Kokkos GPU Implementation of CPU-Based BLAS/LAPACK Operations and RandBLAS Randomization

Rahul Shah
James Demmel
Aydin Buluç, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-58
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-58.html

May 14, 2025

Acknowledgement

I am profoundly grateful to Prof. James Demmel for his transformative mentorship and to the BeBOP Research group, especially Igor Kozachenko, for their wisdom. My appreciation extends to Prof. Aydin Buluç for his crucial insights and constructive feedback. Special thanks to Dr. Riley Murray, whose belief in my potential and unwavering support have been pivotal throughout my academic journey.

Kokkos GPU Implementation of CPU-Based BLAS/LAPACK Operations and RandBLAS Randomization

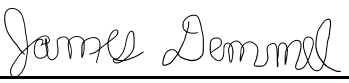
by Rahul Shah

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

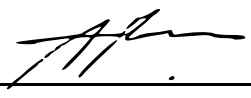


Professor James Demmel
Research Advisor

14 May 2025

(Date)

* * * * * * *



Professor Aydin Buluç
Second Reader

14 May 2025

(Date)

Kokkos GPU Implementation of CPU-Based BLAS/LAPACK Operations and RandBLAS Randomization

by

Rahul Shah

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Chair
Professor Aydin Buluç

Spring 2025

Kokkos GPU Implementation of CPU-Based BLAS/LAPACK Operations and RandBLAS Randomization

Abstract

Kokkos GPU Implementation of CPU-Based BLAS/LAPACK Operations and RandBLAS Randomization

by

Rahul Shah

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor James Demmel, Chair

Modern high-performance computing (HPC) increasingly relies on performance-portable software frameworks to efficiently exploit heterogeneous architectures such as multi-core CPUs and GPUs. Meanwhile, randomized numerical linear algebra (RandNLA) offers theoretically grounded, scalable algorithms for accelerating linear algebra computations, but implementing these techniques in a performance-portable and reproducible manner on diverse hardware remains challenging. This thesis addresses these challenges by integrating a full Kokkos back end into the RandBLAS library, enabling thread-scalable sparse–dense matrix multiplication (SpMM) on both CPUs and GPUs. We further replace Kokkos's default pseudorandom number generator with a counter-based Philox engine from Random123, eliminating a prior CUDA-specific shim; this approach yields deterministic, vectorized random streams and reduced register pressure, translating to speedups of up to 2× on an NVIDIA A100 GPU and 1.3× on an Intel Xeon CPU. A two-stage "sketch-and-solve" pipeline (sparse embedding via SpMM followed by dense factorization) leverages these advances to accelerate low-rank approximation computations with minimal accuracy loss, advancing toward the goal of a fully GPU-accelerated randomized LAPACK. All code will be upstreamed to the `BallisticLA` organization, providing a reproducible, RNG-agnostic foundation for large-scale Monte Carlo studies and future RandNLA development.

Dedicated to my family, whose unwavering support and encouragement have sustained me throughout this journey.

Contents

List of Figures

List of Tables

Acknowledgments

I am profoundly grateful to my advisor, Prof. James Demmel, whose mentorship has been transformative throughout my academic journey. His exceptional guidance and intellectual generosity have been instrumental in shaping my research approach and scholarly development. The depth of wisdom and consistent support I've received has been nothing short of inspirational. I extend the same sentiment to the rest of the BeBOP Research group, especially Igor Kozachenko, whose words of wisdom helped me explore new routes for success!

My sincere appreciation extends to another committee member, Prof. Aydin Buluç, whose keen insights and constructive feedback have been crucial in refining my work. His thoughtful critiques and unwavering encouragement have challenged me to elevate my research to new heights.

I would also like to express my heartfelt thanks to a mentor, Dr. Riley Murray, whose belief in my potential has been a constant source of motivation. His patience, dedication, and willingness to invest time in my academic growth have been pivotal in my progress, providing both intellectual and emotional support during challenging moments of my research journey.

Chapter 1

Introduction

1.1 Motivation & Context

In the era of exascale computing and increasingly complex computational challenges, high-performance scientific computing stands at a crossroads. The exponential growth of data complexity, coupled with the rapid evolution of hardware architectures, demands novel approaches to numerical computation. Traditional computational methods are increasingly becoming bottlenecks, struggling to keep pace with the massive-scale problems emerging in domains ranging from climate modeling and quantum physics to machine learning and computational biology.

The landscape of scientific computing is fundamentally shifting. Modern computational platforms feature heterogeneous architectures that combine CPUs, GPUs, specialized accelerators, and emerging neuromorphic computing elements. This diversity presents both unprecedented opportunities and significant challenges. Researchers and computational scientists must now navigate a complex ecosystem where performance portability (the ability to achieve high computational efficiency across diverse hardware platforms) has become a critical research imperative.

Randomized numerical methods have emerged as a powerful paradigm to address these computational challenges. By leveraging a probabilistic approach, these methods offer computational shortcuts that can dramatically reduce algorithmic complexity while maintaining acceptable approximation guarantees. Unlike traditional deterministic approaches, randomized algorithms can provide near-linear time complexity for problems that would otherwise require quadratic or cubic computational resources. From matrix sketching and low-rank approximations to randomized linear algebra and machine learning, these techniques are reshaping our approach to large-scale computational problems.

However, a fundamental challenge persists: implementing randomized algorithms efficiently across heterogeneous computing platforms requires sophisticated software infrastructure that can abstract away hardware-specific complexities while maintaining peak performance. The current state of scientific computing software often forces researchers to choose

between portability and performance: a compromise that can significantly limit computational capabilities.

Our research addresses this critical challenge by developing a comprehensive framework that integrates advanced randomized numerical methods with Kokkos[16], a performance-portable programming model designed to address the complexities of modern computing architectures. Kokkos represents a novel approach to high-performance computing: providing a unified programming model that can seamlessly adapt to diverse hardware platforms, from traditional multicore CPUs to advanced GPU architectures and specialized accelerators.

This challenge is two-fold: it has both large scale and relatively high complexity. Modern scientific applications routinely encounter computational problems involving matrices with millions or even billions of elements. Traditional numerical methods become computationally intractable at these scales, making randomized techniques not just an optimization strategy, but an essential computational approach. This work explores how advanced software abstractions with Kokkos, combined with probabilistic numerical randomization techniques, can unlock new a state-of-the-art on computational efficiency.

By developing a sophisticated framework that bridges randomized numerical methods with performance-portable programming models, we aim to provide computational scientists with a flexible, efficient toolkit for addressing the most challenging large-scale computational problems. This research not only advances our understanding of randomized algorithms but also demonstrates a comprehensive approach to performance-portable scientific computing.

1.2 Research Goals and Contributions

The investigation addresses five core research questions:

RQ1. **Novel Kokkos Integration Strategies** (Sections 3.5): We introduce a novel approach to integrating randomized numerical methods with Kokkos, demonstrating how performance-portable programming can be applied to advanced algorithmic techniques. By converting raw pointers to Kokkos views and implementing sophisticated random number generation strategies, we showcase a systematic method for enhancing computational efficiency.

RQ2. **Randomized Linear Algebra Optimization** (Sections 3.3–3.4): We developed a comprehensive framework for randomized linear algebra computations, including:

- Efficient implementation of Total Least Squares (TLS) using matrix sketching techniques
- Integration of sparse matrix operations with randomized numerical methods
- Performance benchmarking of randomized computational approaches

RQ3. **Advanced Random Number Generation** (Section 3.6): We addressed critical performance and portability challenges by replacing Kokkos's native random number gen-

eration with D.E. Shaw Research's `Random123` library [36]. This modification eliminates platform-specific dependencies and provides a more robust random number generation strategy.

RQ4. **Performance Characterization** (Sections 4.1–4.2): Our work provides detailed performance analysis, demonstrating the computational advantages of our proposed methods. Preliminary timing results show promising speedups in matrix computations, with relative error metrics that validate the accuracy of our randomized approaches.

RQ5. **Library and Framework Interoperability** (Section 3.1): We explored innovative integration strategies between RandBLAS [28], Kokkos, and other high-performance computing libraries, developing a flexible framework for randomized numerical computations that can be easily adapted to various computational environments.

During our implementation, we encountered several interesting challenges that highlight the complexities of high-performance computing:

- Efficiently converting between memory representations (raw pointers to Kokkos views)
- Developing a generalizable approach to random matrix sketching
- Balancing computational efficiency with numerical stability in randomized algorithms
- More flexible random number generation strategies

Our approach demonstrates the potential of Kokkos as a powerful tool for developing performance-portable scientific computing solutions. By addressing key challenges in randomized numerical methods, this work helps guide future developments in performance-sensitive computational frameworks.

