CoLA Kernels

Jaideep Singh Chawla
New York University, Courant
jc12751@nyu.edu

Rahul Raman
New York University, Courant
rr4549@nyu.edu

Satyanarayana Chillale
New York University, Courant
sc9960@nyu.edu

Abstract

CoLA[5] is a general framework for large-scale linear algebra problems in machine learning. The framework combines a linear operator abstraction with compositional dispatch rules, which automatically constructs memory and runtime efficient numerical algorithms. CoLA supports GPU acceleration with PyTorch and JAX. The framework is written in python, which being a developer friendly language does not provide fine grain access for code optimizations which are obscured from the developer. We developed CUDA kernels for multiple linear algebra operations supported by CoLA. We used concepts such as shared memory, fuse kernels, memory management to optimize the algorithms. Our results shows that the CUDA kernels indeed out perform the python implementation in terms of speedup and memory usage.

1 Introduction

CoLA (Compositional Linear Algebra) is a framework that automates a notorious bottleneck for ML methods: performing large-scale linear algebra (e.g. matrix solves, eigenvalue problems). These ubiquitous operations are at the heart of principal component analysis, Gaussian processes, normalizing flows. It enables efficient manipulation of large linear operators without explicitly materializing them. It provides abstractions for common operations like matrix multiplication, Kronecker products, and matrix decompositions through a unified interface across JAX, PyTorch, and NumPy backends. The framework excels at handling structured matrices and iterative algorithms, making it particularly useful for fields like scientific computing, optimization, and machine learning where working with large matrices is common. CoLA exploits the compositional structure of matrices such as diagonal dominance, sparsity, or a low-rank factorization. Given a structure and a algebraic operations, CoLA finds a lower computational routine for faster computation (e.g., Cholesky for inverse of a matrix, Arnoldi method to find subset of eigenvalues).

While CoLA leverages GPU acceleration for numerical linear algebra routines, there are several performance bottlenecks that stem from primarily four architectural choices that impact GPU utilization:

- Dynamic dispatch creates overhead through Python-level type checking and virtual function calls. Each operation requires method resolution, preventing the compiler from optimizing across operation boundaries.
- Memory management issues arise from creating new allocations for intermediate results in operation chains. This leads to memory fragmentation and excessive GPU memory pressure, particularly in iterative algorithms.
- CPU-GPU synchronization occurs frequently due to Python-level operation composition. Each small operation can trigger a sync point, stalling the GPU pipeline and reducing parallelism.

- The high-level backend abstractions result in many small kernel launches instead of fused operations. What could be a single optimized CUDA kernel becomes multiple smaller kernels with associated launch overhead and reduced GPU utilization. This is especially impactful for complex operation chains like conjugate transpose matrix multiplications.

These bottlenecks compound when dealing with large matrices or composing multiple operations, where the overhead becomes more significant compared to the actual computation time.

We specifically target the fourth kind of bottleneck, where we choose three such algorithms and wrote fused kernels for these routines to demonstrate the advantages of kernel fusion in linear algebra. These algorithms exemplify common patterns in numerical computing where multiple small operations can be combined into efficient monolithic kernels. The Cholesky decomposition showcases triangular matrix operations, Arnoldi iterations demonstrate iterative matrix-vector products with orthogonalization, and Hutchinson estimation illustrates stochastic trace computation. By fusing these operations, we reduce kernel launch overhead, minimize memory transfers, and better utilize GPU resources compared to the CoLA implementation with the PyTorch backend.

Our Algorithms of Interest

- **Cholesky Decomposition**: Factorizes a positive definite matrix A into $LL^\top$ where L is lower triangular. Used for solving linear systems and computing matrix inverses efficiently.
- **Arnoldi Iteration**: Iteratively constructs an orthonormal basis for the Krylov subspace to approximate eigenvalues/eigenvectors of large sparse matrices. Each iteration involves matrix-vector multiplication and Gram-Schmidt orthogonalization.
- **Hutchinson Trace Estimation**: Stochastic algorithm that estimates the trace (sum of diagonal elements) of a matrix using random probe vectors, without explicitly computing diagonal entries. Particularly efficient for large matrices where direct computation is impractical.

