Benchmarking Linear Algebra: Python Vs. NumPy Vs. PyTorch Vs. C

Varak Tanashian

Date: August 26, 2025

Contents

1 Introduction

1.1 Overview

In this project, I compare different tools for conducting common linear algebra operations. The focus is on evaluating their performance in terms of speed, memory usage, and ease of use. This is an important area of research as linear algebra is the backbone of many fundamental algorithms in machine learning and scientific computing. Understanding which tools to use (and why) will give the user both confidence and a deeper understanding of which tools to use.

1.2 Motivation

Understanding performance differences across platforms is important for anyone who is working with linear algebra in their applications, whether in machine learning, scientific computing, or data analysis. By identifying the strengths and weaknesses of each tool, users can make informed decisions about which option is best suited to their needs. Users typically have two options: on the one hand, they can implement a hyper-efficient and specific algorithm for the math they need to do from scratch. On the other hand, they can make use pre-built and optimized libraries that have already implemented these functionalities.

The trade-off lies in resource and time efficiency: on the one hand, it takes a lot of effort to make a computing engine in C, but on the other hand, commonly used libraries like NumPy and PyTorch have a lot of overhead and run as an interpreted language, which makes them significantly slower in real-life.

The choice of which tool to use is going to be up to the user. This project specifically focuses on which implementation is most efficient when only taking into consideration compute times and ignoring overhead runtimes.

1.3 Objectives

The main objectives are to objectively compare linear algebra performance across implementations in Python, NumPy, PyTorch, and C. In the concluding remarks, we will discuss the implications of the results and outline the key suggestions for users when choosing a tool for their specific needs.

2 Background and Related Work

2.1 Historical Context

Linear algebra has underpinned scientific computing for decades, with standardized kernels such as BLAS and LAPACK shaping expectations for performance and numerical stability. High-level ecosystems like Python made these capabilities accessible through libraries that delegate heavy computations to optimized native code. NumPy established a consistent array model with vectorized primitives that map cleanly onto BLAS backends. PyTorch expanded the model with automatic differentiation and optional GPU acceleration while retaining efficient tensor kernels. This project situates itself in that lineage by comparing practical throughput across plain Python, NumPy, PyTorch, and a purpose-built C baseline for common operations.

2.2 Related Studies

Prior benchmarks often assess individual operations or focus on single frameworks under fixed hardware and library configurations. Studies that compare compiled and interpreted stacks typically emphasize the cost of dispatch, data layout, and memory bandwidth rather than pure arithmetic. GPU-focused work highlights the trade-off between kernel launch overhead and arithmetic intensity, where small problems favor CPUs and larger problems favor GPUs. End-to-end ML benchmarks are informative but can obscure microkernel behavior due to preprocessing and I/O effects. This work narrows scope to core tensor arithmetic so differences reflect implementation choices rather than pipeline factors.

2.3 Research Gap

Existing comparisons rarely unify the parsing of test cases, batched execution, and standardized JSON reporting across heterogeneous implementations. They also tend to under-document the mechanics that affect timing repeatability, such as warmups, log-scale visualization, and skipping pathological sizes for naive kernels. Moreover, practical users need guidance that maps problem size and operation type to a suitable tool without overfitting to one machine. This report addresses those gaps by using a single bench-file format and a common runner that enforces comparable timing semantics. The result is a portable harness that favors clarity and reproducibility over benchmark maximalism.

3 System Design and Architecture

3.1 Architecture Overview

The system centers on a Python orchestrator that compiles and invokes all implementations in a single batch, then aggregates results into a JSON report. A shared bench-file format specifies operations, shapes, and data using simple tags so each implementation can parse identically. Each implementation prints one JSON object per test with shapes, run counts, and timing statistics, enabling uniform downstream analysis. A visualization script then groups results by operation and shape, generating bar charts, box plots, and tables on a log scale. This separation of orchestration, execution, and reporting keeps the code modular and easy to extend.

3.2 Core Components

The orchestrator compiles the C binary if needed, discovers tests, filters oversized cases, and runs each implementation with consistent environment variables. The C, NumPy, PyTorch, and Vanilla Python backends each provide a bench mode that parses the bench file and emits structured timing data. The bench parser supports scalar-fill after the DATA tag, allowing concise definitions of large constant tensors. Results are written as newline-delimited JSON records and later merged into a single report for plotting. The plotting module creates per-case visualizations and an aggregate table with consistent labeling and log-scale axes.

3.3 Data Flow

Benchmark blocks are read from a text file and split into independent cases by the runner. Each backend parses the same tokens into in-memory tensors, executes warmups, and then times a fixed number of runs per case. The backends emit JSON lines to stdout, which the runner collects and writes to a consolidated report file. The visualization tool loads this report, groups rows by operation and shape, and renders charts plus tabular summaries to the results directory. This linear flow avoids intermediate mutable state and makes failures localized and easy to diagnose.

4 Implementation Details

4.1 Code Structure

The codebase is organized by implementation, with separate entry points for C, NumPy, PyTorch, and Vanilla Python. The runner script provides build orchestration, environment detection for Python interpreters, and process management with timeouts. Bench parsing logic is duplicated across backends to ensure consistent semantics for shapes, data, and scalar parameters. All backends share the same operation set: element-wise add, matrix multiply, batched matrix multiply, scalar multiply, and dot product. Each backend returns a consistent JSON schema so the plotting stage can remain implementation-agnostic.

4.2 Libraries Used

The C implementation uses the C99 standard library and a simple timer for portability, prioritizing minimal dependencies. NumPy and PyTorch rely on their native tensor engines, which can leverage multi-threaded BLAS and, for PyTorch, optional CUDA acceleration. The Vanilla Python version uses nested lists and straightforward loops to make algorithmic costs explicit. Matplotlib and NumPy are used in the visualization pipeline to render log-scale charts and compute basic statistics. The entire workflow is driven by standard Python tooling for portability across platforms.