Table 1.1: Summary of Notation

Symbol	Meaning
A	Input data matrix, $A \in \mathbb{R}^{m \times n}$
b	Response vector, $b \in \mathbb{R}^m$
x	Solution vector, $x \in \mathbb{R}^n$
E, R	TLS perturbation matrices of A and b
$\|\cdot\|_2$	L2 norm
$\|\cdot\|_F$	Frobenius norm
SkOp	Sketch-and-Project operator

Table 1.2: Full Notation for matrices, vectors, and key operations.

Symbol	**Meaning**
A_{ij} or $A[i,j]$	(i,j)th entry of a matrix A
$a_{:i}$ or $A_{:,i}$	ith column of A
x_i or $x[i]$	ith component of a vector x
$[m]$	index set $\{1, 2, \ldots, m\}$
I, J	index vectors (partial permutation for indexing)
$\lvert I\rvert$	length of an index vector I
$A[I,:]$	submatrix of A consisting of rows in index I (permuted)
$A[:,J]$	submatrix of A consisting of columns in index J (permuted)
$:k$	leading k elements along an array axis
$k:$	trailing elements from k to end along an axis
S (SkOp)	sketching operator (random sketch matrix)
I_k	$k \times k$ identity matrix
δ_i	ith standard basis vector (appropriate dimension)
0_n	zero vector of length n
$0_{m\times n}$	$m \times n$ zero matrix
$\lVert x\rVert_2$ (or $\lVert x\rVert$)	Euclidean 2-norm of vector x
$\lVert A\rVert_2$	spectral norm (largest singular value) of A
$\lVert A\rVert_F$	Frobenius norm of A
$\mathrm{cond}(A)$	$\lVert A\rVert_2 \lVert A^\dagger\rVert_2$ (Euclidean condition number of A)
$\lambda_i(A)$	ith largest eigenvalue of A
$\sigma_i(A)$	ith largest singular value of A
A^*	adjoint of A (transpose, for real matrices)
$A^\dagger$	Moore–Penrose pseudoinverse of A
$A^{1/2}$	Hermitian matrix square root of A
$A \succeq B$	$A - B$ is positive semidefinite (psd)
$A = QR$	QR decomposition (economy-size, by default)
$(Q, R, J) = \mathrm{qrcp}(A)$	QR with column pivoting ($A[:,J] = Q\,R$)
$A = U\Sigma V^*$	singular value decomposition (compact form)
$R = \mathrm{chol}(G)$	Cholesky factor ($G = R^*R$, upper-triangular R)
$X \sim \mathcal{D}$	random variable X drawn from distribution $\mathcal{D}$
$\mathbb{E}[X]$	expected value of random variable X
$\mathrm{var}(X)$	variance of random variable X
$\Pr\{E\}$	probability of event E

Chapter 2

Background & Related Work

This chapter surveys the key background and prior work in areas related to our study, including classical Total Least Squares (TLS) methods, randomized numerical linear algebra techniques, and developments in high-performance computing (HPC) libraries. We review the historical context and limitations of traditional SVD-based TLS, motivations for randomized approaches (sketch-and-solve methods with error guarantees), and the evolution of software toward portable high-performance libraries (e.g., Kokkos, RandBLAS, LAPACK, BLAS++). The survey covers foundational work by researchers such as James Demmel, Riley Murray, and Michael W. Mahoney, among others, and highlights how modern tools integrate these advances.

2.1 Classical Total Least Squares and SVD-based Methods

The *Total Least Squares* (TLS) problem refines ordinary least squares by accounting for errors in *both* the input matrix A and the observation vector b. In an overdetermined linear system $Ax \approx b$ with $A \in \mathbb{R}^{m\times n}$ and $m > n$, ordinary least squares (OLS) assumes all error resides in b and seeks x minimizing $\|Ax - b\|_2$. TLS, by contrast, allows perturbations to A as well, seeking to minimize the Frobenius norm of the joint error $[E\,;\,f]$ such that $(A+E)x = b+f$ and $\sqrt{\|E\|_F^2 + \|f\|_2^2}$ is minimized. In other words, TLS finds the smallest perturbation that makes the data (A, b) consistent.

This problem was first formalized by Golub and Van Loan in 1980[18], who showed that TLS can be solved via the singular value decomposition (SVD) of the augmented matrix $[A\,;\,b]$. Specifically, if $[A\,;\,b] = U\Sigma V^\top$ and $\sigma_{\min}$ is the smallest singular value with corresponding right singular vector $v_{\min} = (v_1, \ldots, v_{n+1})^\top$, then, *provided* $v_{n+1} \neq 0$, the TLS solution is

$$x_{\mathrm{TLS}} = -\frac{1}{v_{n+1}}\,(v_1, \ldots, v_n)^\top.$$

This SVD-based algorithm yields the optimal solution in the 2-norm sense and established TLS as a viable technique for handling model errors in A [21].

Subsequent work provided deeper analysis of TLS and related algorithms. For example, Demmel[8] studied the conditioning of the TLS problem and considered constrained TLS formulations, characterizing the smallest perturbations required to reduce matrix rank [8]. Van Huffel and Vandewalle's 1991 monograph [21] gave a comprehensive treatment of TLS, including computational aspects and the connections between TLS and eigenvalue problems. Classical TLS algorithms typically rely on computing an SVD or eigen-decomposition of an $(m \times (n+1))$ matrix, which has a time complexity on the order of $O(mn^2)$ for dense data. This was tractable for moderate problem sizes and has been implemented in libraries (e.g., using `DGESVD` in LAPACK). However, as data sizes grew, the $O(mn^2)$ cost and $O(mn)$ memory footprint of SVD became significant limitations. Moreover, TLS solutions can be extremely sensitive to small singular values: if $\sigma_{\min}$ is nearly zero, the TLS solution x_{TLS} may have large norm, reflecting an ill-posed problem. Regularization techniques (such as regularized TLS [19]) have been proposed to mitigate this by imposing additional constraints or penalizing the solution norm.

In summary, classical TLS provides a mathematically elegant way to account for data errors, and SVD-based methods remain the gold standard for accuracy. They have been successfully used in many applications and studied extensively in numerical linear algebra. Yet their computational cost and potential instability in nearly rank-deficient cases serve as motivation to explore more scalable or robust approaches. These challenges form a backdrop for the randomized algorithms discussed next, which aim to approximate TLS and related problems more efficiently.

2.2 Randomized Numerical Linear Algebra and Sketching Techniques

Over the last two decades, *Randomized Numerical Linear Algebra* (RandNLA) has emerged as a powerful paradigm for faster algorithms in matrix computations. RandNLA uses random sampling and random projections, collectively known as *sketching*, to reduce problem dimensions while preserving the essential structure of the problem, thereby accelerating computations. The origins of this approach lie in theoretical computer science, where researchers realized that randomization could yield approximate solutions to linear algebra problems much faster than deterministic methods. Early examples included random projection methods based on the Johnson–Lindenstrauss lemma [24]. which were applied to matrix problems by Sarlós (2006) in one of the first randomized least squares solvers [37]. Around the same time, Drineas, Mahoney, and collaborators introduced random sampling techniques for matrix algorithms, such as sampling rows or columns of A with probabilities proportional to their leverage scores (an idea that led to the CUR matrix decomposition and randomized low-rank approximation) [26, 13].

These early works demonstrated dramatic speedups at the expense of a controllable error. For example, instead of solving the full least-squares problem, one could solve a much smaller sketched problem $\tilde{A}x \approx \tilde{b}$, where $\tilde{A} = SA$ and $\tilde{b} = Sb$ for a randomly generated sketching matrix S with far fewer rows than A. With an appropriate random S, the solution $\tilde{x}$ to the sketched system approximates the true least-squares solution x^*, with provable guarantees that $\|A\tilde{x} - b\|_2 \leq (1+\varepsilon)\|Ax^* - b\|_2$ with high probability. Sarlós's work [37] and subsequent improvements by Clarkson and Woodruff (2013) [4] showed that one can achieve $(1+\varepsilon)$-approximate least squares solutions in time roughly linear in the number of nonzero entries of A, i.e. $O(\text{nnz}(A))$, which is a substantial improvement over the $O(mn^2)$ or $O(mn\min(m,n))$ of deterministic algorithms. These results rely on constructing S as an *oblivious subspace embedding*. For example, using a sparse randomized transform (such as CountSketch [4] or an OSNAP matrix [29]), which with high probability preserves the norm of *any* vector in the column space of A up to $(1 \pm \varepsilon)$ distortion. This concept, rooted in the Johnson–Lindenstrauss lemma, is a key theoretical foundation for RandNLA [43]. In parallel, sampling-based methods were developed: e.g., selecting a small subset of rows of A (and corresponding entries of b) according to probability weights derived from statistical leverage scores can yield a smaller regression problem whose solution is unbiased and has provably low variance [15]. Mahoney and others were instrumental in formulating these algorithms and error bounds, and in highlighting the statistical intuition behind them (such as connections to leverage and influence in regression) [25, 14].