All three involve sequences of operations that are typically executed as separate kernels but could benefit from fusion for better GPU utilization. We also considered a CUDA library based implementation of Singular-Value Decomposition for comparing performance with CoLA's PyTorch backend.

2 Literature Survey

2.1 Introduction to Linear Algebra Acceleration Frameworks

Recent years have seen increasing demands for efficient large-scale linear algebra computations in machine learning and scientific computing. Frameworks have emerged to provide high-performance implementations of fundamental operations like matrix multiplication, eigenvalue decomposition, and linear system solving. As matrix sizes grow larger, efficient GPU acceleration becomes crucial for performance. However, as noted in papers like [9], basic GPU acceleration alone is not sufficient—hardware-specific optimization techniques are needed to fully utilize GPU hardware capabilities.

2.2 Current Landscape

2.2.1 Matrix Operation Libraries

Traditional linear algebra libraries like **LAPACK** and **BLAS** have served as the foundation for numerical computations. However, these libraries were designed primarily for CPU architectures and lack native support for modern GPU acceleration patterns. Libraries like **cuBLAS** (optimized vectorized operations) and **MAGMA** provide highly optimized implementations of basic linear algebra operations. **MAGMA** particularly stands out for its hybrid CPU-GPU algorithms that effectively distribute work between host and device [1]. **Ginkgo** offers a modern approach using C++ templates for hardware abstraction [2]. However, these libraries often focus on individual operations rather than operation chains or compositional patterns.

2.2.2 Automatic Optimization Frameworks

Systems like **XLA** demonstrate how automatic optimization can improve performance through operation fusion [6]. XLA implements multiple fusion strategies:

- **Instruction fusion** for simple operation chains
- **Fusion merger** for reducing memory bandwidth requirements
- **Multi-output fusion** for sibling and producer-consumer patterns
- **Horizontal fusion** for similar operations across batches

While XLA can automatically fuse some operations, it struggles with complex linear algebra patterns like triangular solves or iterative methods.

2.2.3 Compositional Approaches

Julia provides an alternative approach through its multiple dispatch system and JIT compilation. Julia's type system allows writing generic linear algebra routines that specialize efficiently for different data types and hardware. Through abstract array interfaces, Julia can support both CPU and GPU computation while maintaining a single implementation. Julia relies on LLVM optimizations and careful manual implementation, while lacking in hardware-specific optimizations[8].

CoLA takes a novel approach by focusing on the compositional structure of linear operators. This allows it to automatically select efficient implementation strategies based on matrix properties like sparsity, diagonal dominance, or low-rank structure [5]. CoLA's focus on structure-aware computation is severely affected by several factors outlined below:

2.3 Key Areas for Optimization in CoLA

2.3.1 Kernel Launch Overhead

A major performance bottleneck in CoLA (and other composable frameworks) comes from launching many small GPU kernels instead of fused operations. This is particularly impactful for iterative algorithms like Arnoldi iteration where each step involves multiple operations that could potentially be fused.

2.3.2 Memory Management

The memory hierarchy in GPUs presents unique challenges for linear algebra operations. Data movement between different memory levels can become a bottleneck, particularly when operations are not properly fused. Managing intermediate results in operation chains poses significant challenges [9]. Key issues include:

- Memory allocation/deallocation overhead
- Data movement between CPU and GPU
- Buffer reuse opportunities
- Memory access patterns affecting coalescing

For computations over large matrices with iterative algorithms, CoLA creates multiple intermediate buffers and performs new allocations for each iteration, severely impacting performance.

2.3.3 Synchronization Overhead

Frequent CPU-GPU synchronization can severely impact performance. This is especially notorious in CoLA where each "op" triggers synchronization through PyTorch, which could potentially be fused into a single CUDA stream.

2.4 Fusion Strategies

Static analysis techniques, as demonstrated in [7], can identify fusion opportunities at compile time. However, they often struggle with dynamic shapes and complex linear algebra patterns that are common in scientific computing. Runtime fusion approaches, like those used in **JAX/XLA**, provide more flexibility but introduce overhead from runtime decision-making. Specialized implementations of linear algebra operations often outperform general-purpose fusion systems as shown in **MAGMA** and **Ginkgo**.

