International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Benchmarking Python–Rust Integration for High-Performance Computing Tasks

Srikant Singh[1], Ravi Raj[2]

[1]Software Engineer Independent Researcher, Hajipur, Bihar, India PIN -844504
[2]University institute of Computing Chandigarh University, Mohali, Punjab, India PIN -140301

Abstract
High-performance computing (HPC) requirements have dramatically increased in recent years for a wide range of tasks, including data analytics, machine learning and system optimization. Although Python is a favored language in science and analysis, its dynamic features can cause performance bottlenecks. On the other hand, Rust is a new systems language that guarantees memory safety while offering near-native execution performance, and as such is a prime contender for high-performance applications. In this paper we conduct a benchmarking study of Python-Rust interoperability: We benchmark pure Python versions against Python code with embedded Rust routines (using the PyO3 package) for various tasks like arithmetic calculations, string operations, list operations, file operations and conditional statements. Our benchmarking experiments reveal that the Rust-enhanced versions systematically execute faster than pure Python code - up to three times faster in some cases. We also analyse the impact of effective language integration and reduced run-time on more sustainable software engineering practices: by reducing the overall number of CPU cycles, memory use and energy consumption, hybrid approaches enable more energy-efficient HPC applications. Finally, we discuss when and how the Python–Rust co-development can be used to create high-performance and energy-efficient solutions in scientific computing.

Keywords: Python–Rust Integration, High-Performance Computing (HPC), PyO3, Performance Benchmarking, Sustainable Software Engineering

INTRODUCTION

In recent years, an increased need for high-performance computing has emerged as data analytics, machine learning and optimization problems have grown in size [5]. With these demands increasing, the performance of the software used to solve problems has become a critical factor.

Scientific and analytical computing frequently use Python because of its ease of use, rich suite of libraries and strong community backing [1]. But the interpreted nature of Python code and its dynamic typing system can lead to inefficiencies in compute-intensive applications [1]. An additional limitation is imposed by the Global Interpreter Lock (GIL) in the CPython implementation, which allows only a single thread to execute Python bytecode at a time, thus limiting full parallelism on multi-core systems [8].

The GIL, though simplifying memory management and avoiding certain concurrency problems, represents a significant bottleneck for high-performance and multi-threaded applications [5]. To overcome these limitations, approaches that mix Python's ease of development with the efficiency of

compiled languages have become a popular approach [2]. These approaches retain Python as the main programming interface while offloading compute-intensive parts of the program to a faster compiled language, without the need to re-implement an application from scratch.

Rust has proven to be an attractive choice for this purpose. Its ownership system and type system guarantee memory safety and completely rule out data races at compile time, while avoiding the cost of a garbage collector [3]. This, along with zero-cost abstractions and concurrency efficiency, makes Rust ideal for performance-critical parts of hybrid systems [3][5]. In particular, the PyO3 library offers a Rust binding to the Python interpreter, and allows for Rust code to be compiled into native Python extension modules that are callable from Python [6].

This enables targeted replacement of bottlenecks in a Python code with Rust code, creating a hybrid system that combines Python's development friendliness with Rust's efficiency [5]. In this paper, we compare implementations of various tasks in pure Python against Python-Rust hybrid implementations using PyO3 in five broad groups of useful tasks: operations on numbers, strings, lists, file I/O and boolean logic operations. Our aim is to determine the performance gain of using Rust for each type of task and to assess where it makes the most difference. This work not only examines the execution time, but also the broader impact of these improvements. Shorter execution times and reduced memory requirements in hybrid variants are related to fewer CPU cycles required to execute a task, which in turn affects the energy required for execution [4].

Given the growing importance of energy efficiency in software engineering, incorporating Rust into a Python development framework to enhance the performance of critical tasks can be viewed not only as a performance optimization, but also as a step towards more energy-efficient computing [4].

The rest of this paper is structured as follows: Section II provides an overview of the related work; Section III details the experimental setup and benchmarking framework; Section IV discusses the results; and finally, the concluding and future work are presented in Section, V and VI.

BACKGROUND AND RELATED WORK

Python is a high-level, general-purpose language that is commonly applied in scientific computing and data analysis, due to its ease of use, extensive package availability and strong community [1]. However, its high-level, interpreted and dynamically-typed nature can cause performance bottlenecks in computationally-intensive applications [1].

One such bottleneck is the Global Interpreter Lock (GIL) in the CPython interpreter, which is a mutual exclusion lock that only permits a single thread to run Python bytecode at a time [8].

While the GIL simplifies memory management, and avoids potential concurrency bugs, it also prevents parallel processing on multi-core processors, and can become a significant bottleneck when multi-threaded programs access the CPU intensively [5][8]. As a result, hybrid programming techniques have developed, which allow compiled language functions to be embedded into a Python program to allow performance-critical code to run outside the Python interpreter [2].