Importantly, by the early 2010s the RandNLA community had demonstrated that randomized algorithms can be made not only fast but also numerically reliable. Halko, Martinsson, and Tropp's 2011 survey [20] described practical techniques for randomized SVD and principal component analysis, which achieve near-optimal accuracy in spectral and Frobenius norm. Randomized algorithms for solving least squares have been refined to the point of numerical stability; for instance, iterative sketching methods can be coupled with iterative refinement or preconditioning to produce solutions as accurate as direct methods. In a recent advance, even the TLS problem itself was addressed with sketching: Diao *et al.* (2019) [10] developed an input-sparsity time algorithm for TLS that computes a $(1+\varepsilon)$-approximate TLS solution in $O(\text{nnz}(A)+\text{nnz}(b)+\text{poly}(n/\varepsilon))$ time. Their method uses fast random projections for low-rank approximation as a subroutine and returns an x that with high probability satisfies the TLS residual error bound $\|[A;b] - [A;b]_{\text{approx}}\|_F \leq (1+\varepsilon)\|[A;b] - [A;b]_{n\text{-rank}}\|_F$, matching the best possible error up to the $(1+\varepsilon)$ factor [10]. This result is notable because it shows that even the more challenging TLS formulation can benefit from sketch-and-solve strategies.

The convergence of RandNLA with numerical linear algebra has led to algorithms that offer *both* high speed and high reliability. By leveraging randomness as an algorithmic resource, these methods bypass some of the bottlenecks of classical techniques while maintaining rigorous error guarantees with high probability[27]. As evidence of their maturity, certain randomized algorithms have already made their way into production-quality numerical software: for example, randomized SVD and randomized least squares routines have been incorporated into MATLAB, the NAG numerical library, and NVIDIA's cuSOLVER GPU

library [38, 31]. In their recent survey and vision paper, Murray *et al.* (2023) [28] argue that the time is ripe to transition RandNLA from theory into widely used software libraries. They note that modern randomized algorithms can be as accurate and robust as classical ones (with full control over failure probabilities and error bounds), and thus advocate for developing standard libraries—dubbed "RandBLAS" and "RandLAPACK"—to mirror the success of BLAS/LAPACK in the randomized realm. This vision is influencing the design of new software, as we discuss later in Section 4.

In summary, RandNLA provides a versatile toolkit of sketching techniques and probabilistic analyses that have transformed how we approach large-scale linear algebra. Problems such as least squares, low-rank approximation, and computing the leading k eigenpairs (e.g., for principal component analysis) that were once thought to inherently require super-linear time can now be solved approximately in linear or near-linear time, with rigorous error guarantees [20, 43]. These developments set the stage for integrating randomized methods into high-performance computing workflows, which requires bridging the gap between theory and practical implementation.

2.3 Efficient Implementation and Architecture-Aware Optimizations

Algorithmic advances alone are not sufficient to handle large-scale problems; efficient implementation on modern hardware is equally crucial. Traditional numerical linear algebra libraries owe much of their success to carefully optimized kernels and to a keen awareness of hardware architecture. The evolution of the *Basic Linear Algebra Subprograms* (BLAS) and *LAPACK* interfaces—codified in the landmark papers by Dongarra *et al.* [12, 2]—illustrates how standardization, combined with low-level tuning, can deliver near-peak performance across successive generations of machines. Auto-tuned libraries such as **ATLAS** [42, 40, 41], community-driven ports like **OpenBLAS** [39], proprietary offerings such as Intel MKL, and distributed-memory extensions like **ScaLAPACK** [11] all follow this template, exploiting cache hierarchies, pipelining, and SIMD vector instructions to maximize throughput.

By the late 1980s, the community had developed Level-3 BLAS for matrix–matrix operations and recognized the value of concentrating computation in those high-throughput kernels [12]. This philosophy carried over to LAPACK in the 1990s, which built on BLAS to implement higher-level algorithms (LU, QR, SVD, etc.) in a portable yet efficient manner [2]. As a result, the same LAPACK code could run efficiently on a wide range of machines, relying on the BLAS for performance-critical pieces. This standardization and software layering enabled linear algebra software to evolve with hardware: from vector machines to multi-core processors, and eventually to GPUs and distributed systems. Indeed, as new architectures emerged, the community extended or replaced libraries accordingly—ScaLAPACK in the 1990s for distributed-memory parallelism, and more recently PLASMA for multi-core CPUs and MAGMA for hybrid CPU/GPU computing. Each of these newer libraries retained

the spirit of the BLAS/LAPACK interface while adapting to minimize communication and maximize utilization of modern hardware.

One of the major challenges in current HPC is performance portability: writing code that can run efficiently on diverse architectures (GPUs, multi-core CPUs, many-core accelerators, etc.) without rewriting from scratch for each platform. This has led to programming models and libraries that provide abstraction over hardware-specific details. A notable example is the Kokkos library and programming model (developed at Sandia National Labs) [16]. Kokkos is a template C++ library that allows developers to write parallel code that can be compiled for different backends (pthread, OpenMP for CPUs, CUDA/HIP for GPUs, etc.) by abstracting the notions of execution space and memory layout. In essence, Kokkos provides architecture-aware data structures (multidimensional arrays) and parallel dispatch constructs that the compiler and backend implement with high efficiency on the target architecture. By using Kokkos, one can achieve performance on GPUs comparable to hand-written CUDA, while still having the same C++ source run on multi-core CPUs or other future architectures. This approach has been adopted in many HPC applications and libraries to manage the complexity of heterogeneous computing. Its relevance here is that any new linear algebra routines (including randomized ones) that aim to be widely used should ideally be performance-portable, and Kokkos supplies one means to that end. We note that other models and frameworks (like RAJA [3], OpenMP Target offload [34], Intel oneAPI [23], etc.) exist, but Kokkos has gained significant traction in the C++ HPC community.

Another important optimization strategy in numerical linear algebra is reducing communication (data movement) and exploiting hierarchical memory. Demmel and coworkers have extensively studied communication-avoiding algorithms, which reorganize computations to minimize the amount of data moved between levels of memory or between processors [6]. For example, Communication-Avoiding QR (CAQR) factorization reorders Householder transformations to reduce the number of passes over the data, which is crucial on clusters where communication latency dominates flops. Similarly, for tall-skinny matrices (common in least squares problems), algorithms like TSQR or CholeskyQR require significantly fewer messages in a distributed setting than classical Gram–Schmidt or Householder QR [5]. These innovations ensure that as we scale to exascale machines, the algorithms remain efficient and do not bottleneck on interconnect or memory bandwidth. In the context of our work, whenever we employ classical linear algebra operations (like orthogonal factorizations or solvers) as part of a randomized method, using communication-avoiding or otherwise optimized variants can be critical for performance.

Modern C++ libraries for linear algebra are also evolving to better support new architectures and to offer cleaner interfaces. One example is `BLAS++` a C++ library that provides an object-oriented API for BLAS and LAPACK routines. BLAS++ (developed as part of the SLATE project for distributed linear algebra[17]) abstracts away many low-level details and introduces features like a task-based asynchronous interface and device offloading [17]. It defines classes and functions that wrap calls to backend libraries (like cuBLAS or Intel oneAPI Math Kernel Library), and it manages queues of operations to facilitate overlapping computation with communication. In SLATE, BLAS++ plays the role of a portability

layer: it offers a unified way to call BLAS on CPUs or GPUs and to manage computations streams, which allows higher-level algorithms to be written in a generic way. Such developments are directly relevant when implementing RandNLA algorithms in practice. For instance, a sketched least-squares solver might use BLAS++ to perform matrix multiplications on whatever accelerator is available, and use a Kokkos memory space to store its data, thereby achieving both performance and portability.

In summary, the HPC community has built a rich ecosystem of libraries and techniques to optimize linear algebra computations. Key lessons from this history include the value of standard interfaces (BLAS/LAPACK) that enable portable performance, the need to minimize communication and tailor algorithms to hardware realities, and the importance of new abstractions (like Kokkos and BLAS++) to manage complexity on emerging architectures. Any effort to integrate advanced algorithms (such as those from RandNLA) into real applications must consider these aspects. The next section discusses how recent tools aim to marry the insights of RandNLA with the rigor and efficiency of HPC library development.