2.5 Structure-Aware GPU Acceleration

While frameworks like **CoLA** provide powerful abstractions that exploit the structure of matrices to decide the theoretically best algorithm, achieving optimal performance requires careful attention to GPU-specific optimizations. Bridging the gap between high-level mathematical abstractions and efficient hardware utilization is the primary focus of this project. We target specialized fusion patterns for common algorithms that enable better exploitation of matrix properties and improved handling of irregular computation patterns.

2.6 Pytorch CUDA implementation

TorchDynamo and TorchInductor are key advancements in PyTorch, addressing challenges in dynamic graph optimization and backend integration with CUDA. TorchDynamo captures PyTorch graphs by transforming Python bytecode, enabling backend compilation and autotuning. Torch Inductor leverages **Triton** for GPU optimization and **C++/OpenMP** for CPU parallelism, ensuring flexibility and portability. Using symbolic math via SymPy, TorchInductor supports dynamic shapes and strides while enabling efficient memory management and guard-based recompilation. These tools enhance PyTorch's performance and adaptability for diverse execution environments, unifying research and production workflows.

3 Proposed Idea

3.1 Arnoldi

The Arnoldi decomposition involves constructing two matrices: a set of Arnoldi vectors and a Hermitian matrix. Although these are inherently two-dimensional, they are stored as one-dimensional arrays in CUDA to optimize memory access. In our implementation, unlike ColA which accesses elements across columns, we process elements row by row. This approach facilitates contiguous data transfer to Streaming Multiprocessors (SMs), enhancing efficiency. In the Arnoldi algorithm iterations progresses from 1 to k-1 for each k ranging from 1 to maximum number of iterations.

To further optimize the algorithm, we employed privatization with shared memory and striding techniques to compute normalization factors and angles between Arnoldi vectors. This involved a sum reduction process where shared memory was allocated equal to the number of threads per block. The reduction started with a stride equal to the block dimension, halving it in each iteration for efficient parallel summation within the block. At the end of each block's execution, the sum is stored at index 0 of the shared memory array and then written to a global memory array at the index corresponding to the block number. These optimizations minimized global memory usage and boosted computational performance, enabling efficient handling of large-scale matrix operations

3.2 Cholesky Inverse

The Cholesky decomposition and its inverse for a positive semi-definite (PSD) matrix in parallel using CUDA. The problem can be divided into three main steps: Cholesky decomposition, in-place normalization of the columns, and the parallelization of the computation of the inverse of the lower triangular matrix. This approach is implemented through a series of three kernels, each performing a specific task within the overall matrix factorization process.

The `decompose_cholesky` kernel computes the Cholesky decomposition of a matrix by first updating the diagonal elements in-place. A **grid-level synchronization** is required since all diagonal elements must be computed before the rest of the blocks can proceed. To achieve this, CUDA **cooperative-groups** are used for synchronization. The normalization of the columns is parallelized, with each entry below the diagonal divided by the corresponding diagonal element. Row updates for all columns are also parallelized, ensuring efficient element-wise computations. Finally, the upper triangular part of the matrix is zeroed out, leaving only the lower triangular elements intact. This approach leverages parallelism to reduce the complexity from $O(n^3)$ to $O(n^2)$.

The `inverse_lower` kernel computes the inverse of a lower triangular matrix L stored in $AInv$ resulting from Cholesky decomposition. The kernel then parallelizes the computation of the lower triangular matrix inverse, with diagonal elements computed sequentially and the off-diagonal elements updated using **atomicAdd** to prevent race conditions. Finally, the diagonal elements of the inverse matrix are computed as the reciprocal of the original matrix's diagonal elements.

The `multiply_lower` kernel computes the product of a lower triangular matrix L and its inverse L^{-1}, storing the result directly in L^{-1}. The diagonal elements are scaled in parallel, while `atomicAdd` is used for updating off-diagonal elements to prevent race conditions. Due to sequential row-wise update for each diagonal element and updating column-wise elements based on the diagonal, the kernel required many grid-level synchronization.

