Tensor Computation Benchmarks: `Python` Vs. `NumPy` Vs. `PyTorch` Vs. `C`

Varak Tanashian

Date: August 27, 2025

Contents

1 Introduction

1.1 Overview

In this project, I compare different tools for conducting common tensor computation 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 tensor computations are 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. Even the most simple developments and innovations can lead to great leaps in performance as the effects of increased efficiency compound.

1.2 Motivation

Understanding performance differences across platforms is important for anyone who is working with high-throughput numerical workloads 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.

Typically, they have two options: they can either implement highly specialized, efficient algorithms tailored to their specific problems, or leverage pre-built, optimized libraries that provide these functionalities out of the box.

The trade-off centres on resource and time efficiency: building a computing engine in `C` demands significant effort, but widely used libraries like `NumPy` and `PyTorch` introduce considerable overhead and, being tied to interpreted languages, often run noticeably slower in practice. For any given large-scale application, this is a decision point that can affect not only computation speed but also the rate of development.

The choice of tool depends on the user's priorities. This project specifically focuses on raw compute-time efficiency: setting aside overhead and runtime environment considerations. As applications become more resource-intensive and advanced, the need for efficient computation becomes even more critical. The intention of this report is to provide a clear framework for when to make a tool from scratch and when to use a pre-built library.

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. This way, developers could focus on high-level logic and technological innovations rather than the calculations that happen in the background. `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 favours clarity and reproducibility over benchmark maximalism.

3 System Design and Architecture

3.1 Architecture Overview

The system centres on a `Python` orchestrator that compiles and invokes all implementations in a single batch, then writes results into a `JSON` report. A shared bench-file format specifies operations,

shapes, and data using simple tags so each implementation parses identically. For each implementation, a JSON object per test with shapes, run counts, and timing statistics is produced for later analysis. A visualisation script 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. Future tests can expand this report by adding modules, creating more tests, and refining existing ones.

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. The plotting module creates visualizations for all cases and aggregates a table with all of the results.

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 remains implementation-agnostic. Part of the motivation behind the ”vanilla” implementations is to show that even unoptimized hand-written custom engines can be competitive for small and medium problems, and sometimes faster.

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 behaviour. 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 the 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 favour 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.

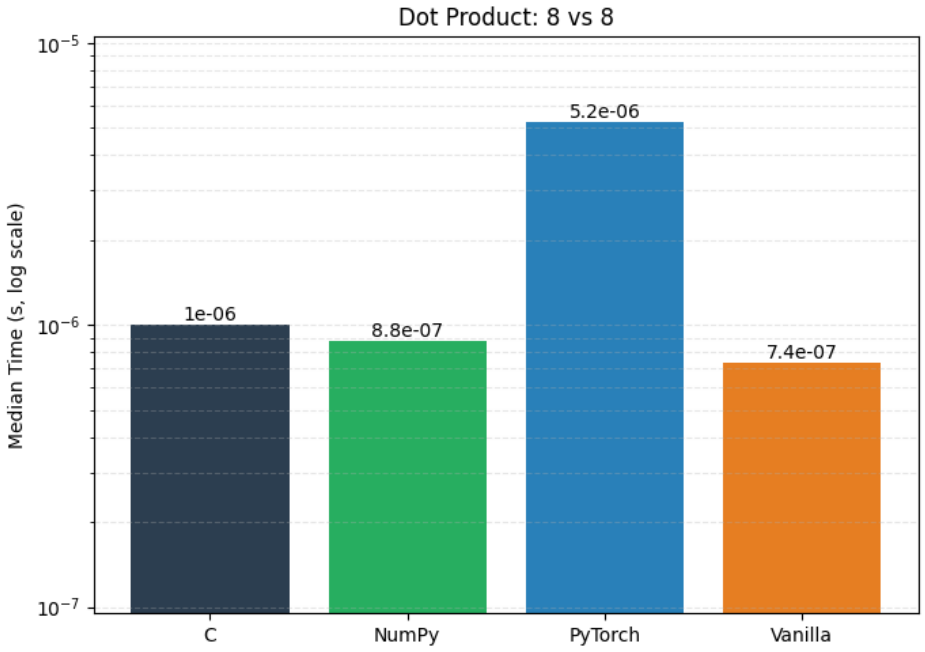

Figure 1: Results of a simple dot product operation.

