization algorithms for s-step GMRES within the Trilinos software framework [34, 70]. Trilinos is a collection of open-source software libraries, called packages, for solving linear, non-linear, optimization, and uncertainty quantification problems. It is intended to be used as a building block for developing large-scale scientific or engineering applications. Hence, any improvement in the solver performance could have direct impacts to the application performance. In addition, Trilinos software stack provides portable performance of the solver on different hardware architectures, with a single code base. In particular, our implementation is based on Tpetra for distributed matrix and vector operations and Kokkos-Kernels for the on-node portable matrix and vector operations (which also provides the interfaces for the vendor-optimized kernels like NVIDIA cuBLAS, cuSparse, and cuSolver).

On a GPU cluster, our GMRES uses GPUs to generate the orthonormal basis vectors, where the matrices and vectors are distributed among MPI processes in 1D block row format (e.g., using a graph partitoner like ParMETIS). The operations with the small projected matrices, including solving a small least-squares problem, is redundantly done on CPU by each MPI process.

Our focus is on the block orthogonalization of the vectors, which are distributed in 1D block row format among the MPI processes. The orthogonalization process mainly consists of dot-products, vector updates, and vector scaling (e.g., $\boldsymbol{R}_{1:j-1,j} := \boldsymbol{Q}_{1:j-1}^T \boldsymbol{V}_j$ and $\boldsymbol{V}_j := \boldsymbol{V}_j - \boldsymbol{Q}_{1:j-1}\boldsymbol{R}_{1:j-1,j}$ of BCGS in Algorithm 14, and $\boldsymbol{Q}_j := \boldsymbol{V}_j \boldsymbol{R}_{j,j}^{-1}$ of CholQR in Algorithm 8, respectively). The dot-products $\boldsymbol{Q}_{1:j-1}^T \boldsymbol{V}_j$ requires a global synchronization among all the MPI processes, and the resulting matrix $\boldsymbol{R}_{1:j-1,j}$ is stored redundantly on all the MPI processes. Given the upper-triangular matrix, the vectors can be updated and scaled locally without any additional communication. All the local computations are performed by optimized kernels through Kokkos Kernels.

6.8 Performance Results

			$\widehat{s}$			
	GMRES	s-step	5	20	40	60
# iters	60251	60255	60255	60260	60280	60300
spmv	100.1	103.6	103.4	103.7	104.3	103.8
Ortho	150.4	128.6	102.8	96.9	75.2	61.1
Total	249.7	232.3	206.4	201.3	180.2	165.7

Table 6.2: Time-to-solution for 2D Laplace, $n = 2000^2$ on 4 NVIDIA V100 GPUs, with the two-stage approach using different values of second step size $\widehat{s}$, while the first step size is fixed as $s = 5$. The first two columns, "GMRES" and "s-step", show the time using the standard and s-step GMRES, respectively.

We now study the impact of different block orthogonalization schemes on the performance of s-step GMRES. We used the restart length of 60 (i.e., $m = 60$), and considered GMRES to have converged when the relative residual norm is reduced

	GMRES + CGS2				s-step + BCGS2-CholQR2				s-step + BCGS-PIP2				s-step + Two-stage($\widehat{s} = m$)			
# nodes	# iters	SpMV	Ortho	Total	# iters	SpMV	Ortho	Total	# iters	SpMV	Ortho	Total	# iters	SpMV	Ortho	Total
1	60251	63.5	100.2	164.3	60255	64.2	71.9	134.1	60255	66.2	54.5	117.8	60300	66.6	32.0	99.2
							1.4×	1.2×			1.8×	1.4×			3.1×	1.7×
2	60251	38.2	72.9	108.5	60255	35.2	43.9	78.9	60255	35.0	30.1	65.2	60300	35.7	18.8	54.7
							1.7×	1.4×			2.4×	1.7×			3.9×	2.0×
4	60251	27.7	59.8	85.6	60255	25.3	30.8	57.1	60255	25.2	19.9	45.4	60300	27.1	12.6	40.2
							1.9×	1.5×			3.0×	1.9×			4.7×	2.1×
8	60251	20.0	51.9	70.8	60255	20.0	27.2	47.0	60255	20.1	16.4	36.3	60300	19.5	10.8	30.6
							1.9×	1.7×			3.2×	2.0×			4.8×	2.3×
16	60251	17.1	48.0	64.3	60255	16.7	22.8	40.2	60255	17.1	14.1	30.9	60300	16.8	9.3	26.1
							2.1×	1.6×			3.4×	2.1×			5.2×	2.5×
32	60251	16.0	46.9	61.9	60255	15.6	22.3	38.2	60255	15.6	12.6	28.1	60300	16.0	8.7	24.5
							2.1×	1.8×			3.7×	2.2×			5.4×	2.5×