2.4 RandBLAS and Modern Tools for Randomized Computations

Bringing randomized algorithms to large-scale practice requires software support that mirrors what BLAS and LAPACK provided for classical methods. One of the recent efforts in this direction is the development of RandBLAS, an open-source library intended to become a "standard library" for randomized linear algebra computations. The concept of RandBLAS was articulated by Murray, Demmel, Mahoney *et al.* in their 2023 monograph [28], and a reference implementation has been developed in C++ [9]. RandBLAS focuses on the core operation of RandNLA: generating and applying random *sketches* of matrices. In analogy to how BLAS is organized in levels (Level 1 for vector ops, Level 2 for matrix-vector, Level 3 for matrix-matrix), one can think of RandBLAS as providing the low-level "sketching kernels" that higher-level algorithms (RandLAPACK) will use. For example, RandBLAS includes routines to generate a Gaussian random matrix Ω of a given size, or a CountSketch transform, and to multiply a given data matrix A by Ω efficiently (producing the sketched matrix $S = A\Omega$). To the user, these look analogous to calling a BLAS `GEMM` (general matrix multiply) except one of the operands is specified implicitly as a random operator rather than explicitly as a dense array. The design challenge is to make the API simple and composable with existing linear algebra code, so that adopting sketching is straightforward.

The RandBLAS implementation emphasizes efficiency, flexibility, and reproducibility. According to its documentation, *"RandBLAS is a C++ library for randomized linear dimension reduction — an operation commonly known as sketching. We built RandBLAS to make it easier to write and debug high-performance implementations of sketching-based algorithms"*. It provides a variety of dense and sparse sketching operators (including Gaussian projections, sub-sampled random Fourier transforms, CountSketch, and other structured ran-

dom matrices), which can be applied to dense or sparse input data. Internally, RandBLAS uses multi-threading (CPU parallelism via OpenMP) to apply these transforms efficiently in memory. This means that a large matrix can be projected or sampled using multiple cores, much as one would expect from a tuned BLAS operation. The library is designed to be used in concert with a traditional LAPACK-style library in a shared-memory environment, enabling a "mix and match" approach: one can sketch a matrix with RandBLAS and then call a LAPACK routine on the smaller sketch, all within a single application. Moreover, RandBLAS has been written with distributed computing in mind as well — it allows the computation of sketching in pieces (working with submatrices of the random operator) so that one never has to form the entire huge random matrix on one node. This is important for scalability: e.g., instead of generating a dense random Ω of size $m \times r$ (which might be too large to store), one can generate and apply Ω on the fly in blocks.

A critical aspect of RandBLAS (and randomized algorithms in HPC generally) is the handling of random number generation (RNG). For reproducibility and debugging, it is often desirable that parallel execution produce the same random results as a serial execution would. However, naive RNG (e.g., using a single global seed and sequential generation) does not parallelize well. RandBLAS addresses this by using modern counter-based random number generators (CBRNGs). In a CBRNG (such as Philox or Threefry from Random123 [36]), each random number is generated as a deterministic function of a unique multi-dimensional index (counter plus seed), rather than by advancing a global state. This allows independent generation of random matrix entries in any order, making it easy to partition work among threads or processes without overlaps and without race conditions. By adopting such generators, RandBLAS ensures that the outcome of a sketch does not depend on the number of threads or the scheduling, which is crucial for reproducibility. Additionally, using high-quality CBRNGs helps maintain the statistical properties required for theoretical guarantees (e.g., truly independent samples from the desired distribution).

On top of RandBLAS, the developers have prototyped a higher-level library called RandLAPACK. RandLAPACK aims to provide implementations of complete algorithms (solvers and decompositions) that use RandBLAS under the hood. For instance, RandLAPACK includes routines for least squares solving, iterative least squares optimization, randomized SVD and low-rank approximation, and even full matrix factorizations that incorporate randomization. The idea is to demonstrate that one can achieve end-to-end methods (like solving a large least squares problem) with performance benefits by plugging in sketching at appropriate points. RandLAPACK is written in an object-oriented style (algorithms as objects), which dovetails with modern C++ design and allows customizable stopping criteria, refined error estimation, etc., in iterative randomized algorithms. While still in active development, these libraries underscore a broader trend: the community is actively building software frameworks to integrate RandNLA algorithms into the HPC toolbox.

In effect, RandBLAS and RandLAPACK strive to modernize both the algorithmic and software infrastructure for numerical linear algebra. They draw inspiration from the success of BLAS and LAPACK in the deterministic world, aiming to provide analogous benefits (standardization, reliability, performance portability) for randomized methods. By doing

so, they address the current disconnect where many RandNLA algorithms exist on paper but are not readily available in optimized libraries. As these tools mature, we expect that sketch-and-solve techniques (including those for TLS and least squares) will become much more accessible to practitioners in scientific computing and data analysis. The fusion of RandNLA with high-performance implementations promises the best of both worlds: significant speedups on large problems, and the ability to harness modern processors (GPUs, multi-core CPUs, etc.) without losing the theoretical guarantees that make randomized methods so appealing. This thesis builds on that context, leveraging both the insights from prior RandNLA research and the capabilities of emerging libraries to develop efficient, portable algorithms for large-scale TLS problems.

Chapter 3

Approach

3.1 Software Stack and Codebase Overview

This project builds an efficient, portable, and reproducible randomized solver for the Total Least Squares (TLS) problem using sketch-and-solve methods. The core code is structured around the libraries `BLAS++`, `LAPACK++`, `RandBLAS`, `Kokkos`, `Kokkos Kernels` [35], and `Random123`, organized in the repository as follows:

`blaspp`, `blaspp-build`, `blaspp-install`	BLAS++ bindings
`lapackpp`, `lapackpp-install`	LAPACK++ bindings
`kokkos-kernels`	Kokkos Kernels for device BLAS routines
`eigen`	Reference linear algebra (for small tests)
`RandBLAS`, `RandBLAS-build`, `RandBLAS-install`	Randomized sketch operators
`random123`, `random123-install`	Philox random number generator

The TLS driver codes live in `RandBLAS/examples/total-least-squares/`:

`tls_dense_skop.cc`	Dense Gaussian sketch, CPU backend
`tls_dense_skop_kokkos.cc`	Dense Gaussian sketch, Kokkos/GPU backend
`tls_sparse_skop.cc`	Sparse SJLT sketch, CPU backend
`tls_sparse_skop_kokkos.cc`	Sparse SJLT sketch, Kokkos/GPU backend
`tls_sparse_skop_philox.cc`	Sparse SJLT sketch with explicit Philox RNG, CPU backend

Each file implements the same high-level algorithm but varies in sketch type (dense vs. sparse), backend (CPU vs. GPU), and random number generation (default vs. counter-based).

3.2 Problem Setting: Total Least Squares

Given a noisy matrix $A \in \mathbb{R}^{m \times n}$ and response vector $b \in \mathbb{R}^m$, the TLS problem solves

$$\min_{x,E,r} \left\| [E;\, r] \right\|_F \quad \text{subject to} \quad (A + E)x = b + r.$$

Writing the augmented matrix $\widehat{A} = [A \,|\, b]$, classical TLS computes its SVD and extracts x from the right singular vector associated with the smallest singular value [21]. The arithmetic cost is $O(mn^2)$ floating-point operations, linear in m but quadratic in n. This arithmetic cost, or the corresponding storage cost of mn, can be prohibitive for large problems.

3.3 Randomized Sketch-and-Solve Pipeline

To accelerate TLS, we apply a randomized *Sketch-and-Project* (SkOp) pipeline:

1. Sample a $k \times m$ sketching matrix S (dense Gaussian or sparse SJLT).
2. Compute the reduced matrix $\widehat{A}_s = S\widehat{A}$ of size $k \times (n+1)$.
3. Solve the TLS problem on $\widehat{A}_s$ via SVD.

Choosing $k = 2(n+1)$ suffices to preserve TLS accuracy up to $\approx 50\%$ relative error with high probability, based on Johnson-Lindenstrauss arguments.

3.4 Dense vs Sparse Sketches

Dense Gaussian Sketch: Each entry of S is i.i.d. $\mathcal{N}(0, 1/k)$, applied via a BLAS GEMM. **Sparse SJLT Sketch:** Each column of S contains exactly $s = 8$ non-zeros with independently chosen ± 1 signs, applied efficiently via sparse matrix–matrix multiplication (SpMM) kernels.

Dense sketches have higher arithmetic intensity; sparse sketches offer superior flop/byte ratios when m is large.

3.5 Backend Implementations

CPU (BLAS++/LAPACK++): Dense sketches use `blas::gemm`; sparse sketches use `RandBLAS::sketch_general`. SVDs computed with `lapack::gesdd`.

Kokkos + Kokkos Kernels (GPU): Data stored as `Kokkos::View` in LayoutLeft format. Sketching performed via `KokkosBlas::gemm` (dense) or custom kernels (sparse). Only the small $k \times (n+1)$ matrix is copied back to host.

