GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Impact of GIL-less Cpython on Performance and Compatibility

Frederik Hennecke

MatrNr: 21765841

Supervisor: Patrick Höhn

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2025

Abstract

The Global Interpreter Lock (GIL) in CPython has long been a performance bottleneck for multi-threaded CPU-bound tasks, limiting Python's ability to use modern multi-core processors. This study assesses the impact of PEP 703, which proposes an optional GIL-less CPython, on performance and compatibility. By benchmarking diverse workloads, including string operations, compression, dictionary manipulations, Input/Output (I/O), numerical computing (NumPy, Pandas), and machine learning (PyTorch), we analyze the trade-offs introduced by GIL removal. Results demonstrate performance gains (up to 80%) for CPU-bound multi-threaded tasks like compression and dictionary operations, confirming the GIL as a critical bottleneck. However, performance deteriorations were observed in single-threaded execution and libraries such as NumPy and Pandas, likely due to overhead from the new threading model. While GIL-less Python unlocks true parallelism for specific workloads, compatibility issues and variable performance outcomes show the need for further optimization.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work, I have used ChatGPT or a similar AI-system as follows:

- ☐ Not at all
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☑ Other, namely: Grammarly for wording

I assure you that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Figures

List of Listings

List of Abbreviations

AI Artificial Intelligence

API Application Programming Interface

CPU Central Processing Unit

GIL Global Interpreter Lock

GPU Graphics Processing Unit

I/O Input/Output

JIT Just-In-Time

PEP Python Enhancement Proposal

RAM Random-Access Memory

1 Introduction

1.1 Background and Motivation

One of the most popular programming languages is Python, renowned for its ease of use, readability, and library ecosystem. The GIL, a feature of CPython that prevents Python threads from running in true parallel, is one of its more established drawbacks. In order to prevent multi-core use for Central Processing Unit (CPU)-bound applications, the GIL ensures that only one thread runs Python bytecode at a time. This design decision has historically been a major performance bottleneck for multi-threaded systems[2], despite making memory management and thread safety simpler.

Many programming languages, such as Java, C++, and Go, provide true parallelism for multi-threaded workloads, allowing efficient use of multi-core processors. In contrast, Python developers seeking parallel execution have traditionally relied on multiprocessing (which spawns separate processes instead of threads), C extensions (e.g., NumPy, which runs multi-threaded computations in C), or Just-In-Time (JIT)-based solutions like PyPy. While these approaches offer workarounds, they introduce overhead and complexity, making Python less competitive for high-performance computing and concurrent applications.

Python Enhancement Proposal (PEP) 703[8] suggests making the GIL optional in order to get around this restriction and enable Python to function without it as needed. By removing the GIL, Python may fully utilize multi-core CPUs, which enhances performance for jobs that depend on multi-threading. New difficulties are brought about by this change, mainly in the areas of garbage collection, memory management, and compatibility with current C extensions that depend on the GIL for implicit thread safety.

This study aims to evaluate whether the proposed GIL-free Python offers tangible performance improvements while maintaining backward compatibility and usability. By benchmarking different workloads, including numerical computing, data manipulation, I/O operations, and deep learning inference, this research provides insights into whether Python can evolve into a truly multi-threaded language without sacrificing its ease of use and existing ecosystem.

1.2 Objectives

The primary objective of this study is to evaluate the performance and compatibility of GIL-less Python, as proposed in PEP 703, compared to the traditional GIL-enabled CPython. We will conduct a series of benchmarks to determine whether removing the GIL provides benefits for multi-threaded workloads while maintaining single-threaded performance. Additionally, we examine how well libraries such as NumPy, Pandas, and PyTorch interact with the new threading model.

This study also investigates compatibility challenges introduced by the removal of the GIL. Many Python extensions are designed under the assumption that the GIL provides implicit thread safety, and its removal could result in unexpected behavior.

1.3 Structure

This report has the following structure:

- Section 2: Methodology is an overview of Python and the GIL. Furthermore, we talk about the proposed changes for the GIL-less CPython implementation.

- Section 3: Compatibility examines how Python libraries interact with the new threading model, and we discuss the potential challenges the developers have to adapt their existing codebase to work without the GIL.
- Section 4: Benchmarks presents a series of performance tests. These tests include numerical computing, data manipulation, compression, I/O, and machine learning inference.
- Section 5: Performance Analysis shows and interprets the benchmark results, highlighting the advantages and disadvantages of GIL-less Python.
- Section 6: Conclusion summarizes the key findings and discusses future directions.

2 Methodology

This section will introduce the GIL, which is used in CPython. It will also present its advantages and disadvantages. Finally, we will also examine PEP 703.

2.1 Python

Python is widely used for its simplicity and ease of development, making it a popular choice for prototyping and research-oriented applications. However, its design as an interpreted and dynamically typed language introduces significant performance limitations.