Table 6.3: Parallel Strong Scaling of time-to-solution with 9-points 2D Laplace, $n = 2000^2$. On each node, we launched six MPI processes (one MPI per GPU), and hence used 192 GPUs on 32 nodes. The table also shows the speedup gained using s-step and two-stage over standard GMRES for orthogonalization and total solution time.

by six orders of magnitude. We generated the right-hand-side vector such that the solution is a vector of all ones.

As discussed before, the step size s may need to be carefully chosen. For example, in Figure 6.3, our two-stage algorithm pre-processes the basis vectors at every fifth step to keep the condition number of the generated basis vectors small, but without the pre-processing step, the condition number will continue to increase exponentially after the fifth step. In practice, it is often infeasible to tune the step size as the condition number of the matrix could change significantly during the simulation. Hence, to avoid numerical instability of MPK, in practice, a conservative step size like $s = 5$ is used as the default step size. Since we are interested in improving the performance of block orthogonalization while using the small step size to maintain the stability of MPK, we use this default step size of $s = 5$ for all the performance results shown in this section, and study the effects of the two-stage algorithms on the performance of s-step GMRES.

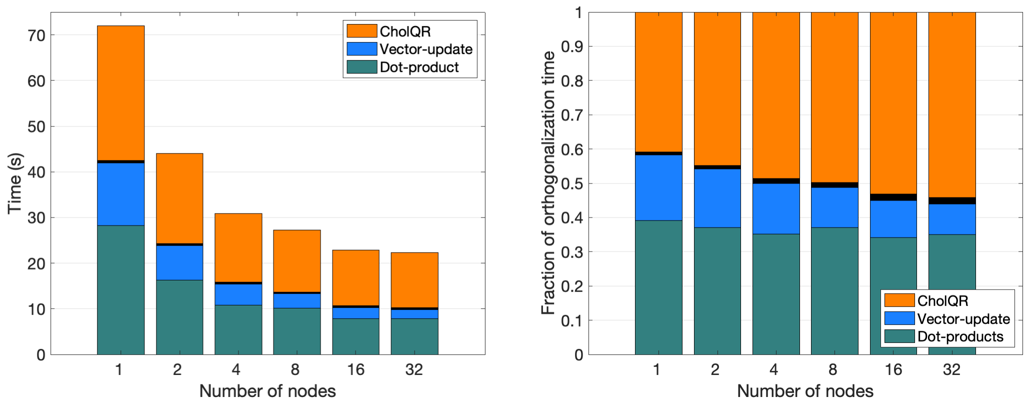
Table 6.2 shows the performance with the two-stage approach using different values of the second step size $\widehat{s}$. The performance tests were conducted on the Advanced System Technology Testbed named Vortex at the Sandia National Laboratories. Each node of Vortex has dual IBM Power 9 CPUs and four NVIDIA V100 GPUs. We compiled our code using GCC version 8.3 and CUDA version 11.0 compilers. As expected, the two-stage approach obtained higher performance using a larger step size, and it obtains the best performance when $\widehat{s} = m$. For these experiments, the pre-processing stage allowed us to maintain the numerical stability of the block orthogonalization process.

We conducted the remaining of our performance tests on the Summit supercomputer at Oak Ridge National Laboratory. Each compute node of Summit has two 21-core IBM Power 9 CPUs and six NVIDIA Volta V100 GPUs. The code was compiled using g++ compiler version 7.5 and NVIDIA CUDA 11.0, and linked to the IBM Engineering and Scientific Subroutine Library (ESSL) version 6.3 and Spectrum MPI version 10.4.