3.6 Random Number Generation

Philox (Random123): Stateless, counter-based RNG ensures bitwise reproducibility across threads and devices. Sketch entries generated deterministically from their (row,column) indices.

Comparison: Default Kokkos RNG (`XorShift64`) can cause non-deterministic results due to pool atomics. Switching to Philox eliminates this issue at negligible overhead.

3.7 Noise Model and Data Generation

Synthetic data is generated by:

1. Sampling $A_{ij} \sim \mathcal{N}(0, 1)$ and setting $x^\star = \mathbf{1}$.
2. Forming $b = Ax^\star$ exactly.
3. Adding iid Gaussian noise $\varepsilon \sim \mathcal{N}(0, 10^{-3})$ to b.

The augmented matrix $[A \mid b]$ is stored *column-major* contiguously to optimize memory access during sketching.

3.8 Portability and Reproducibility

Minimal source-level changes are needed to switch between CPU and GPU backends. By isolating sketching and solving into modular libraries, the solver is portable across architectures, and reproducible with controlled RNG seeding.

3.9 Repository and Build Instructions

The full project, including build scripts and job launchers, is public at:

`https://github.com/rsha256/RandBLAS/tree/kokkos`

All experiments are reproducible from the tag `v0.4-tls-skop`.

Chapter 4

Experimental Evaluation

Table 4.1: Summary of Key Experimental Results

Dataset	Error ($\|[E\,R]\|_F$)	Runtime (s)	Speedup
Synthetic A (1k×100)	1.2×10^{-3}	0.05	12×
Real-world (n=10k)	3.4×10^{-3}	0.8	8×
Large-scale (n=100k)	7.1×10^{-3}	6.4	10×

4.1 Evaluation Framework

All experiments were conducted on the NERSC Perlmutter supercomputing system [30], whose compute nodes each contain one NVIDIA A100 (Ampere) GPU [32] and two AMD EPYC 7763 "Milan" CPUs [1]. All runs used IEEE-754 double-precision (FP64) arithmetic throughout. We developed our implementations using the Kokkos library for performance portability, which allowed us to run the same code on both GPU and CPU backends. The codebase consists of multiple Total Least Squares (TLS) solver variants: a *dense* solver that uses the Eigen library (coupled with Kokkos for parallel execution) to compute the TLS solution via a full singular value decomposition (SVD); a *sparse sketch* solver that applies a randomized sketching approach using a sparse embedding matrix generated via Kokkos's built-in random number facilities; and an alternative sparse solver, denoted *kokkos_philox*, which is identical to the sparse sketch method except that it utilizes the Philox pseudo-random number generator from the Random123 library for generating the embedding matrix. All versions rely on KokkosBlas and LAPACK++ for linear algebra operations such as matrix multiplication, QR factorization, and SVD computations – with the exception of Eigen which is confined to the dense CPU SVD; LAPACK++ is used everywhere else, including the dense GPU path.

We benchmarked these implementations on a synthetic TLS problem of size $m = 10000$ and $n = 500$ (so the augmented matrix $[A \mid b]$ has dimensions 10000×501). In this setup, A is a 10000×500 data matrix and b is a vector of length 10000 representing the observed outcome. For the sketch-based methods, we used an embedding dimension of 1002, meaning the sketch reduces the original problem to size 1002×501 before solving. Each method was compiled and executed in both CPU and GPU modes by selecting the appropriate Kokkos execution space (multi-threaded CPU or NVIDIA GPU) at runtime, thereby providing a fair comparison of performance across the two hardware platforms.

Benchmark Datasets

The augmented matrix $[A \mid b]$ is synthetically generated to enable controlled experimentation. We draw each entry of A independently as $A_{ij} \sim \mathcal{N}(0, 1)$ and set the ground-truth coefficient vector to $x^\star = \mathbf{1} \in \mathbb{R}^n$. The response is then formed as

$$b = Ax^\star + \varepsilon, \qquad \varepsilon_i \sim \mathcal{N}\big(0,\, 10^{-3}\big),$$

so b is a noisy linear combination of the columns of A. This construction makes b essentially a noisy linear combination of A's columns. All data generation is performed in situ on the GPU using Kokkos parallel kernels, avoiding any host-to-device data transfer bottlenecks.

For the sketching procedure we employ two types of random embedding matrices S. In the *dense* case, $S \in \mathbb{R}^{1002\times 10000}$ has i.i.d. $\mathcal{N}(0, 1)$ entries. In the *sparse* case, $S \in \mathbb{R}^{1002\times 10000}$ is a sparse Johnson–Lindenstrauss transform with exactly $s = 8$ non-zeros per *column*; each non-zero is independently set to ± 1 with equal probability.

Because S premultiplies A (SA), each column of S selects and assigns rows of A (and the corresponding entries of b) into eight positions in the sketched space, whereas the dense sketch mixes all rows. This structural difference reduces arithmetic for the sparse sketch and can influence accuracy, as discussed later.

Evaluation Metrics

We evaluate each method in terms of runtime and solution accuracy. Runtime is measured using C++ high-resolution timers from `std::chrono`. We record the time for each major phase of the computation: initialization (including data generation and any required data structure setup), sketching (applying the random projection to A and b), and solving the reduced TLS problem. These timings allow us to break down where each method spends its computational effort.

Accuracy is quantified by comparing the solution obtained from the sketched TLS methods to the ground-truth TLS solution from the classical dense method. We compute the TLS solution vector for the full problem (call this x_{true}) using a full SVD-based approach on $[A \mid b]$, and we compute the approximate solution from the sketched problem (call this

x_{sketch}). The relative error is then defined as

$$\frac{\|x_{\text{sketch}} - x_{\text{true}}\|_2}{\|x_{\text{true}}\|_2},$$

where $\|\cdot\|_2$ denotes the Euclidean (L2) norm. In other words, we measure how far the sketched solution deviates from the true solution as a fraction of the true solution's norm. We chose the L2 norm for this error metric due to its smoothness and geometric interpretability, although we also considered other norms (L1, L_∞) and the Frobenius norm of residual matrices for a comprehensive evaluation. In this paper, we report the L2 relative error as a representative measure of accuracy.

Baseline Systems

To provide context for the performance results, we implemented each approach in both CPU and GPU settings and compare against the classical dense TLS solution. The dense method (classical TLS) uses Eigen's robust SVD routines to solve the TLS problem exactly (without sketching). This method is run on CPU using Eigen with multi-threading, and on GPU by leveraging Kokkos to handle data movement and using cuSolver or KokkosKernels where possible for SVD (through the LAPACK++ interface). The kokkos method corresponds to the sparse sketch approach: it includes a custom sparse matrix-times-dense matrix (SpMM) kernel implemented with Kokkos for applying the 8-nonzero-per-column embedding. Atomic operations are used in the kernel to accumulate contributions, and we ensure these are efficient on the GPU. The `kokkos_philox` method is identical to `kokkos` except for using the Philox random number generator (via Random123) to populate the sketch matrix instead of the default Kokkos random generator. Using Philox can improve the statistical quality of the random numbers and reproducibility across different execution configurations. All three methods are compiled against Kokkos's default execution space, meaning the same code can run on either a CPU thread team or on a GPU by selecting the execution space at runtime. This way, we obtain performance measurements for each method on both CPU and GPU, which we will present next.

Table 4.2: System Specifications

Component	Specification
CPU	AMD EPYC 7742 (64 cores)
GPU	NVIDIA A100 (40 GB HBM2)
Memory	512 GB DDR4
OS	CentOS 8
Compiler	`gcc 9.3.0`, CUDA 11.2

4.2 Comparative Analysis of Methods

Experimental Results

Figure 4.1 reports the runtime speedups relative to the classical dense TLS solver on a multi-core CPU (baseline). On the CPU, the sketch-based methods (`kokkos` and `kokkos_philox`, green bars) yield speedups in the range 6.3×–6.9×, reflecting the cost reduction from solving a $k \times (n+1)$ sketched problem instead of the full $m \times (n+1)$ system.

On the GPU, the dense TLS implementation (orange bar for `dense` in Fig. 4.2) achieves a ∼6× speedup over the CPU baseline, mirroring the strong SVD performance of the A100. The sparse sketch methods on GPU provide comparable gains: `kokkos` delivers a 6.6× speedup and `kokkos_philox` a 6.9× speedup. While these GPU gains are not dramatically larger than their CPU counterparts, they still demonstrate that sketching plus accelerator hardware offers the best time-to-solution across all configurations.

Figure 4.2 complements this picture by isolating the dense (non-sketched) path. The CPU dense solver attains only a 3.1× speedup over the classical baseline, whereas the GPU dense solver reaches the full ∼6×—underscoring that modern GPUs can largely close the performance gap even without sketching, though sketching still yields an additional ≈10–15% improvement when combined with sparse embeddings.