One major bottleneck in Python is the GIL. Which we will talk about in subsection 2.2. Another key limitation is dynamic typing. While it provides flexibility by allowing variables to change types during execution, this results in increased overhead since the interpreter must perform type checking at runtime. In contrast, statically typed languages perform these checks at compile time, making execution faster.

Python also uses automatic memory management, including reference counting and garbage collection. While these features simplify programming, they introduce unpredictable performance overhead due to periodic memory deallocation pauses, which can slow down performance.

2.2 The Global Interpreter Lock (GIL)

In CPython, the global interpreter lock, or GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once.[5] It simplifies memory management by ensuring thread safety for Python objects, making multithreaded code much more uncomplicated.

The GIL works by allowing only one thread at a time to execute Python bytecode. When a thread runs, it acquires the GIL mutex, executes the instructions, and then releases the GIL so that other threads can follow. This creates a bottleneck in multithreaded programs.

However, the GIL is less of an issue for certain tasks, like I/O bounded tasks, or for specific libraries like NumPy. I/O bounded tasks, such as network or file operations, where threads spend most of the time waiting for external resources, are not necessarily worse due to the GIL because some threads can execute their instructions while others

are waiting for their external resources. Numpy itself is built around the GIL, but more on that in section subsubsection 5.1.6.

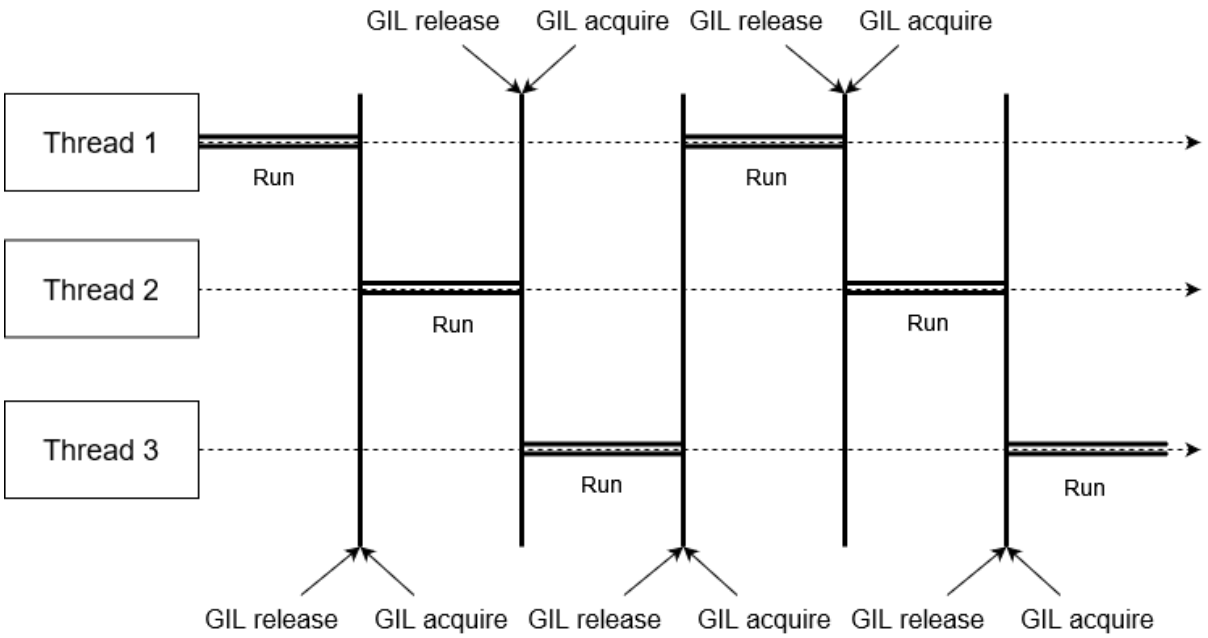


Figure 1: Only one thread can compute at the same time due to the GIL.

2.3 PEP703: Making the Global Interpreter Lock Optional in CPython

PEP 703 proposes several modifications to CPython's runtime that allow Python to run without the GIL. The primary objectives include:

- Remove the GIL as a Barrier to Concurrency: Enable true parallel execution of threads, improving performance for multi-threaded programs.
- Maintain Backward Compatibility: The GIL-optional mode should not break existing code or libraries that rely on the GIL's behavior.
- Enable Gradual Adoption: The plan is to initially introduce the GIL-optional mode as an opt-in feature, allowing developers and library maintainers to test and adapt their code over time.

The main changes are the following:

- GIL-Optional Build Mode: CPython will be modified to support a build mode where the GIL can be disabled.
- New Threading Model: The internal threading model of CPython will be updated. This includes changes to memory management, reference counting, and other core components so that they work without relying on the GIL.
  - Atomic Reference Counting: Reference counting will be made atomic to prevent certain race conditions when multiple threads modify the same object.