Table 6.3 shows the time to solution of s-step GMRES for solving 2D Laplace problem on a 5-point stencil (strong parallel-scaling), using different block orthogonalization schemes:

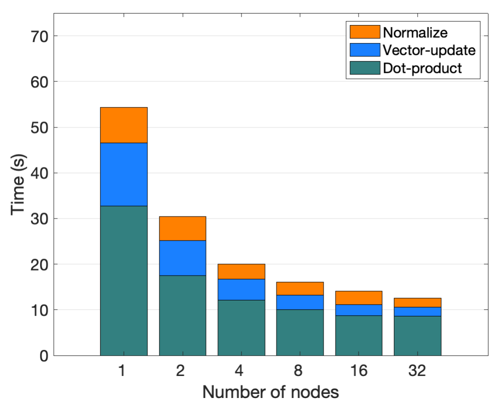
- Compared to BCGS2 with CholQR2 that the original s-step GMRES uses, BCGS-PIP2 reduces the number of synchronizations from five to two at every s steps, and lowers the computational cost of the intra-block factorization by a factor of $1.5\times$. The table shows that BCGS-PIP improved the performance, especially as the latency starts to become more significant on a larger number of nodes. Specifically, BCGS-PIP reduced the orthogonalization time by a factor of $1.3\times$ and $1.7\times$ over the original s-step method on 1 and 32 nodes, respectively, while achieving the respective speedups of $1.1\times$ and $1.3\times$ for the time-to-solution.

- Two-stage approach further reduces the orthogonalization time, and with $\widehat{s} = m$, it obtained the speedups of $1.7\times \sim 1.4\times$ over BCGS-PIP2, and hence, the time-to-solution was also reduced by factors of about $1.2\times$.

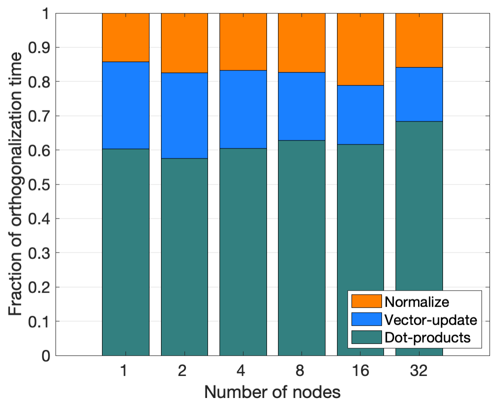


(a) Time in seconds. (b) Fraction of time.

Figure 6.5: Orthogonalization time breakdown using BCGS2 with CholQR2 for 2D Laplace, $n = 2000^2$.



(a) Time in seconds.

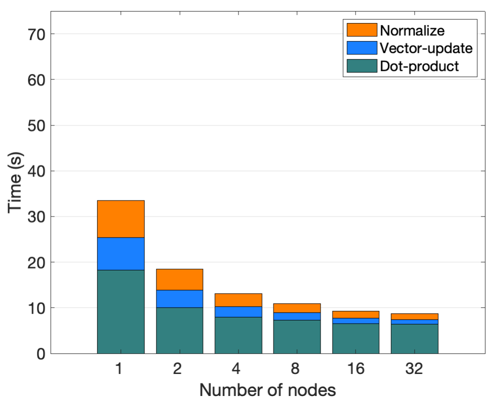


(b) Fraction of time.

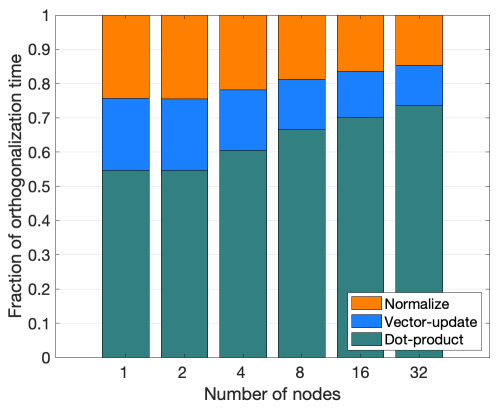
Figure 6.6: Orthogonalization time breakdown using BCGS-PIP2 for 2D Laplace, $n, = 2000^2$.