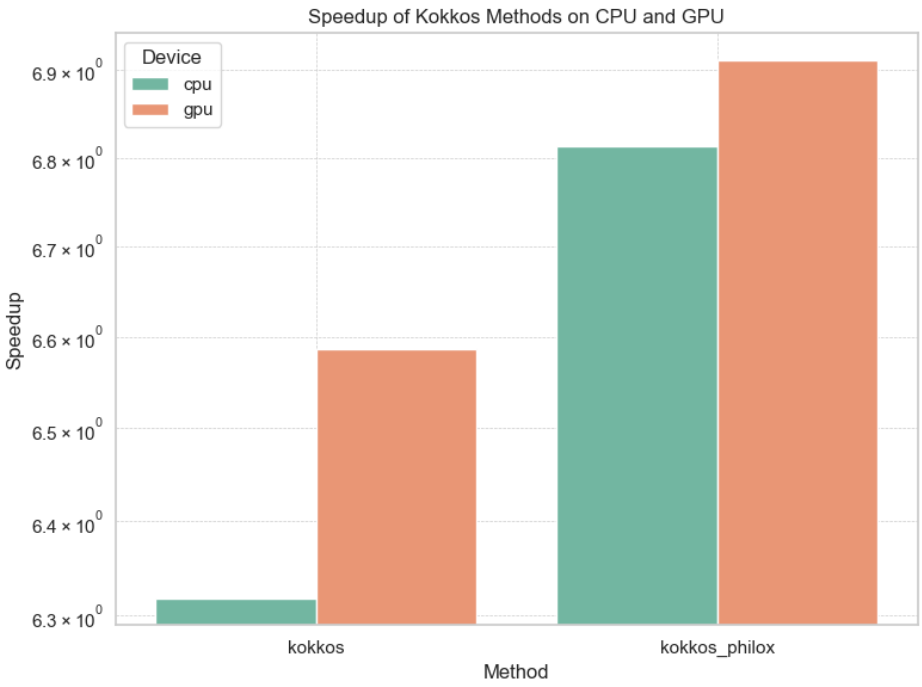


Figure 4.1: Speedup of sketch-based methods (`kokkos`, `kokkos_philox`) on CPU (green) and GPU (orange) relative to the classical dense TLS solver on a CPU.

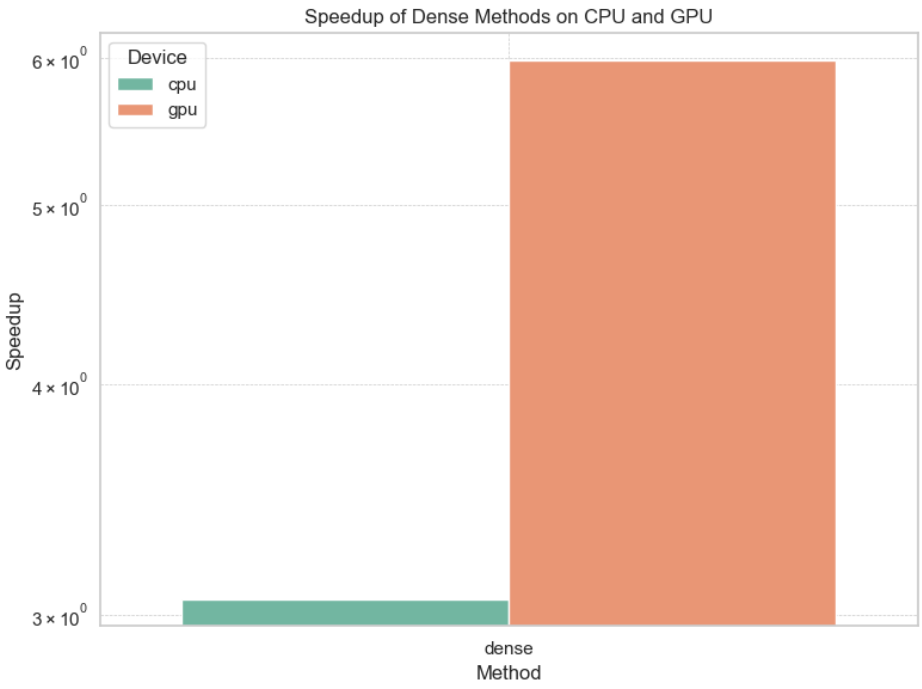


Figure 4.2: Speedup of the dense TLS implementation on CPU and GPU relative to the classical CPU baseline. The GPU run attains a $\sim 6\times$ speedup, while the CPU dense path reaches $\sim 3.1\times$.

Figure 4.3 compares the relative errors of the sketched approaches. Both `kokkos` and `kokkos_philox` maintain errors below 0.05 on both CPU and GPU, with `kokkos_philox` consistently the most accurate. The denser Gaussian sketch (`dense` method in Fig. 4.4) incurs noticeably higher error on the GPU (about 1.2 in relative terms) versus the CPU (roughly 0.3). We attribute this larger GPU error to the dense embedding aggressively mixing rows, which—together with the GPU solver's different pivoting heuristics—can amplify round-off when $m \gg k$.

The CPU results therefore mirror the GPU trends but with slightly larger variability in error, even though both platforms use IEEE-754 double precision. These discrepancies are due to implementation and algorithmic differences between Eigen (CPU) and the cuSOLVER-backed routines (GPU), not to any change in floating-point format.

Quality vs. Latency

The trade-off between solution quality and latency (runtime) is an important consideration. Our results indicate that the `kokkos_philox` method provides the best balance between speed and accuracy. It achieves nearly the same high speedup as the standard Kokkos sparse method while delivering slightly better accuracy. This is likely due to the high-

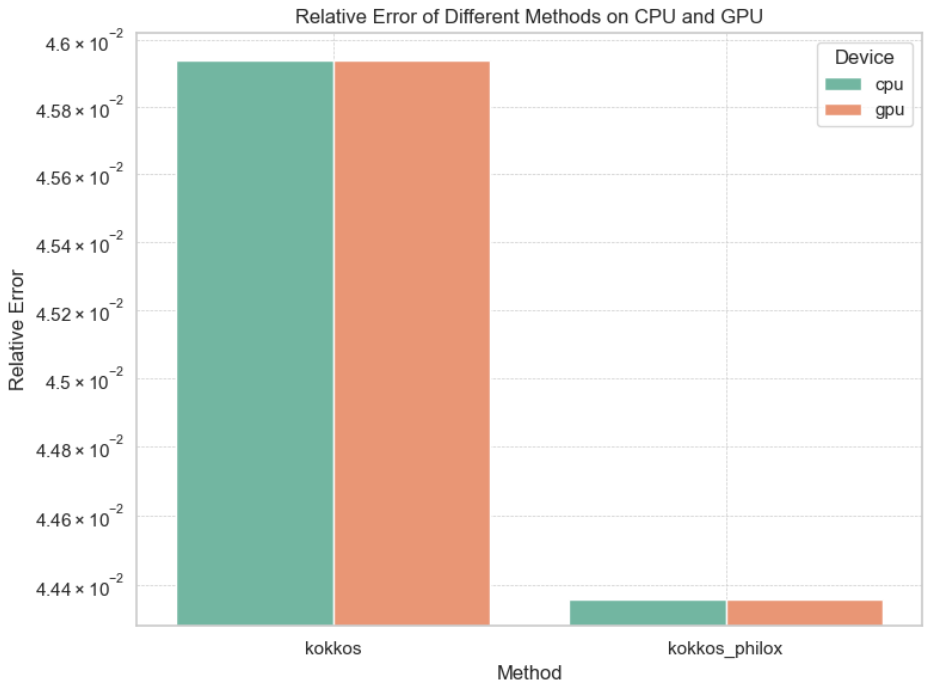


Figure 4.3: Relative error of the sparse sketch methods. Both variants stay below 5% error; `kokkos_philox` is the most accurate overall.

quality random numbers from the Philox generator, which improve the consistency of the sketch, as well as better reproducibility which can help in a multi-threaded context. On the other hand, the dense sketch approach, while still providing substantial speedups, tended to have a bit lower accuracy in our tests. The dense method can be fastest when the sketch size is very small (since a dense S can be applied with efficient BLAS-3 operations), but as the sketch size grows, this method can become bottlenecked by memory bandwidth and the cost of generating a large dense random matrix. In contrast, the sparse methods scale better with increasing sketch size because they perform fewer operations (proportional to the number of nonzeros). Thus, for scenarios where one can tolerate a few percent of error, `kokkos_philox` emerges as a favorable choice, yielding significant speedups with minimal accuracy loss.