3.3 Hutchinson

The Hutchinson trace estimator implementation achieves efficient parallel computation of matrix diagonal elements through three specialized CUDA kernels, each incorporating distinct optimization strategies. The algorithm employs a stochastic estimation approach, utilizing GPU architecture for parallel processing while minimizing memory transfers and maximizing computational throughput.

The first kernel, responsible for random vector generation, implements an optimized approach to parallel random number generation using cuRAND. The kernel utilizes shared memory to store random number generator states, reducing global memory access latency. Through a grid-stride processing pattern, each thread processes multiple elements, generating either Rademacher or normal distributions. Memory coalescing is achieved by ensuring consecutive threads access consecutive memory locations, while privatization techniques in shared memory minimize bank conflicts and reduce memory access overhead.

The second kernel leverages cuBLAS's optimized matrix-vector multiplication (SGEMM) routines for efficient parallel computation. This kernel processes multiple random vectors simultaneously through batch processing, maximizing GPU occupancy and computational efficiency. The implementation maintains memory coalescing through careful data layout and employs asynchronous execution with CUDA streams to overlap computation and memory transfers. The batch processing approach significantly reduces kernel launch overhead and improves overall throughput, particularly for large matrices.

The final kernel focuses on computing diagonal estimates using Welford's online algorithm for numerical stability. This kernel employs a shared memory reduction pattern for computing running statistics, with careful attention to avoiding warp divergence. Thread block synchronization is minimized through careful work distribution, while atomic operations are employed only when necessary for updating global statistical estimates. Convergence monitoring is implemented through parallel reduction techniques, computing relative errors efficiently across thread blocks while maintaining numerical stability.

3.4 Singular Value Decomposition

Our project focused on Linear Algebra for Dense Linear operators, we explored CUDA libraries such as **cuBLAS**[3] and **cuSOLVER**[4], which are optimized for high-performance GPU computations. The `cuBLAS` library provides basic linear algebra subroutines, enabling efficient matrix and vector operations on NVIDIA GPUs. The `cuSOLVER` library offers advanced routines for solving linear systems, eigenvalue problems, and matrix factorizations, tailored for scientific and engineering applications. For our `SVD decomposition`, we have specifically explored:

Kernel Name	Algorithm
fuse_kernel_2	Cholesky Inverse
fuse_kernel_3	Singular Value Decomposition
fuse_kernel_4	Arnoldi
fuse_kernel_5	Hutchinson

Table 1: Kernel names used for corresponding algorithms in CoLA Kernels.

- `cublasSgemm`: Stands for Single-precision General Matrix Multiplication. Performs a general matrix multiplication (GEMM) operation. This routine provides Fast MVMs and MMs that are needed during SVD and stochastic estimation.

- `cusolverDnSsyevd_bufferSize and cusolverDnSsyevd`: stands for Single-precision Symmetric Eigenvalue Decomposition. Computes the workspace buffer size and solves the problem required for eigenvalue and eigenvector decomposition of symmetric matrices, ensuring efficient memory allocation during computations.

4 Experimental Setup

To test the performance of our custom CUDA kernels, we built our package using PyTorch's `cpp_extension`, allowing the kernels to be integrated into our `CoLA-kernels` Python library. The experiments were run on the CIMS CUDA3 server. We set **cudaSetDevice(1)** to ensure the GPU had full memory bandwidth and to avoid interference from other CUDA programs. This setup allowed us to measure the performance of our kernels accurately. We have profiled our code using **Pytorch Profiler** to profile the CPU usage, CUDA usage, memory usage, and total number of kernel calls.

5 Experiments and Analysis

5.1 Arnoldi

To evaluate the performance of the CUDA kernel implementation for computing eigenvalues and eigenvectors using the Arnoldi decomposition, we conducted benchmarking experiments across various configurations. These experiments compared the CUDA kernel's performance against CoLA CUDA, CoLA CPU, and PyTorch implementations. We tested block sizes of 256, 512, and 1024, identifying 1024 block size as the optimal configuration for this algorithm. The benchmarking involved varying the square matrix sizes and the number of iterations to assess scalability and efficiency under different computational loads and memory usage. The results demonstrate the CUDA kernel's superior performance relative to CoLA CUDA, CoLA CPU, and PyTorch when subjected to identical workloads.