Figure 6.5 shows the breakdown of the orthogonalization time, using BCGS2 with CholQR2. For BCGS2, we show the time needed to compute the dot-products and vector-updates. On a larger number of GPUs, the orthogonalization time becomes dominated more by the dot-products which require global synchronizations, which are needed not only for BCGS2 but also for CholQR. In comparison, Figure 6.6 shows the breakdown of the orthogonalization time using BCGS-PIP2 where the orthogonalization time was reduced by avoiding global synchronizations and reducing the cost of intra-block orthogonalization of CholQR. Finally, Figure 6.7 shows the breakdown of the orthogonalization time using the two-stage approach with $\widehat{s} = m$. The two-stage approach further avoids global synchronizations and further reduced the orthogonalization time.

To summarize the performance studies, Table 6.4 compares the performance of s-step GMRES for 3D model problems and matrices from the SuiteSparse Matrix



(a) Time in seconds.



(b) Fraction of time.

Figure 6.7: Orthogonalization time breakdown using two-stage approach for 2D Laplace, $(n, \widehat{s}) = (2000^2, m)$.

Collection. Since these matrices have similar dimensions, the required orthogonalization time and the speedups gained using s-step with respective orthogonalization algorithms were similar. Though the ratio of the orthogonalization time over the iteration time depends on the required time for `spmv` with the matrices, BCGS-PIP reduced the orthogonalization and iteration time by factors of $1.8 \sim 2.0\times$ and $1.3 \sim 1.8\times$ over the original s-step GMRES, which had already obtained the respective speedups of $1.8 \sim 2.8\times$ and $1.3 \sim 1.8\times$ over the standard GMRES. The two-stage approach further improved the performance obtaining the respective speedups of $1.4 \sim 1.8\times$ and $1.1 \sim 1.3\times$ for the orthogonalization and time-to-solution.

Finally, Figure 6.8 shows a similar performance trend when a local Gauss-Seidel preconditioner (block Jacobi with Gauss-Seidel in each block [5]) was used. We used the multicolor Gauss-Seidel [22] from Kokkos Kernels to get good performance on the GPU.

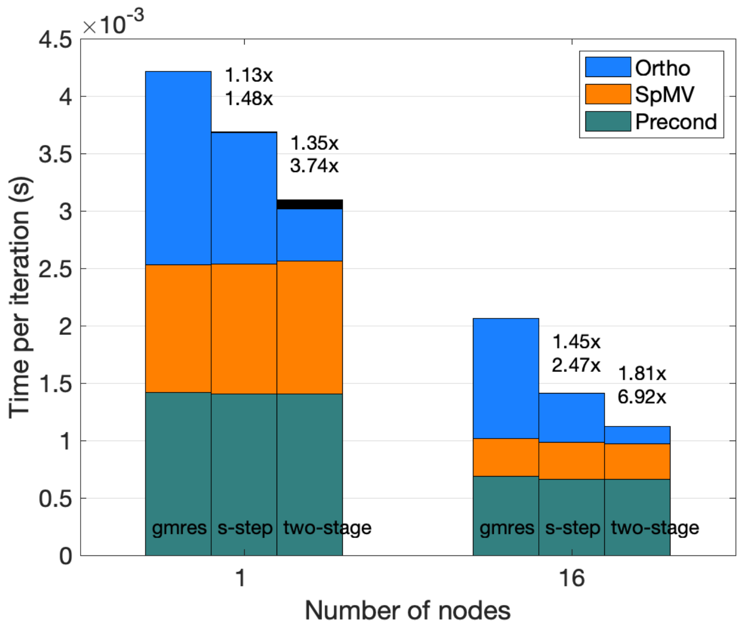


Figure 6.8: Time per iteration breakdown of s-step GMRES with Gauss-Seidel preconditioner for 2D Laplace, $(n, \widehat{s}) = (2000^2, m)$, along with the speedups over standard GMRES for the orthogonalization (bottom) and iteration (top) time.