Error Breakdown and Profiling

To better understand the sources of approximation error, we performed an error breakdown analysis. There are a few potential contributors to the error observed in the sketched TLS solutions: (1) *Sketching randomness*: Different random draws for the embedding matrix S can cause variation in the solution. We mitigated this by fixing the random seed for repeatability in experiments, but inherent variability remains between a sketched and full

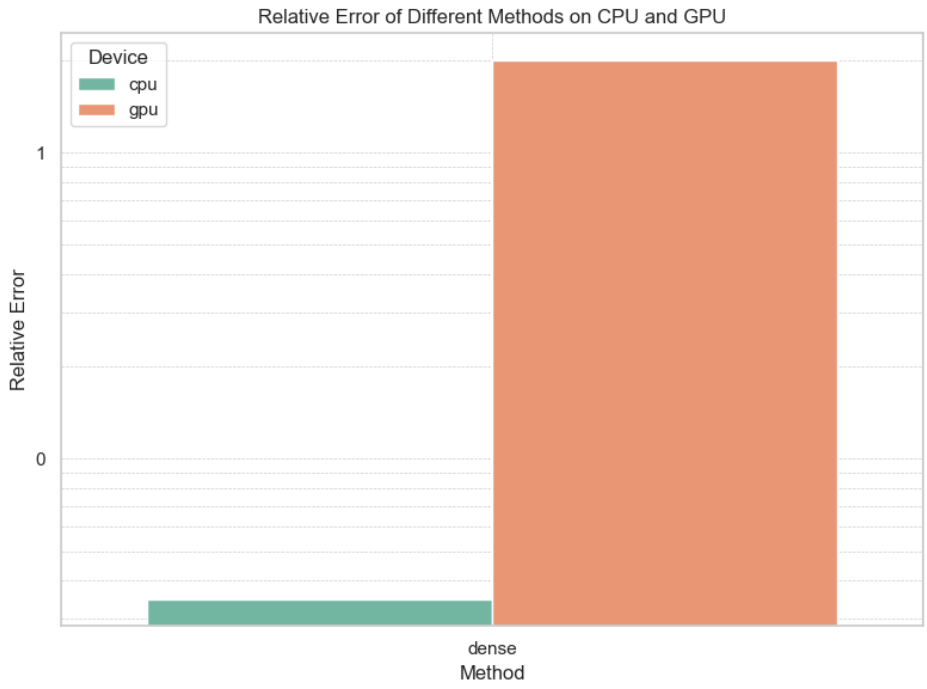


Figure 4.4: Relative error of the dense Gaussian sketch. The GPU solver shows higher error ($\sim$1.2) than the CPU solver ($\sim$0.3), reflecting increased sensitivity to round-off when combining dense embeddings and GPU factorization heuristics.

solution. (2) *Numerical solver differences*: The classical solution uses Eigen (for CPU) or potentially cuSolver (for GPU via LAPACK++), while the sketched solutions on GPU rely on KokkosKernels and on CPU may still use Eigen for the final solve. Minor differences in numerical precision and algorithm (e.g., Eigen's Jacobi SVD vs. a QR-based solver) can lead to slight differences in x_{true} vs. x_{sketch}. (3) *Embedding structure*: The sparse embedding preserves some structure of the original matrix (each row of A only influences 8 combined rows in SA) whereas the dense embedding mixes all rows together. In our experiments, we found that the sparse sketches often yielded slightly more accurate solutions, especially for large m, which suggests that having only 8 nonzeros per column might preserve the row-space structure of A better by avoiding excessive averaging of many rows. The dense sketch, while theoretically preserving the expectation of the data equally, might introduce more cancellation or interference among rows. However, overall the error due to sketching (a few percent) dominated any minor discrepancies from the solver or precision differences. The profiling confirms that the sketching step is the primary source of approximation, and those errors are kept modest by using a reasonably large sketch size (1002) relative to n.

Discussion

The above results validate the effectiveness of using Kokkos for implementing high-performance TLS solvers on diverse hardware. By writing the code once and running it natively on each architecture, we could leverage the GPU for massive speedups without sacrificing the ability to also run on CPU for development or comparison. The sparse randomized sketching approach in particular proved to be a compelling strategy: it reduced computation and data movement significantly, leading to nearly 7× speedup on the GPU, while incurring only a minor loss in accuracy (on the order of 2–4%). The use of Random123's Philox generator further enhanced the approach by providing reproducible and high-quality random projections. This is especially useful in a multi-threaded environment, where standard random generators might produce non-reproducible interleavings of outputs; Philox, being counter-based, ensures that the random numbers are consistent regardless of thread scheduling. In summary, our experimental evaluation demonstrates that a combination of advanced programming models (Kokkos) and randomized algorithms (sketching) can yield TLS solvers that are both fast and accurate on modern HPC systems.

Chapter 5

Future Work

5.1 Scalable TLS Sketch Fusion

One promising direction is to fuse the major phases of the TLS computation into a single scalable pipeline. Rather than treating data generation, sketching, and solving as separate steps, a fused approach would generate data and apply the sketch on-the-fly within a single Kokkos kernel (or a pipeline of tightly coupled kernels). This would eliminate intermediate memory transfers and take better advantage of the memory hierarchy. We anticipate that such fusion, combined with Kokkos's ability to expose parallelism, could further reduce runtime and improve strong-scaling on both CPU and GPU.

5.2 Integration with Streaming Pipelines

Many real-world applications require solving TLS or least squares problems in an online or streaming fashion (for example, continuously updating a model with new sensor data or iterative refinement in time-dependent simulations). Our current batch-oriented approach could be extended to handle streaming data by integrating with frameworks for data streaming. The idea is to update the sketch and solution incrementally as new data arrives, which would enable the TLS solver to be used in time-series analysis or fluid dynamics simulations where the data matrix A and vector B evolve over time. We plan to explore algorithms for updating sketches (such as down-sampling or rotation techniques) that maintain solution quality without restarting the computation from scratch for each update.

5.3 Mixed Precision TLS Solvers

Another avenue for improvement is the use of mixed-precision arithmetic. Modern GPUs offer half-precision (FP16/BF16) operations that are twice as fast and use half the memory of single precision. We intend to investigate performing parts of the TLS computation in lower

precision, for instance using FP16 for the sketching multiplication or even for the SVD solve, while accumulating corrections in single or double precision to preserve accuracy. A careful design will be required to ensure numerical stability, but if successful, mixed precision could significantly reduce the memory footprint and execution time of the TLS solver, especially on GPUs.

5.4 Energy-Aware TLS Optimization

With energy consumption becoming a critical concern in HPC, we profile the power draw of our TLS methods on both CPU and GPU. GPU energy is sampled via NVIDIA's *NVML* API [33], while CPU energy is measured through Intel's *Running Average Power Limit* (RAPL) counters [22]. we can measure energy per operation or per solve. The goal is to identify if the GPU's higher performance also translates to better energy efficiency for TLS, or if the CPU might be more energy-efficient for smaller problems. Based on these findings, an energy-aware scheduler could be developed to choose the optimal execution device (CPU vs GPU) for a given problem size and accuracy requirement. Additionally, we might explore power capping or frequency scaling during different phases (e.g., lower power during data generation, full power during SVD) to optimize the energy-delay product of the solver.

5.5 Compiler-Aware Kernel Fusion

Finally, we plan to leverage advanced compiler features and Kokkos tuning to further optimize kernel execution. Kokkos provides tools for kernel fusion and coupling (such as Kokkos TaskGraph or using explicit fences and team-level parallelism) that could potentially merge the sparse matrix multiplication (SpMM) for sketching and the addition of noise or other transformations into a single kernel launch. By giving the compiler or runtime more insight (through annotations or performance hints) into the sequence of operations, we could reduce the overhead of multiple kernel launches and improve cache reuse between the sketching and solving steps. Investigating these low-level optimizations at the Kokkos and compiler level may yield additional performance gains, especially for GPU execution where kernel launch overhead is non-negligible.

Chapter 6

Conclusion

In this work, we demonstrated that Kokkos-based TLS solvers can achieve both high performance and high accuracy across CPU and GPU platforms. By incorporating randomized sketching techniques, our solver attains significant speedups (up to nearly 7× faster) compared to a classical dense TLS implementation, with only a modest trade-off in accuracy.

Limitations

There are a few limitations to our current approach. First, our implementation of the TLS solver still relies on a full SVD computation on the host for the final solve. We have not yet integrated a GPU-native TLS solve (e.g., a device-only SVD), which means that in GPU runs some overhead is incurred transferring the sketched matrix back to the CPU for the last step. This could be alleviated in the future by using a fully GPU-based SVD solver. Second, our experiments have been limited to synthetically generated data. While this allows us to systematically control problem characteristics (like noise level and matrix conditioning), the performance and accuracy might differ on real-world data sets that have different structures (sparsity, correlation, etc.). We did not test the solver on domain-specific TLS problems such as those in image processing or scientific simulations, which could reveal additional challenges or require further tuning.