Rust is a Systems programming language that offers the performance of native code with compile-time memory safety. Its ownership and type system ensure memory safety without a garbage collector, thus eliminating common programming bugs such as data races and dangling pointers [3]. This, together with the zero-cost abstractions and support for concurrency, make Rust an excellent choice for performance-critical applications that need to be efficient and safe [3]. The Rust standard library supports parallelism, allowing it to safely and efficiently leverage multi-core processors [3].

PyO3 is a library that provides access to the Python interpreter in rust and also allows Rust code to be built as Python extension modules that can be imported into Python [6]. This allows a hybrid language approach to be adopted in which Python can be used for the management of the code and performance-critical parts of the code can be swapped out for Rust code, without the need to rewrite the Python code [6].

Minin [5] investigated the use of Rust concurrency in Python by using a Python-callable Rust library and demonstrated that Rust-based solutions were faster than Python for both single- and multi-threaded CPU-intensive scenarios.

This research primarily focused on the concurrency-based performance gains by removing the GIL. Beierlieb et al.

[11] benchmarked R, Python and Rust data processing on a real-world data set of 65 GB. They discovered significant performance differences among languages, with Rust performing with an average run time of 8.40 seconds, compared to Python 3.11's 28.45 seconds for a single 1.9 GB file (R was much slower) [11]. They also found that Rust yielded further performance gains with parallelization via Rayon library, but I/O bottlenecks prevented linear scaling in terms of the number of threads [11].

While other research has focused on language-specific performance or interoperability with a focus on concurrency, there is a need for holistic benchmarking of Python-Rust hybrid systems for many different types of general tasks.

This study addresses this gap by comparing the run-time performance of pure Python, to PyO3-based Python-Rust hybrid solutions for five task types - arithmetic, strings, list processing, I/O and conditional statements - using a consistent and rigorous statistical methodology [9]. This study also explores the effects of reduced runtimes and memory usage on green software engineering and energy consumption [4].

METHODOLOGY

A. Experimental Environment

All experiments were conducted on a system running Ubuntu 24.04 LTS through Windows Subsystem for Linux (WSL) on Windows 11 [10]. The host machine was equipped with an 11th Gen Intel® Core™ i7-1185G7 processor operating at 3.00 GHz (1.80 GHz base frequency) and 16 GB of RAM (15.4 GB usable). The WSL environment was configured with 8 CPU cores and 8 GB of RAM (7.46 GB usable).

Python benchmarks were executed using Python 3.12.3, and Rust functions were compiled with Rust 1.91.0 via the PyO3 library to enable cross-language integration [6]. The execution time was measured using the timeit module, which internally leverages time.perf_counter() for high-precision wall-clock timing [7].

B. Integration Workflow

Through the PyO3 interoperability layer, the experimental workflow blends the expressiveness of Python with the speed of Rust [6]. A specific lib.rs file contained the definitions of the Rust functions, which were then compiled into a shared library. For runtime invocation, the compiled module was sub-sequently imported into Python scripts (embedded_python.py). With this design, developers can take advantage of Rust's memory safety, zero-cost abstractions, and concurrency model for CPU-intensive workloads while maintaining Python's scripting flexibility for orchestration [3].

The execution pipeline can be summarized as follows:

- Python Layer: Defines test harnesses and invokes Rust functions via PyO3 bindings.
- Rust Layer: Implements optimized algorithms for arithmetic, string manipulation, list operations, and other compute-heavy functions.
- Benchmark Execution: The Python script executes both Python-native and Rust-integrated implementations using identical input sizes and iteration counts, capturing timing data via the timeit module.

This architecture ensures that performance comparisons reflect language efficiency rather than structural or algorithmic differences.

C. Function Implementations and Explanations

To compare the efficiency of different types of tasks, a number of test functions were implemented and run. Each and standard deviation [9] were computed for each function as follows:

function is designed to test a particular type of operation that is commonly used in general programming. In order to ensure a fair and balanced comparison between the two programming Languages, we have built implementations in both Python and Rust using the same algorithm.

The implemented functions are summarized below:

$$\bar{t} = \frac{1}{x}\sum_{i=1}^{x} t_i \quad (1)$$

$$\sigma = \sqrt{\frac{1}{x}\sum_{i=1}^{x}(t_i - \bar{t})^2} \quad (2)$$

1. Arithmetic Operations: Performs complex arithmetic calculations using the addition, subtraction, multiplication, division, and modulus operations in a loop for each integer n from 1 . . . x. This evaluates arithmetic and numeric operations in each language.
2. String Manipulation: Concatenates and reverses random strings n times to measure string manipulation and memory efficiency.
3. List Manipulation: Creates a list of n random integers, sorts the list and filters out even numbers. This is a test of dynamic data handling, sorting and filtering.
4. If-Else Logic: Returns the English word for a number between 0 and 100, using a series of if-else statements. This benchmark explores control-flow performance.
5. File I/O: Repeatedly writes to and reads from a large file n times to measure file system performance and input/output delays.