  - Memory Management Changes: The memory allocator and garbage collector will be updated to make them thread-safe.

- API and ABI Compatibility: The Python C API will be updated to support GIL-optional mode while maintaining compatibility with existing extensions.

- Performance Optimizations: Various performance optimizations will be implemented to minimize the overhead of running without the GIL.

2.4 Enabling and Disabling GIL-less Python

Since Python 3.13, users have the option to enable or disable the GIL at runtime or compile time. Python can be built in either GIL or GIL-less mode. This is done using a compiler flag. There is also an environment variable that can be used to toggle GIL behavior without recompiling Python. This can only be used if CPython was compiled in GIL-less mode.

In GIL-less CPython builds, developers can toggle the GIL dynamically within a running application. This means that even in a GIL-less environment, a program can re-enable the GIL if needed for compatibility with existing libraries or legacy code. This allows a gradual transition toward a fully GIL-free Python system. To ensure compatibility with libraries and dependencies, applications can check and modify the GIL settings.

While it is possible to toggle between GIL and GIL-less, performance decreases still occur due to the overhead of the new threading model in the GIL-less version with GIL enabled.

2.5 Related Work

Several studies have explored ways to improve multi-threading in Python and mitigate the limitations of the GIL. One notable approach is the integration of OpenMP with Python, as demonstrated in recent work on native OpenMP implementations for multi-threading. This research suggests that well-designed OpenMP integrations can significantly improve performance for computationally intensive tasks, particularly in a GIL-free Python environment[7].

Beyond OpenMP, alternative approaches such as JIT compilation (e.g., PyPy) and multiprocessing-based parallelism have been investigated as potential solutions to Python's threading limitations. While PyPy improves execution speed through Just-In-Time compilation, it does not directly address the multi-threading constraints imposed by the GIL[3]. The multiprocessing module, on the other hand, allows parallel execution by spawning separate processes but introduces higher memory overhead and inter-process communication costs[1].

The proposed GIL removal in Python 3.13 (PEP 703) represents the most direct effort to achieve true parallel execution.

3 Compatibility

3.1 Library Support and Challenges

The removal of the GIL introduces challenges for existing Python libraries that rely on CPython internals or C extensions. Many libraries assume that the GIL ensures thread safety. This simplifies memory management. But without changes, this becomes problematic in GIL-less Python. Some challenges include:

- Thread Safety in C Extensions: Many extensions rely on the GIL for memory management and reference counting. Without it, these libraries must implement their own thread safety.
- Concurrency Issues: Libraries that interact with low-level threading mechanisms may need to be refactored to handle concurrent execution safely.
- Performance Considerations: Some operations may experience performance overhead due to new fine-grained locking mechanisms replacing the GIL.

3.2 Making Libraries Compatible

Below are some considerations and steps the developers can take to achieve compatibility. These items are mostly for packages written in languages other than Python.

Global variables, class attributes, or shared data structures might need explicit synchronization mechanisms like `threading.Lock`, `threading.RLock`, or `threading.Semaphore`.

Python's C Application Programming Interface (API) now provides new functions and macros to manage thread safety explicitly. Library developers may have to switch to the new API.

Immutable data structures are inherently thread-safe because they cannot be modified after creation. Datatypes like `tuple`, `frozenset`, or libraries like `pyrsistent` may help reduce the need for synchronization.

Python allows dynamic toggling of the GIL, enabling developers to transition libraries gradually rather than requiring an immediate rewrite.

4 Benchmarks

4.1 Setup

We use pyenv, a tool for managing multiple Python versions for the benchmarks. This allows us to install and switch between different Python builds, including the standard version and the GIL-less version proposed in PEP 703. We decided to use Python 3.12.8 as the baseline version, Python 3.13.1 with the GIL enabled, and Python 3.13.1 without the GIL. Installation steps:

- `pyenv install 3.12.8` – Installs a standard Python 3.12.8 version.
- `python-build 3.13.1 .pyenv/versions/3.13.1-gil` – Builds Python 3.13.1 with the default GIL-enabled configuration.

- `PYTHON_CONFIGURE_OPTS='--disable-gil' pyenv install 3.13.1` – Installs Python 3.13.1 with the GIL disabled, allowing us to compare performance.

These installations provide a controlled environment for testing and ensure ease of use across different Python versions.

As for the library versions, we used the following in all Python versions:

- NumPy: 2.2.3
- Pandas: 2.2.3
- PyTorch:
  - torch: 2.4.0
  - torchvision: 0.19.0

Each benchmark was executed 20 times, and the average runtime was used to ensure accuracy. The workload for each test was adjusted to ensure that it ran for an adequate duration, preventing execution times that were too short or too long.