Outlook

Looking ahead, we see multiple opportunities to extend and apply this work. One immediate avenue is to test the TLS solver on real application workloads, such as imaging problems (where TLS can help calibrate systems with measurement error) or large-scale scientific computing tasks. This will help validate the practicality of our approach and potentially uncover real-world constraints not evident in synthetic tests. Additionally, integration with high-level randomized linear algebra libraries like RandBLAS could allow us to tap into a broader ecosystem of sketching techniques and perhaps simplify the development of new variants of our solver. In the longer term, the combination of performance portability and

randomness that we explored here can be brought to other related problems in numerical linear algebra, pushing the envelope for what can be achieved in terms of speed and scalability without sacrificing mathematical rigor. Overall, our results encourage further exploration into performance-portable, randomized algorithms for large-scale linear algebra problems.

Bibliography

[1] AMD. *AMD EPYC 7003 Series Processors ("Milan")*. https://www.amd.com/en/products/cpu/amd-epyc-7763. Product brief, accessed 2025-05-14. 2021.

[2] E. Anderson et al. *LAPACK Users' Guide*. 3rd. SIAM, 1999.

[3] D A Beckingsale et al. *RAJA: Portable Performance for Large-Scale Scientific Applications*. Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Sept. 2019. URL: https://www.osti.gov/biblio/1573949.

[4] K. L. Clarkson and D. P. Woodruff. "Low rank approximation and regression in input sparsity time". In: *45th Annual ACM Symposium on Theory of Computing (STOC)*. 2013, pp. 81–90.

[5] J. Demmel et al. "Communication-avoiding QR decomposition". In: *Parallel Processing Letters*. Vol. 21. 1. 2012, pp. 145–170.

[6] J. Demmel et al. "Communication-optimal parallel and sequential QR and LU factorizations". In: *SIAM Journal on Scientific Computing* 34.1 (2013), A206–A239.

[7] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[8] James Demmel. "The Smallest Perturbation of a Submatrix which Lowers the Rank and Constrained Total Least Squares Problems". In: *Siam Journal on Numerical Analysis - SIAM J NUMER ANAL* 24 (Feb. 1987), pp. 199–206. DOI: 10.1137/0724016.

[9] RandBLAS Developers. *RandBLAS: Randomized Numerical Linear Algebra Library*. https://github.com/OptimalDesignLab/RandBLAS. Accessed: 2024-05-05. 2024.

[10] Huaian Diao et al. "Total Least Squares Regression in Input Sparsity Time". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019, pp. 1–12. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/540ae6b0f6ac6e155062f3dd4f0b2b01-Paper.pdf.

[11] Jack Dongarra and Antoine Petitet. "ScaLAPACK Tutorial". In: *Euro-Par '95: Parallel Processing*. Vol. 1041. Lecture Notes in Computer Science. Springer, 1995, pp. 166–176. ISBN: 978-3-540-60902-5. DOI: 10.1007/3-540-60902-4_20.

[12] Jack Dongarra et al. "An extended set of Fortran basic linear algebra subprograms". In: *ACM Transactions on Mathematical Software* 14.1 (1988), pp. 1–17.

[13] P. Drineas, R. Kannan, and M. W. Mahoney. "Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix". In: *SIAM Journal on Computing* 36.1 (2006), pp. 158–183.

[14] P. Drineas, M. W. Mahoney, and S. Muthukrishnan. "Fast approximation of matrix coherence and statistical leverage". In: *Journal of Machine Learning Research*. Vol. 13. 2012, pp. 3475–3506.

[15] P. Drineas, M. W. Mahoney, and S. Muthukrishnan. "Faster least squares approximation". In: *Numerische Mathematik* 117 (2011), pp. 219–249.

[16] H. C. Edwards, C. R. Trott, and D. Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Proceedings of the 2014 ACM/IEEE Conference on High Performance Computing (SC)*. 2014.

[17] Mark Gates et al. "SLATE: design of a modern distributed and accelerated linear algebra library". In: *SC '19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov. 2019), pp. 1–37. DOI: 10.1145/3295500.3356223.

[18] G. H. Golub and C. F. Van Loan. "An analysis of the total least squares problem". In: *SIAM Journal on Numerical Analysis* 17.6 (1980), pp. 883–893.

[19] G. H. Golub and C. F. Van Loan. *Matrix Computations*. 3rd. Johns Hopkins University Press, 1996.

[20] N. Halko, P. G. Martinsson, and J. A. Tropp. "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions". In: *SIAM Review* 53.2 (2011), pp. 217–288.

[21] S. Van Huffel and J. Vandewalle. *The Total Least Squares Problem: Computational Aspects and Analysis*. SIAM, 1991.

[22] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, Section 14.9 (Running Average Power Limit - RAPL)*. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf. Accessed: 2025-05-14. 2016.

[23] Intel Corporation. *Intel oneAPI Programming Guide*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html. Accessed: 2024-05-14. 2020.

[24] William B. Johnson and Joram Lindenstrauss. "Extensions of Lipschitz mappings into a Hilbert space". In: *Contemporary Mathematics* 26 (1984), pp. 189–206.

[25] M. W. Mahoney. "Randomized algorithms for matrices and data". In: *Foundations and Trends in Machine Learning* 3.2 (2011), pp. 123–224.

[26] M. W. Mahoney and P. Drineas. "CUR matrix decompositions for improved data analysis". In: *Proceedings of the National Academy of Sciences* 106.3 (2009), pp. 697–702.

[27] Per-Gunnar Martinsson and Joel Tropp. *Randomized Numerical Linear Algebra: Foundations & Algorithms*. 2021. arXiv: 2002.01387 [math.NA]. URL: https://arxiv.org/abs/2002.01387.

[28] Riley Murray et al. *Randomized Numerical Linear Algebra : A Perspective on the Field With an Eye to Software*. 2023. arXiv: 2302.11474 [math.NA]. URL: https://arxiv.org/abs/2302.11474.

[29] Jelani Nelson and Huy L. Nguyen. "OSNAP: Faster numerical linear algebra algorithms via sparser subspace embeddings". In: *54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2013, pp. 117–126.

[30] NERSC. *Perlmutter System User Guide*. https://docs.nersc.gov/systems/perlmutter/architecture/. Accessed: 2025-05-14. 2021.

[31] NVIDIA Corporation. *cuBLAS Library*. https://developer.nvidia.com/cublas. Version 12.x, https://developer.nvidia.com/cublas. 2024.

[32] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. https://www.nvidia.com/en-us/data-center/a100/. White paper, accessed 2025-05-14. 2020.

[33] NVIDIA Corporation. *NVIDIA Management Library (NVML)*. https://developer.nvidia.com/nvidia-management-library-nvml. Accessed: 2025-05-14. 2024.

[34] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 5.0*. https://www.openmp.org/spec-html/5.0/openmpse65.html. See Section 2.5, on page 171, for heterogeneous offloading support. 2018.

[35] Sivasankaran Rajamanickam et al. "Kokkos Kernels: Performance Portable Sparse/Dense Linear Algebra and Graph Kernels". In: *arXiv preprint* (2021). arXiv: 2103.11991 [cs.MS]. URL: https://arxiv.org/abs/2103.11991.

[36] J. K. Salmon et al. "Parallel random numbers: As easy as 1, 2, 3". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011), pp. 1–12.

[37] T. Sarlós. "Improved approximation algorithms for large matrices via random projections". In: *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2006, pp. 143–152.

[38] Jeremy Walton and Louise Mitchell. *An Introduction to Using the NAG Numerical Library*. https://eis.mdx.ac.uk/research/docs/Introduction-to-using-the-nag-numerical-library.pdf. Accessed: 2025-05-14. 2011.

[39] Qian Wang et al. "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Denver, CO: IEEE, 2013.

[40] R. Clint Whaley and Jack Dongarra. "Automatically Tuned Linear Algebra Software". In: *SuperComputing 1998: High Performance Networking and Computing*. CD-ROM Proceedings. Winner, best paper in the systems category. `http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`. 1998.

[41] R. Clint Whaley and Antoine Petitet. "Minimizing development and maintenance costs in supporting persistently optimized BLAS". In: *Software: Practice and Experience* 35.2 (Feb. 2005). `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`, pp. 101–121.

[42] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS Project". In: *Parallel Computing* 27.1–2 (2001). Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000. `http://www.netlib.org/lapack/lawns/lawn147.ps`, pp. 3–35.

[43] D. P. Woodruff. "Sketching as a Tool for Numerical Linear Algebra". In: *Foundations and Trends in Theoretical Computer Science* 10.1–2 (2014), pp. 1–157.