6.9 Conclusions and Outlook

We surveyed the current state-of-the-art block orthogonalization algorithms for s-step GMRES, and this motivated a new method called BCGS-PIP2. We showed BCGS-PIP2 reduces the cost of the orthogonalization and improves the performance of s-step GMRES. Nevertheless, since s-step basis vectors can be extremely ill-conditioned for a large step size s, to maintain the stability in practice, a small step size needs to be used, which limits the performance gain that s-step GMRES can bring. In order to improve the performance of block orthogonalization using a small step size, we introduced a two-stage version of BCGS-PIP2, which preprocesses the s basis vectors at a time to maintain the well-conditioning of the basis vectors but delay the orthogonalization until enough basis vectors are generated to obtain higher performance. We presented numerical and performance results to demonstrate its potential.

We are exploring combining this two-stage approach with other techniques, such as random sketching. In a similar way that we developed `rand_cholQR` to significantly improve the stability of `cholQR2` without incurring a substantial performance hit in Chapter 5, integrating random sketching into our two-stage block orthogonalization algorithm may allow us to perform more stable operations at a lower computation cost, thereby removing many of the conditions required to guarantee its stability without significant performance overhead.

	# iters	Time / iter (ms) `spmv`	Ortho	Total
Laplace3D (Structured 3D model, SPD, $n = 100^3$, $nnz/n = 6.9$)				
standard	454	0.36	0.87	1.15
s-step	455	0.38	0.43 (2.0×)	0.76 (1.5×)
bcgs-pip2	455	0.37	0.24 (3.6×)	0.60 (1.9×)
two-stage	480	0.37	0.16 (5.4×)	0.52 (2.2×)
Elasticity3D (Structured 3D model, SPD, $n = 3 \cdot 100^3$, $nnz/n = 5.7$)				
standard	36	0.37	0.80	1.17
s-step	40	0.39	0.45 (1.8×)	0.88 (1.3×)
bcgs-pip2	40	0.37	0.23 (3.5×)	0.65 (1.8×)
two-stage	60	0.33	0.14 (5.7×)	0.51 (2.3×)
atmosmodl (CFD, numerically non-symmetric, $n = 1.5$M, $nnz/n = 6.9$)				
standard	213	0.31	0.79	1.06
s-step	215	0.37	0.38 (2.1×)	0.79 (1.3×)
bcgs-pip2	215	0.31	0.19 (4.2×)	0.50 (2.1×)
two-stage	240	0.35	0.14 (5.6×)	0.47 (2.3×)
dielFilterV2real (Electromagnet, symmetric indefinite, $n = 1.2$M, $nnz/n = 41.9$)				
standard	491856	0.36	0.99	1.22
s-step	493145	0.33	0.36 (2.8×)	0.66 (1.8×)
bcgs-pip2	491865	0.30	0.19 (5.2×)	0.48 (2.5×)
two-stage	491880	0.31	0.11 (9.0×)	0.42 (2.9×)
ecology2 (Circuit, SPD, $n = 1.0$M, $nnz/n = 5.0$)				
standard	3471536	0.25	0.80	1.04
s-step	3471540	0.24	0.34 (2.4×)	0.58 (1.8×)
bcgs-pip2	3471535	0.24	0.18 (4.4×)	0.42 (2.5×)
two-stage	3471540	0.25	0.10 (8.0×)	0.36 (2.9×)
ML_Geer, (Structural, numerically non-symmetric, $n = 1.5$M, $nnz/n = 73.7$)				
standard	1596564	0.28	0.74	1.00
s-step	1664400	0.29	0.37 (2.0×)	0.65 (1.5×)
bcgs-pip2	1613060	0.28	0.20 (3.7×)	0.47 (2.1×)
two-stage	1517460	0.28	0.11 (6.2×)	0.39 (2.6×)
thermal2 (Unstructured thermmal FEM, SPD, $n = 1.2$M, $nnz/n = 7.0$)				
standard	139188	0.26	0.81	1.06
s-step	139190	0.26	0.36 (2.2×)	0.61 (1.7×)
bcgs-pip2	139190	0.25	0.20 (4.1×)	0.44 (2.4×)
two-stage	139200	0.27	0.13 (6.2×)	0.39 (2.7×)

Table 6.4: Time per iteration for 3D model problems and matrices from SuiteSparse Matrix Collection on 16 Summit nodes; ParMETIS to distribute the matrix among 96 GPUs.

REFERENCES CITED

[1] Dimitris Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66:671–687, 2003. Special Issue on PODS 2001.