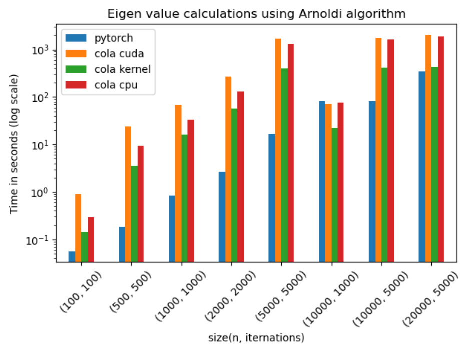

Figure 1: Time vs size(n, iterations)

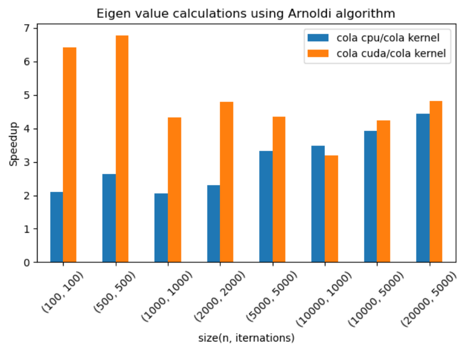

Figure 2: Speedup compared to CoLA kernels

CoLA kernels exhibit notable performance enhancements over both CoLA CUDA and CoLA CPU implementations, achieving up to 5x speedups, as shown in Figures 1 and 2. The largest matrix size we experimented with is (20000, 20000). Beyond this we ran into CUDA out of memory issues. It is important to note that PyTorch computes all the eigenvalues and eigenvectors of a matrix, whereas CoLA is designed to compute only the eigenvalues and eigenvectors corresponding to the number of iterations. We included the times for completeness.

One of the reason for speedup is due the improved Streaming Multiprocessor (SM) utilization, as demonstrated in Table 1. To monitor SM utilization and memory usage, we used the `nvidia-smi dmon` command, a diagnostic tool that provides real-time insights into GPU metrics such as SM utilization, power consumption, and memory usage.

From the experimentation, we analyzed that the improved Streaming Multiprocessor (SM) utilization, Overhead of launch of kernels, coalescing, privatizations are the reason for speedup. One interesting

Implementation/Matrix size	100k	400k
Pytorch	67	80
ColA GPU	20	20
ColA Cuda	60	70

Table 2: SM utilization (%)

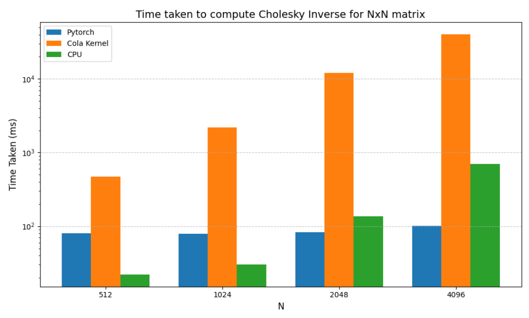

Figure 3: Performance of Cholesky Inverse kernels.

finding is that CoLA CPU execution time is comparable to CoLA CUDA which can be explained by the poor utilization of SMs by CoLA.

5.2 Cholesky Inverse

From the experiments (Figure3), we observe that our Cholesky decomposition implementation is significantly slower than the PyTorch implementation. Using the PyTorch Profiler, we analyzed the performance in detail. Table 3 shows the profiling results for the PyTorch implementation, while Table 7 presents the corresponding results for our kernel.

Kernel Name	**CPU Time**	**CUDA Time**	**CUDA Memory**
Memcpy DtoH (Device → Pinned)	0 ms	2.5 ms	0 MB
`aten::linalg_cholesky`	66.074 ms	791.001 μs	16.00 MB
`aten::linalg_cholesky_ex`	59.751 ms	0 ms	16.00 MB
`aten::cholesky_inverse`	28.812 ms	8.444 ms	32.00 MB
Total (Cholesky)	8 ms	6 ms	128 MB

Table 3: Performance analysis of Pytorch decomposition kernel.