4.3 Testing

Small illustrative examples in each backend print intermediate results to validate shape handling and arithmetic behavior. The bench mode focuses on timing and does not perform cross-implementation numerical comparisons during batch runs. Correctness is encouraged by shared input definitions and by using simple, deterministic test values in the bench file. Users can extend the harness with explicit correctness checks when needed, trading off additional overhead for verification. In practice, the primary goal here is consistent timing under a fixed input specification.

5 Benchmarking Methodology

5.1 Experimental Setup

The runner enforces consistent timing parameters via environment variables for runs and warmups, and it executes all tests in one process per implementation. By default, it compiles the C binary with O2 and enables skipping of computationally expensive or oversized cases for C and Vanilla Python. The Python interpreter can be selected from a local virtual environment to stabilize library versions. GPU timing in PyTorch includes synchronization to ensure that recorded durations reflect completed work. Hardware specifics are not recorded automatically, so results should be interpreted relative to the host system.

5.2 Measurement Techniques

All Python backends use `time.perf_counter()` and the C backend uses `clock()` to measure elapsed time. Each case runs a configurable number of warmups to amortize initialization effects before recording timed iterations. Per-case statistics include median, mean, population standard deviation, and the raw list of trial times for box plots. Visualization uses log-scale axes with decade ticks to compare a wide range of magnitudes cleanly. Only execution time is measured; memory and utilization metrics are out of scope for this version.

5.3 Data Analysis

After execution, the runner writes a single JSON report consolidating all implementations and test cases. The analysis script groups rows by operation and the pair of operand shapes to enable side-by-side comparisons. Bar charts display medians across implementations, while box plots show variation when multiple timings are available. An aggregate table is generated in both CSV and Markdown formats, substituting "skipped" when a backend omits a case. Outputs are written under a versioned results directory for easy inspection and sharing.

6 Results and Analysis

6.1 Performance Metrics

The primary metric is median execution time per case because it is robust to occasional outliers. Mean and standard deviation provide context for variability across runs and can flag unstable scenarios. Box plots visualize the distribution of raw timings on a log scale to surface skew and spread. Results are reported per operation and shape, allowing fair comparisons across backends on identical inputs. Skipped entries are retained in tables so readers can see which sizes are intentionally filtered by policy.

6.2 Comparative Analysis

Comparisons are made within each operation class to keep semantics consistent across implementations. Element-wise addition and dot products are typically latency-bound and favor low-overhead paths on small inputs. Matrix multiplications benefit from optimized kernels and threading in NumPy and PyTorch, particularly at larger sizes. Batched matrix multiplication amplifies these trends by increasing arithmetic intensity relative to dispatch overhead. The C implementation provides a transparent baseline for algorithmic cost, while Python list code clarifies the impact of interpreter overhead.

6.3 Key Findings

On small problems, the lowest-overhead path often wins, making the C and Vanilla Python implementations competitive for simple operations. As problem sizes grow, vectorized libraries close the gap and then exceed naive baselines due to tuned kernels and better cache behavior. For large matrix multiplications, optimized backends routinely outperform unoptimized loops by orders of magnitude. PyTorch can benefit from GPU acceleration when available, but launch overhead dominates at tiny sizes. These patterns are consistent with the aggregate tables and per-case plots generated by the harness.

7 Discussion

7.1 Interpretation of Results

The results emphasize matching the tool to the workload by considering operation type, tensor shape, and available hardware. Small, frequent operations may benefit from lower-level or lower-overhead paths, while larger, compute-heavy tasks justify library dispatch costs. Warmups and batched execution materially improve timing stability and should be retained for reproducibility. Skipping extreme cases in C and Vanilla Python keeps total runtime reasonable without obscuring trends. Practitioners should calibrate thresholds and runs to their own machines for decision-grade comparisons.

7.2 Limitations

The current harness does not collect memory usage, hardware counters, or threading details, which limits multi-factor analysis. Library backends and BLAS configurations can differ across systems, introducing variability not controlled by the runner. The C timer's granularity and OS scheduling can affect microsecond-scale measurements, especially on very fast kernels. Numerical correctness is demonstrated on small examples but not re-validated across implementations during batch mode. GPU availability and configuration are environment-dependent and may change the relative standings.

7.3 Implications

These constraints suggest using the harness as a comparative guide rather than an absolute performance oracle. Teams should supplement timings with correctness checks and environment capture if decisions hinge on narrow margins. For production, the decisive factors include maintainability, deployment constraints, and library availability, not just raw speed. Iterative benchmarking on representative workloads will give more actionable results than synthetic extremes. The present framework provides a solid baseline that can be extended where additional fidelity is required.

8 Conclusion and Future Work

8.1 Summary

This work presents a unified, practical framework for benchmarking core tensor operations across C, NumPy, PyTorch, and Vanilla Python. A shared bench format, consistent timing semantics, and standardized JSON reporting enable clear comparisons. Visualizations on log scales and aggregate tables make large performance differences readable at a glance. The results align with expectations: low-overhead paths excel on small tasks, while optimized libraries dominate large matrix operations. The system favors reproducibility and clarity so users can make grounded, context-specific choices.

8.2 Future Directions

Future work will add automated correctness checks, hardware metadata capture, and optional memory profiling to enrich context. Extending the operation set to include convolutions, reductions, and sparse primitives would broaden applicability. Parallel and distributed variants could measure scaling properties beyond single-process runs. A CUDA-backed C path or bindings to vendor libraries would enable deeper GPU comparisons. Finally, a configuration file for thresholds and environment settings would make the harness easier to tune and share.