As we approach medium-sized problems of low complexity, the vanilla `Python` implementation quickly starts to lag behind, while the `C` implementation remains ahead of the pack.

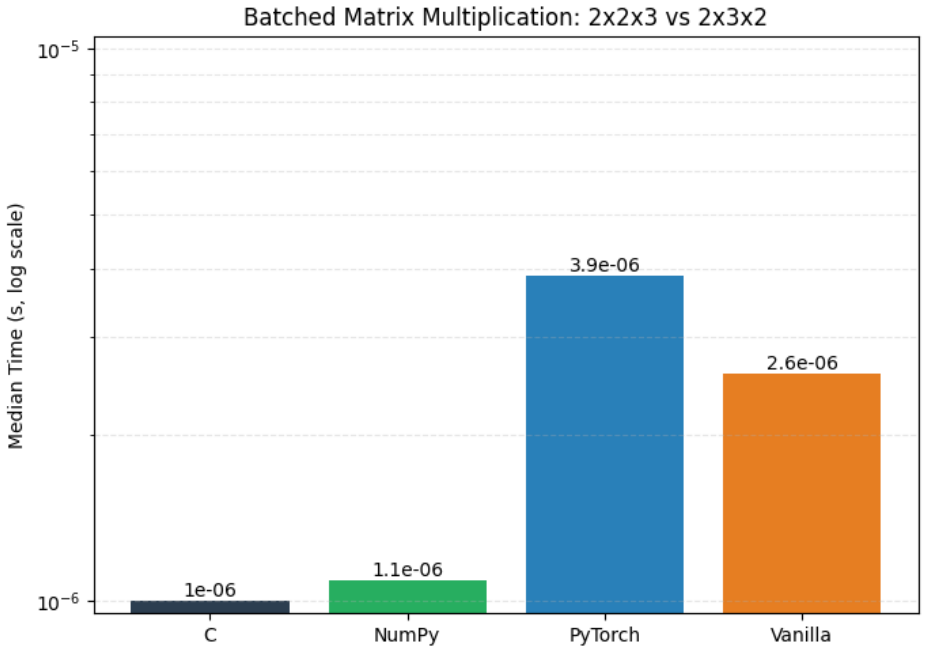

Figure 2: Results of 3D tensor multiplication.

In these early tests, `PyTorch` lags behind significantly, since it is not optimized for small workloads nor specifically designed for microbenchmarks. As problem sizes increase, the overhead of `PyTorch` becomes less significant and it begins to catch up with `NumPy`.

As problem sizes grow further, vectorized libraries close the gap and exceed naive baselines due to tuned kernels and better cache behaviour. 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.

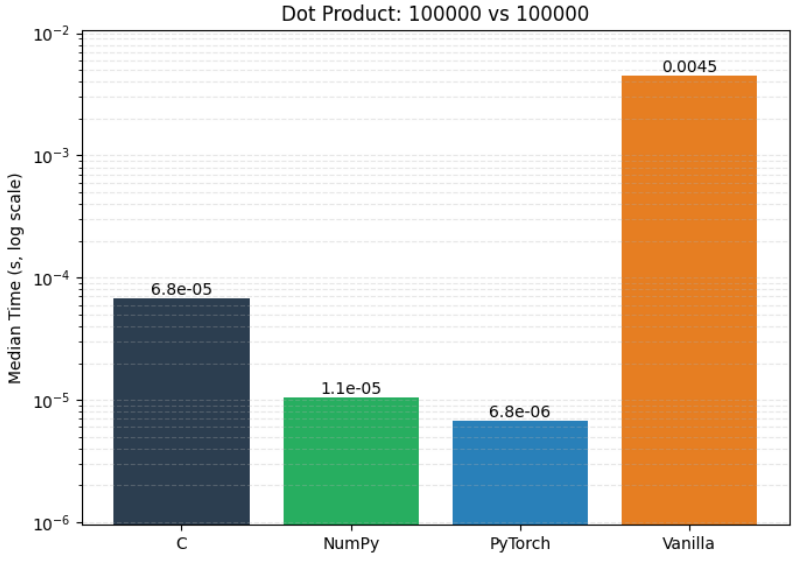

Figure 3: Dot product of 100,000-dimensional vectors.

Eventually, the operations become so large that the optimizations made by `NumPy` and `PyTorch` start to pay off, and they begin to outperform the `C` implementation as well.