In the PyTorch implementation (Table 3), we observe that part of the Cholesky decomposition computation is executed on the CPU. This approach leverages the fact that the computations are highly interdependent across data, making certain parts more efficient when performed on the CPU.

Conversely, our kernel (Table 7) performs the entire computation on the GPU. While this avoids CPU-GPU data transfers, it incurs a performance penalty due to the use of **cooperative groups and atomicAdd** operations required by our algorithm. The second observation is for the problem size `n=2048`, the memory required by our kernel is `16 MB`, as all operations are performed **in-place**. In comparison, the PyTorch implementation requires `128 MB`, which is 8 times more memory for the same problem size. And finally, our kernel completes the computation using `4 kernel calls`, whereas the PyTorch implementation utilizes `95 kernel calls` to handle the same task. Overall these operations introduce significant overhead, resulting in slower overall execution compared to PyTorch's hybrid CPU-GPU approach.

5.3 Hutchinson

The performance of our kernel is much slower than CoLA, currently.

The analysis of the batch size pattern reveals that smaller batch sizes, such as 16 and 32, are being used, which are lower than typically expected optimal values. This anomaly suggests potential

Kernel Name	CPU Time	CUDA Time	CUDA Memory
`decompose_cholesky(float*, int)`	0 ms	859.062 ms	16.00 MB
`cudaLaunchCooperativeKernel`	0 ms	905.684 μs	0 MB

Table 4: Performance analysis of our Cholesky decomposition kernel.

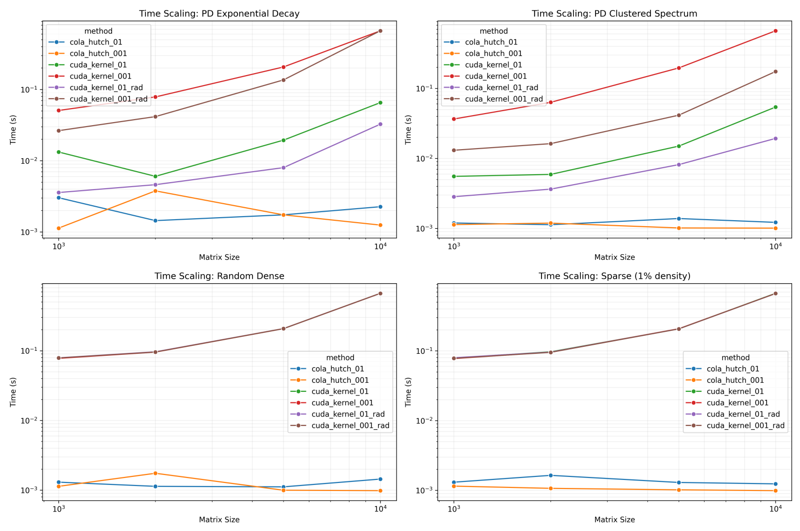

Figure 4: Scaling Performance of CoLA-Kernel in Diagonal Estimation

memory constraints or load balancing issues within the CUDA kernel. To address this, it would be beneficial to experiment with tuning the grid size and block size parameters in the CUDA kernel configuration.

Regarding the tolerance pattern, there is an evident inconsistency in the tolerance values used for different sizes. For instance, a tolerance of 0.001 is used for sizes 1000 and 5000, while a higher tolerance of 0.1 is used for size 2000. This inconsistency points to potential issues with convergence, and the observed high error rates (ranging from 0.77 to 1.77) further indicate that the estimations might not be optimal.

Lastly, examining the maximum iterations parameter, it is observed that all configurations use the minimum number of iterations, set at 100. Coupled with the high error rates, this suggests that the algorithm might be stopping early, potentially before reaching an optimal solution. This highlights the need for reevaluating the stopping criteria to ensure better convergence and more accurate results.

5.4 Singular Value Decomposition

For Singular Value Decomposition (SVD), we utilize NVIDIA's cuBLAS and cuSolverDn libraries. SVD primarily consists of matrix multiplications and the computation of eigenvalues and eigenvectors. For matrix multiplication, we leverage the highly optimized **cublasSgemm** function, while for computing eigenvalues and eigenvectors, we use **cusolverDnSsyevd** (further details can be found in the Proposed Idea section).