Each benchmark was implemented in two parallel versions:

- raw_python.py – containing pure Python implementations.
- embedded_python.py – invoking equivalent Rust functions defined in lib.rs through PyO3 bindings.

To reduce noise in the performance measurements, these functions were run several times under the same circumstances. Because the **Python** and **Rust** implementations use the same logic, observed differences can only be ascribed to runtime behavior and language-level execution, not algorithmic differences.

D. Benchmarking Process

For the Python and Rust implementations, the benchmarking procedurewas created to guarantee both statistical dependability and reproducibility. Th timeit module, which internall depends on the high-resolution time, was used to time each computational function.For accurate wall-clock measurements, use

the perf_counter() function. Each function was run once as a warm-up before the main timing runs in order to minimize noise and caching effects. timeit was then used to carry out the primary benchmarking.repeat() generates independent timing samples by running each function with number=1 for a predetermined number of repetitions r = 5. From the gathered timing samples $t_1, t_2, \ldots, t_r$, the mean execution time

The performance distribution for each task was summarized using these formulas, which were directly implemented in the Python benchmarking scripts. This measurement procedure was consistently applied across all computational workloads, including arithmetic operations, string manipulation, list sorting and filtering, file I/O, and conditional branching, by the routine benchmark_func().

RESULTS AND ANALYSIS

Twenty separate iterations of each function were used to record the benchmark results for the Python and Rust implementations. Using the statistical formulas outlined in Section III, the mean time and standard deviation were recorded for each execution. Table I provides a summary of the findings.

TABLE I
EXECUTION TIME COMPARISON OF PYTHON AND RUST IMPLEMENTATIONS

Implementation	Function	Mean Time (s)	Std Dev
Rust	arithmetic operations	0.455470	0.001698
Rust	string processing	4.938672	0.012465
Rust	list manipulation	0.072411	0.000854
Rust	file io	2.539615	0.030167
Rust	number to name	6.835317	0.071374
Python	arithmetic operations	0.971004	0.006519
Python	string processing	0.611439	0.004426
Python	list manipulation	0.048115	0.000439
Python	file io	0.674027	0.009163
Python	number to name	3.340496	0.007858

A. Performance Overview

The findings unequivocally demonstrate that performance traits differed among tasks based on the computational pattern:

Rust outperformed Python in arithmetic-operations, finishing the assignment in almost half the time. The enhancement clearly benefits CPU-bound numerical workloads due to Rust's low-level loop efficiency and lack of interpreter overhead.

Python significantly outperformed Rust in string-processing tasks. Because of repeated string allocations and the overhead of converting data between Python and Rust, Rust had a significantly higher mean execution time. For this task, Python's internal optimizations and built-in string functions proved to be far more effective.

List manipulation favored Python, which outperformed Rust in terms of execution time. Despite Rust's inherent speed, Python completes the task faster because its list

operations are highly optimized in C and the vector conversion overhead on the Rust side adds extra

latency.

File I/O performance also leaned toward Python, has a significantly shorter run-time. The efficient file manipulation capabilities in Python's WSL environment showed greater throughput because I/O-intensive tasks are highly dependent on system file buffering (caching) and disk access times.

The number_to_name function showed the widest performance gap, Python is significantly faster. Rust's implementation led to the need for multiple string builds, thus being inefficient. In contrast, Python is better able to achieve success thanks to its string formatting and dictionary operations.

B. Comparative Trend Analysis

Our results are consistent across all the tasks. Rust was very good for arithmetic-intensive tasks, where native execution is beneficial for number crunching, numerical loops and tight inner loops. Python was also extremely successful in workloads that had significant contributions from high-level built-ins (ex. list and string operations), as the C-implemented libraries and low overhead of the interpreter greatly accelerate these operations. Python again outperformed the other languages in the I/O-bound workloads thanks to better buffering in the WSL environment, and lower cross-language costs.

C. Task-Specific Bottleneck Analysis

There are consistent patterns across the workloads. Rust was very good for arithmetic-intensive tasks, where native execution is beneficial for number crunching, numerical loops and tight inner loops. Python was also extremely successful in workloads that had significant contributions from high-level built-ins (ex. list and string operations), as the C-implemented libraries and low overhead of the interpreter greatly accelerate these operations. Python again outperformed the other languages in the I/O-bound workloads thanks to better buffering in the WSL environment, and lower cross-language costs.

D. Stability of the Systems

Analysis of the standard deviations of the times, shows less variance (more stability) for most workloads in Python, particularly with list and string operations. Rust had greater variance in file-I/O and string tasks, largely due to the cost of allocations and system issues in WSL. Both have low variability in arithmetic operations, where predictable time is required to perform CPU-intensive operations.