[2] Eric Anderson, Zhaojun Bai, Christopher Bischof, S. Blackford, James W. Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and Dannny C. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[3] Yehia Arafa, Abdel-Hameed A. Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. PPT-GPU: Scalable GPU performance modeling. *IEEE Computer Architecture Letters*, 18(1):55–58, 2019.

[4] Sheldon Axler. *Linear Algebra Done Right*. Undergraduate Texts in Mathematics. Springer International Publishing, 2023.

[5] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. Multigrid smoothers for ultraparallel computing. *SIAM Journal on Scientific Computing*, 33:2864–2887, 2011.

[6] Oleg Balabanov. Randomized Cholesky QR factorizations, 2022. arXiv:2210.09953.

[7] Oleg Balabanov and Laura Grigori. Randomized Gram–Schmidt process with application to GMRES. *SIAM Journal on Scientific Computing*, 44:A1450–A1474, 2022.

[8] Jesse L. Barlow. Some added flexibility for block classical Gram-Schmidt with reorthogonalization, 2021. Talk presented at the SIAM Conference on Applied Linear Algebra. Virtual.

[9] Jesse L. Barlow and Alicja Smoktunowicz. Reorthogonalized block classical Gram–Schmidt. *Numerische Mathematik*, 123:395—423, 2013.

[10] The Belos Project Team. The Belos Project Website. `https://docs.trilinos.org/dev/packages/belos/doc/html/classBelos_1_1PseudoBlockGmresSolMgr.html`. Accessed: 2021-12-01.

[11] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[12] Erin Carson. *Communication-avoiding Krylov subspace methods in theory and practice*. PhD thesis, EECS Dept., U.C. Berkeley, 2015.

[13] Erin Carson, Kathryn Lund, and Miroslav Rozložník. The stability of block variants of Classical Gram-Schmidt. *SIAM Journal on Matrix Analysis and Applications*, 42(3):1365–1380, 2021.

[14] Erin Carson, Kathryn Lund, Miroslav Rozložník, and Stephen Thomas. Block Gram-Schmidt algorithms and their stability properties. *Linear Algebra and its Applications*, 638:150–195, 2022.

[15] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In P. Widmayer, S. Eidenbenz, F. Triguero, R. Morales, R. Conejo, and M. Hennessy, editors, *Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 693–703, Berlin, Heidelberg, 2002. Springer. ICALP 2002.

[16] Anthony T. Chronopoulos and C. William Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[17] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[18] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1), December 2011.

[19] Eric de Sturler and Henk A. van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Applied Numerical Mathematics*, 18:441–459, 1995.

[20] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34:A206–A239, 2012.

[21] James W. Demmel, Laura Grigori, Maek Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34:A206–A239, 2012.

[22] Mehmet Deveci, Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Parallel graph coloring for manycore architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 892–901, 2016.

[23] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[24] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[25] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM Journal on Scientific Computing*, 14(2):470–482, 1993.

[26] Roland W. Freund and Noël M. Nachtigall. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60(3):315–340, 1991/92.

[27] Takeshi Fukaya, Ramaseshan Kannan, Yuji Nakatsukasa, Yusaku Yamamoto, and Yuka Yanagisawa. Shifted Cholesky QR for computing the QR factorization of ill-conditioned matrices. *SIAM Journal on Scientific Computing*, 42(1):A477–A503, 2020.

[28] Takeshi Fukaya, Yuji Nakatsukasa, Yuka Yanagisawa, and Yusaku 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, Los Alamitos, CA, 2014. IEEE Computer Society.

[29] Anne Greenbaum, Miroslav Rozložník, and Zdeněk Strakoš. Numerical behaviour of the modified Gram-Schmidt GMRES implementation. *BIT Numerical Mathematics*, 37:706–719, 1997.

[30] Laura Grigori and Sophie Moufawad. Communication avoiding ILU0 preconditioner. *SIAM Journal on Scientific Computing*, 37:C217–C246, 2015.

[31] Laura Grigori, Sophie Moufawad, and Frederic Nataf. Enlarged Krylov subspace conjugate gradient methods for reducing communication. *SIAM Journal on Matrix Analysis and Applications*, 37:744–773, 2016.