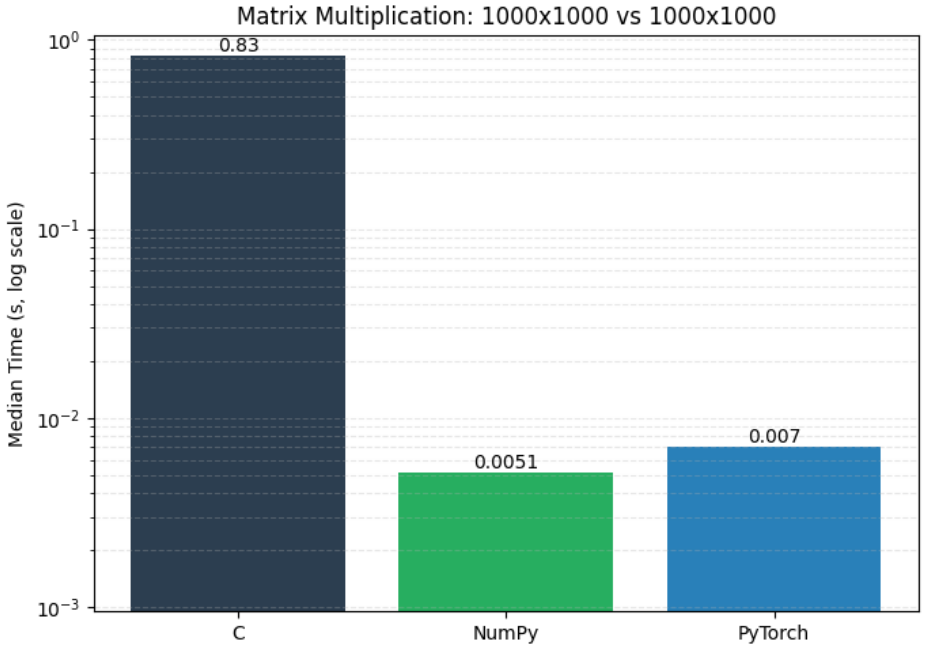

Figure 4: Matrix Multiplication of 1000x1000 matrices.

Soon enough, the difference becomes so drastic that it is not even worth measuring the `C` and Vanilla `Python` implementations, as they are so far behind that they are not even competitive. `NumPy` and `PyTorch` finish in milliseconds, while `C` and Vanilla `Python` can take minutes.

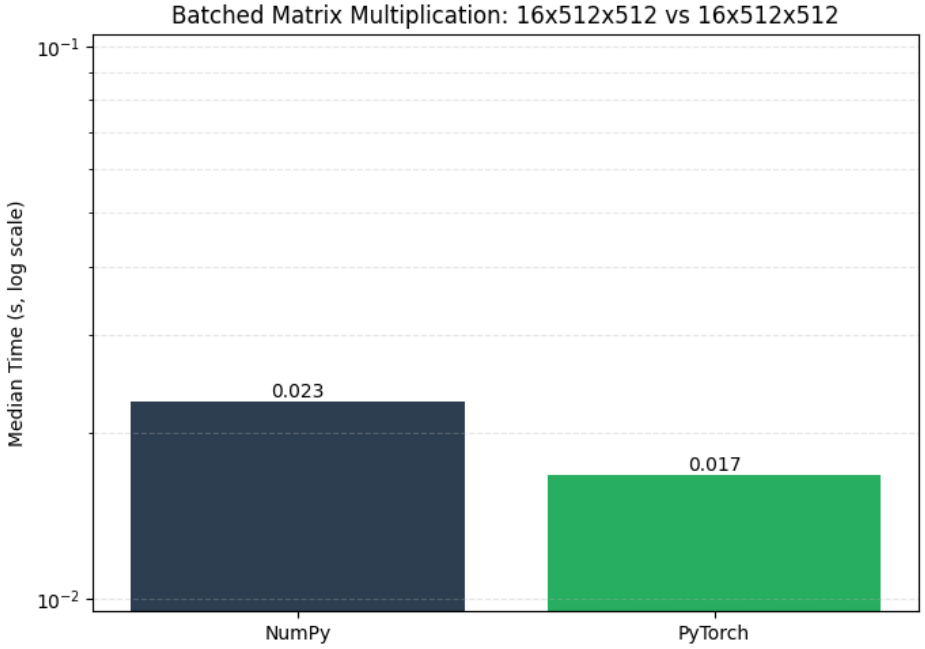

Figure 5: Batched matrix multiplication for tensors of shape 16x512x512.