DISCUSSION

The standard deviations of Python indicate its run-times were more consistent in most of the tasks, in particular list and string operations. The variance value for Rust in file-I/O and string processing were greater mostly due to the memory allocation and other system-level overhead of WSL. The standard deviations in both languages for arithmetic operations were low, as the CPUs were mostly consistently used.

However, Python had a better performance in tasks that were the result of the effective built-in functions, like list and string manipulations. These are highly-optimised functions, programmed directly in C in the Python interpreter. They took longer before due to additional operations. This test is exemplified by the slow number-to-name test (where dynamic string creation in Rust was slow).

I/O's performance was erratic. We believe that Python's performance was a result of the test environment (WSL) where the buffering and translation layers of the system affected the performance during the runtime of the programs.

The results indicate a better combination of Python and Rust. Python is suited at higher-level operations that benefit from built-in optimisations and Rust should be used for the intensive operations that can be

bottlenecks. This gives the speed of development and the required performance.

CONCLUSION

This study looked at how integrating Python and Rust can improve the performance of Python apps with CPU-heavy workloads. Testing outcomes show that due to its compiled and efficient memory management, Embedding Rust shines in certain areas such as arithmetic operations and loop-based processing. Embedding Rust was still slower than Python for operations such as string and list operations that relied on Python's powerful built-in functions. File I/O, which showed varying results, was influenced by the execution environment. Overall, the performance results show that combining Python and Rust is a viable approach to improving performance without compromising Python's productivity. By delegating some performance-intensive tasks to Rust for faster execution, programmers can continue to use Python for problem-solving tasks. Such an approach is suitable for modern data-intensive and compute-intensive applications as it combines
speed and productivity.

FUTURE WORK

There are several ways this research can be furthered in the future. First, the performance of Python-Rust can be evaluated by applying more advanced techniques of optimization, such as tuning, benchmarking, compiler optimizations, and profiling, rather than just benchmarking. These techniques would help us to understand how these languages behave under different optimizations.

Second, there's an opportunity to develop a tool or library to make it easier for developers to bridge Python and Rust seamlessly. These can make it easier for developers to use Rust without needing to know all the details of the Rust ecosystem by automating the generation of modules, dependency tracking and build management.

The third opportunity is to better verify the usefulness of such integration in real-world applications. Testing in practical applications such as data pipelines, scientific computing, web or automation systems would facilitate performance comparisons under realistic conditions and constraints.

Additionally, to rule out any WSL-induced overhead, further research might include testing on a native Linux environment, memory consumption, and multi-core performance tests.

REFERENCES

1. M. Lutz, Learning Python, 5th ed. Sebastopol, CA, USA: O'Reilly Media, 2013.
2. N. A. Chow, "CPython: Enhancing Python's Performance and Versatil-ity," ResearchGate, 2023. doi: 10.13140/RG.2.2.17190.50247/4.
3. S. Klabnik and C. Nichols, The Rust Programming Language, San Francisco, CA, USA: No Starch Press, 2019.
4. S. Agarwal, A. Nath, and D. Chowdhury, "Sustainable approaches and good practices in green software engineering," *International Journal of Research and Reviews in Computer Science (IJRRCS)*, vol. 3, no. 1,
5. pp. 1425–1429, Feb. 2012.
6. M. Minin, "Enhancing Python Performance With Rust Integration," 2025 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), Sochi, Russian Federation, 2025, pp. 879-883, doi: 10.1109/ICIEAM65163.2025.11028364.

7. [12] PyO3 – Rust bindings for the Python interpreter, version-0.27.1. [Online]. Available: https://docs.rs/crate/pyo3/latest. Accessed: Nov. 13, 2025.
8. S. Singh, Benchmarking Scripts “rust-lab”, GitHub repository, 2025. [Online]. Available: https://github.com/srikantsingh673/Benchmarking-Python-Rust-HPC.git Accessed: Dec. 01, 2025.
9. D. Beazley, “Understanding the Python GIL,” PyCon US, 2010. [Online]. Available: https://dabeaz.com/python/UnderstandingGIL.pdf
10. D. S. Moore, G. P. McCabe, and B. A. Craig, Introduction to the Practice of Statistics, 8th ed., New York, NY, USA: W. H. Freeman and Company, 2014
11. J. Saroha, ”Windows Subsystem for Linux (WSL) 2.0: A Review,” Int.
12. J. Enhanced Res. Sci. Technol. Eng., vol. 13, no. 7, pp. 94-101, Jul. 2024
13. L. Beierlieb, A. Bauer, R. Leppich, L. Iffla¨nder, and S. Kounev, ”Effi-cient Data Processing: Assessing the Performance of Different Programming Languages,” in Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE), 2023, pp. 83–87.