[32] Martin H. Gutknecht. Block Krylov subspace methods for linear systems with multiple right-hand sides: An introduction. In Abul Hasan Siddiqi, Iain S. Duff, and Ole Christensen, editors, *Modern Mathematical Models, Methods and Algorithms for Real World Systems*, chapter 10, pages 420–447. Anamaya Publishers, New Dehli, 2006.

[33] Martin H. Gutknecht and Thomas Schmelzer. Updating the QR decomposition of block tridiagonal and block Hessenberg matrices. *Applied Numerical Mathematics*, 58(6):871–883, 2008.

[34] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.

[35] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE, 2001.

[36] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, second edition, 2002.

[37] Mark Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, 2010.

[38] Intel. Intel math kernel library. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`. Accessed: 2022-04-24.

[39] Wayne D. Joubert and Graham F. Carey. Parallelizable restarted iterative methods for nonsymmetric linear systems. II: parallel implementation. *International Journal of Computer Mathematics*, 44:269–290, 1992.

[40] Michael Kapralov, Vamsi Potluru, and David Woodruff. How to fake multiply by a Gaussian matrix. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pages 2101–2110. Proceedings of Machine Learning Research, 2016.

[41] Julien Langou. *Solving large linear systems with multiple right-hand sides*. PhD thesis, INSA de Toulouse, 2003.

[42] Jörg Liesen and Zdeněk Strakoš. *Krylov subspace methods. Principles and analysis*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, 2013.

[43] Xiangrui Meng and Michael W. Mahoney. Low-distortion subspace embeddings in input-sparsity time and applications to robust linear regression. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, page 91–100, New York, 2013. Association for Computing Machinery.

[44] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 36:1–36:12, 2009.

[45] Yuji Nakatsukasa and Joel A. Tropp. Fast and accurate randomized algorithms for linear systems and eigenvalue problems, 2021. arXiv:2111.00113.

[46] NVIDIA. cuBLAS documentation. `https://docs.nvidia.com/cuda/cublas/index.html`. Accessed: 2021-12-08.

[47] NVIDIA. CUDA toolkit documentation. `https://docs.nvidia.com/cuda/`. Accessed: 2023-06-21.

[48] NVIDIA. cuSOLVER documentation. `https://docs.nvidia.com/cuda/cusolver/index.html`. Accessed: 2023-06-21.

[49] NVIDIA. cuSPARSE documentation. `https://docs.nvidia.com/cuda/cusparse/index.html`. Accessed: 2021-12-08.

[50] NVIDIA. GPU performance background user's guide - NVIDIA docs. `https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html`. Accessed: 2024-02-27.

[51] NVIDIA. Matrix multiplication background user's guide - NVIDIA docs. `https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html`. Accessed: 2024-02-27.

[52] NVIDIA. NVIDIA V100 Tensor Core GPU. `https://www.nvidia.com/en-us/data-center/v100`. Accessed: 2021-04-07.

[53] Dianne P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra and its Applications*, 29:293–322, 1980. Special Volume Dedicated to Alson S. Householder.

[54] Christopher C. Paige, Miroslav Rozložník, and Zdeněk Strakoš. Modified Gram-Schmidt (MGS), least squares, and backward stability of MGS-GMRES. *SIAM Journal on Matrix Analysis and Applications*, 28(1):264–284, 2006.

[55] Sivasankaran Rajamanickam, Seher Acer, Luc Berger-Vergiat, Vinh. Q. Dang, Nathan D. Ellingwood, Evan Harvey, Brian Kelley, Christian R. Trott, Jeremy J. Wilke, and Ichitaro Yamazaki. Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels, 2021. arxiv:2103.11991.

[56] Somaiyeh Rashedi, Ghodrat Ebadi, Sebastian Birk, and Andreas Frommer. On short recurrence Krylov type methods for linear systems with many right-hand sides. *Journal of Computational and Applied Mathematics*, 300:18–29, 2016.

[57] Vladimir Rokhlin and Mark Tygert. A fast randomized algorithm for overdetermined linear least-squares regression. *Proceedings of the National Academy of Sciences of the United States of America*, 105:13212–13217, 2008.

[58] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, 2nd edition, 2003.