6.4 Aggregate Performance Results

Table 1 aggregates the median execution times for each operation type and tensor shape combination across all test cases. Timing values are reported in seconds. Entries marked as "skipped" indicate cases where the implementation was intentionally excluded from testing due to computational infeasibility for the Vanilla implementations on large problems.

Op	Shape A	Shape B	C	NumPy	PyTorch	Vanilla
ADD	3x3	3x3	1e-06	3.90999e-07	2.284e-06	1.723e-06
MM	3x3	3x3	1e-06	1.653e-06	1.553e-06	2.169e-06
SCAL	3x3	-	1e-06	6.72e-07	3.006e-06	9.61998e-07
DOT	3	3	1e-06	8.91996e-07	3.356e-06	5.50994e-07
BMM	2x2x3	2x3x2	1e-06	1.092e-06	3.8825e-06	2.585e-06
ADD	2x3x4	2x3x4	1e-06	3.81006e-07	1.392e-06	3.9575e-06
MM	64x64	64x64	0.00014	7.0735e-06	6.397e-06	0.00981652
SCAL	100	-	1e-06	6.71498e-07	3.016e-06	6.502e-06
DOT	100	100	1e-06	9.11998e-07	3.336e-06	4.0675e-06
BMM	4x16x16	4x16x16	9e-06	2.285e-06	6.1115e-06	0.000667536
ADD	8x8	8x8	1e-06	3.80998e-07	1.373e-06	7.85e-06
MM	8x8	8x8	1e-06	1.002e-06	2.48e-06	2.2402e-05
SCAL	8x8	-	1e-06	6.81001e-07	4.659e-06	4.889e-06
DOT	8	8	1e-06	8.81999e-07	5.239e-06	7.35999e-07
BMM	2x8x8	2x8x8	1e-06	1.202e-06	7.9855e-06	4.7504e-05
ADD	1000x1000	1000x1000	0.0007365	0.000205315	7.9754e-05	skipped
MM	1000x1000	1000x1000	0.826766	0.00514185	0.00701244	skipped
DOT	100000	100000	6.8e-05	1.05195e-05	6.763e-06	0.00450608
SCAL	1000x1000	-	0.000465	0.000198016	0.000691967	skipped
ADD	100x100x100	100x100x100	0.000719	0.000243632	0.000141315	skipped
BMM	16x512x512	16x512x512	skipped	0.0227862	0.0167363	skipped

Table 1: Aggregate performance results across all implementations and test cases. Times are reported in seconds using median execution time. Skipped entries indicate computationally infeasible cases.
Acronyms*: **ADD = Addition, **MM** = Matrix Multiplication,
SCAL = Scalar Multiply, **DOT** = Dot Product, **BMM** = Batched Matrix Multiplication.

7 Discussion

7.1 Interpretation of Results

From these results, we understand the importance of using the correct tool for the correct workload. Small, frequent operations benefit from lower-level or lower-overhead paths, especially pre-compiled ones like those in `C`. As the computations get larger, they start to 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. Given the magnitude of the differences and the sample sizes used for running these tests, it would be reasonable to expect similar trends in other environments.

7.2 Limitations

The current setup does not collect memory usage, hardware counters, or threading details, which limits multi-factor analysis. Library backends and **BLAS** configurations may differ across systems, which would introduce variability not controlled by the runner. The `C` timer's granularity and OS scheduling can affect microsecond-scale measurements, especially on very fast kernels. GPU availability and configuration are environment-dependent and may change the relative standings.

7.3 Implications

Given the constraints, one should take these results as indicative of clear trends rather than exact measures of performance. Small, simple, and highly repeated tasks benefit from custom (even unoptimized) pre-compiled implementations, while heavier, deeper computations require sophisticated approaches like those found in `NumPy` or `PyTorch`. If decisions hinge on narrow margins, teams should supplement timings with correctness checks and environment capture to make a decision. For production, the decisive factors include maintainability, deployment constraints, and library availability, not just raw speed.

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

Potential future improvements could add correctness checks, hardware metadata capture, and optional memory profiling to enrich context. Extending the operation set to include convolutions, reductions, inverses, and sparse primitives would also be a good idea. 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.