In our experiments, we observed a significant performance improvement, with at least a **10x** speedup when comparing our custom kernel with PyTorch's native implementation. From the profiling results obtained from PyTorch, we observed frequent calls to functions such as `aten::transpose`, `aten::clone`, and `aten::copy`, indicating that a significant portion of the computation time is spent on tensor manipulations rather than the core SVD operations themselves. Furthermore, as shown in Table 5, we note that approximately 99% of the time is spent in the `svd_batch_rotate` operation.

In contrast, when using our custom kernel, which is built on top of the most optimized NVIDIA GPU libraries, we observe from Table 6 that the majority of the time is spent in the Symmetric Matrix-Vector Multiplication (SYMV) algorithm, a key step in the cuSolverDn routine. This shift in bottleneck from tensor manipulations in PyTorch to SYMV in our optimized kernel suggests that the use of optimized libraries like cuBLAS and cuSolverDn for matrix operations and eigenvalue computations provides a significant efficiency gain over the PyTorch implementation.

Kernel Name	**Total CUDA Time**	**Avg. CUDA Time**	**No. of Kernel calls**
`svd_column_rotate_batch`	18.146 s	1.366 ms	13286
`svd_row_rotate_batch`	13.981 s	2.105m s	6643

Table 5: Profiler of Pytorch SVD

Kernel Name	**Total CUDA Time**	**Avg. CUDA Time**	**No. of calls**
`cuds_symv_alg6_stage1_upper`	1.541 s	188.064 μs	8192

Table 6: Profiler of Pytorch SVD

Kernel Name	**Total CUDA Time**	**Avg. CUDA Time**	**No. of Kernel calls**
`svd_column_rotate_batch`	18.146 s	1.366 ms	13286
`svd_row_rotate_batch`	13.981 s	2.105m s	6643

Table 7: Performance analysis of our Cholesky decomposition kernel.

6 Conclusions

1. PyTorch's high-level abstractions introduce overheads for flexibility and ease of use, which we eliminated by writing custom GPU code. This allowed us to fine-tune memory usage, parallelism, and grid configurations, resulting in significant speedup and improved efficiency.
2. Profiling CoLA revealed significant overhead from numerous kernel launches. To address this, we optimized the Cholesky inverse by consolidating it into a single kernel. However, the iterative nature of the algorithm limited full SM utilization. Hence for other algorithms, we developed separate kernels, merging some, which resulted in notable speedup.
3. Better memory bandwidth utilization, aggressively using shared memory, and ensuring coalesced memory access patterns are only scratching the surface of CUDA optimizations.

References

[1] Ahmad Abdelfattah, Natalie Beams, Robert Carson, Pieter Ghysels, Tzanio Kolev, Thomas Stitt, Arturo Vargas, Stanimire Tomov, and Jack Dongarra. Magma: Enabling exascale performance with accelerated blas and lapack for diverse gpu architectures. *The International Journal of High Performance Computing Applications*, Jun 2024.

[2] Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software*, 5(52):2260, Aug 2020.

[3] NVIDIA Corporation. cublas library user guide, 2024. Accessed: 2024-11-20.

[4] NVIDIA Corporation. cublas library user guide, 2024. Accessed: 2024-11-20.

[5] Andres Potapczynski, Marc Finzi, Geoff Pleiss, and Andrew Gordon Wilson. Cola: Exploiting compositional structure for automatic and efficient numerical linear algebra, 2023.

[6] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[7] Daniel Snider and Ruofan Liang. Operator fusion in xla: Analysis and evaluation, 2023.

[8] Sophie Xuan, Evelyne Ringoot, Rabab Alomairy, Felipe Tome, Julian Samaroo, and Alan Edelman. Synthesizing numerical linear algebra using julia.

[9] Shaoshuai Zhang, Vivek Karihaloo, and Panruo Wu. Basic linear algebra operations on tensorcore gpu. *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, page 44–52, Nov 2020.