[59] Yousef Saad and Martin H Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[60] Tamás Sarlós. Improved approximation algorithms for large matrices via random projections. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 143–152, Los Alamitos, CA, 2006. IEEE Computer Society.

[61] Valeria Simoncini and Efstratios Gallopoulos. An iterative method for nonsymmetric systems with multiple right-hand sides. *SIAM Journal on Scientific Computing*, 16:917–933, 1995.

[62] Valeria Simoncini and Efstratios Gallopoulos. Convergence properties of block GMRES and matrix polynomials. *Linear Algebra and its Applications*, 247:97–119, 1996.

[63] Aleksandros Sobczyk and Efstratios Gallopoulos. Estimating leverage scores via rank revealing methods and randomization. *SIAM Journal on Matrix Analysis and Applications*, 42:199–1228, 2021.

[64] Aleksandros Sobczyk and Efstratios Gallopoulos. pylspack: Parallel algorithms and data structures for sketching, column subset selection, regression, and leverage scores. *ACM Transactions on Mathematical Software*, 48:1–27, 2022.

[65] Peter Sonneveld and Martin B. van Gijzen. IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM Journal on Scientific Computing*, 31(2):1035–1062, 2009.

[66] Kirk Soodhalter. Block Krylov subspace recycling for shifted systems with unrelated right-hand sides. *SIAM Journal on Scientific Computing*, 38:A302–A324, 2016.

[67] Andreas Stathopoulos and Kesheng Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM Journal on Scientific Computing*, 23(6):2165–2182, 2002.

[68] TOP500. November 2023 — top500. `https://www.top500.org/lists/top500/2023/11/`. Accessed: 2024-02-27.

[69] Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[70] The Trilinos Project Team. *The Trilinos Project Website: `https://trilinos.github.io`*.

[71] Henk A. van der Vorst. Bi-CGStab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[72] Roman Vershynin. *High-Dimensional Probability: An Introduction with Applications in Data Science*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2018.

[73] Brigitte Vital. *Etude de quelques méthodes de résolution de problèmes linéaires de grande taille sur multiprocesseur*. PhD thesis, Université de Rennes I, 1990.

[74] Homer F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 9:152–163, 1988.

[75] Hermann Weyl. Das asymptotische Verteilungsgesetz der Eigenwerte linearer partieller Differentialgleichungen (mit einer Anwendung auf die Theorie der Hohlraumstrahlung). *Mathematische Annalen*, 71:441–479, 1912.

[76] David P. Woodruff. Sketching as a tool for numerical linear algebra. *Foundations and Trends in Theoretical Computer Science*, 10:1–157, 2014.

[77] Yusaku Yamamoto, Yuji Nakatsukasa, Yuka Yanagisawa, and Takeshi Fukaya. Roundoff error analysis of the Cholesky QR2 algorithm. *Electronic Transactions on Numerical Analysis*, 44:306–326, 2015.

[78] Ichitaro Yamazaki, Mark Hoemmen, Piotr Luszczek, and Jack Dongarra. Improving performance of GMRES by reducing communication and pipelining global collectives. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1118–1127, 2017.

[79] Ichitaro Yamazaki, Sivasankaran Rajamanickam, Erik G. Boman, Mark Hoemmen, Michael A. Heroux, and Stanimire Tomov. Domain Decomposition Preconditioners for Communication-avoiding Krylov Methods on a Hybrid CPU-GPU Cluster. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 933–944, 2014.

[80] Ichitaro Yamazaki, Stephen Thomas, Mark Hoemmen, Erik G. Boman, Katarzyna Świrydowicz, and James J. Elliott. Low-synchronization orthogonalization schemes for s-step and pipelined Krylov solvers in Trilinos. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing (PP)*, pages 118–128, 2020.

[81] Ichitaro Yamazaki, Stanimire Tomov, Tingxing Dong, and Jack Dongarra. Mixed-precision orthogonalization scheme and adaptive step size for improving the stability and performance of CA-GMRES on gpus. In *High Performance Computing for Computational Science - VECPAR*, volume 8969, pages 17–30, 2014.

[82] Ichitaro Yamazaki, Stanimire Tomov, Jakub Kurzak, Jack Dongarra, and Jesse Barlow. Mixed-precision block Gram-Schmidt orthogonalization. In *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2015. Article No. 2.