Each benchmark was executed on the following hardware:

- CPU: Intel i7 10700k; 8 cores, 16 threads, 5.10 GHz frequency
- Random-Access Memory (RAM): 32GB DDR4 3000Mhz in dual-channel
- Storage: 2TB SATA SSD
- Graphics Processing Unit (GPU): AMD Radeon RX 7900 XT

We used Debian 12 as our operating system, ensuring that no background processes were active during the operation. Therefore, we maintain system performance and integrity.

For each benchmark, we have plots showing both the raw runtime and the performance improvement relative to Python 3.12.8, which serves as the baseline. Each number of threads has its own baseline, ensuring that we evaluate the performance improvements of different Python versions independently rather than focusing on whether increasing the number of threads improves performance.

4.2 Overview

All benchmarks follow a consistent structure to ensure reliable and comparable results. Each benchmark:

- Runs with 1, 2, 4, 8, and 16 threads to measure scalability and multi-threading performance.
- Generates data outside of the timed operations to avoid including setup overhead in the measurements.
- Uses `time.perf_counter` to get accurate performance measurements.
- Focuses on common Python workloads to evaluate real-world performance impact.

The selected benchmarks test frequently used Python capabilities, including numerical computing (NumPy, pure Python), data manipulation (Pandas, dictionaries), string operations, compression, file I/O, and deep learning (PyTorch).

All our benchmarks can be found in our GitLab repository[1].

4.3 Benchmark Scenarios

4.3.1 String Operations

The benchmark seen as a short snippet in Listing 1 evaluates the performance of string manipulation tasks, such as reversing strings, converting them to uppercase, and concatenating strings. The benchmark first generates a list of strings, each 50 characters long, and distributes the workload across multiple threads. Each thread processes only its designated subset of the string list.

```
def process_strings(thread_id, data, start_idx, end_idx, iterations):
    result = ""
    for _ in range(iterations):
        for i in range(start_idx, end_idx):
            s = data[i]
            processed = s[::-1].upper() + s.upper()
            result += processed
```

Listing 1: Processes string data by reversing and uppercasing each string.

This benchmark was chosen as string manipulation is an important task in text processing, logging, data transformation, and natural language processing. Removing the GIL might improve performance in this benchmark because string operations are CPU-bound, and multiple threads could execute concurrently without the GIL limiting parallelism.

4.3.2 Compress, Decompress

This benchmark evaluates the performance of data compression and decompression using the built-in `zlib` module. The benchmark generates a dataset of strings, each containing 10,000 characters, and distributes the workload across all threads. Each process works on its distinct dataset and repeatedly compresses and decompresses the strings.

```
def compression_worker(thread_id, data, start_idx, end_idx, iterations):
    for _ in range(iterations):
        for i in range(start_idx, end_idx):
            text = data[i]
            compressed = zlib.compress(text.encode())
            decompressed = zlib.decompress(compressed)
```

Listing 2: Compresses and decompresses data slices using zlib.

We chose this benchmark since CPU-intensive compression algorithms involve significant memory operations. It is also used in real-world applications like streaming, data storage, and distributed computing. For this benchmark, the removal of the GIL might improve performance because compression and decompression are CPU-bound tasks, and multiple threads could execute concurrently without the GIL limiting parallelism, especially for large datasets.

[1] https://gitlab.gwdg.de/frederik.hennecke/pythonnogil

4.3.3 Dictionary (Intersection, Union)

This benchmark evaluates the performance of dictionary and set operations in Python, which are widely used for data manipulation and lookups. Dictionary operations are essential in caching, data indexing, and fast membership testing. Since these operations involve significant memory access patterns and hashing, they provide a useful test case for evaluating multi-threaded performance under a GIL-less Python implementation.

```
def data_manipulation_worker(thread_id, data, start_idx, end_idx, iterations):
    for _ in range(iterations):
        data_chunk = data[start_idx:end_idx]
        sorted_data = sorted(data_chunk)
        dictionary = {i: val for i, val in enumerate(sorted_data)}
        for k in range(0, len(sorted_data), 10):
            dictionary[k] = dictionary.get(k, 0) + 1
        # Set operations
        data_set = set(sorted_data)
        set_intersection = data_set.intersection(set(range(len(data_chunk) // 2)
    ))
        set_union = data_set.union(set(range(len(data_chunk) * 2)))
```

Listing 3: Performs sorting dictionary and set operations on a data chunk.

The benchmark generates a large dataset of random integers and distributes the workload across multiple threads. Each thread processes a subset of the data, performing sorting operations, dictionary manipulations, and set operations. Specifically, the benchmark sorts the assigned data chunk, constructs a dictionary mapping indices to values, and modifies selected dictionary entries. Additionally, it performs set operations, including intersections and unions, which test Python's handling of hash-based structures.

The removal of the GIL might improve performance in this benchmark because dictionary and set operations are CPU-bound and involve significant memory access patterns. Without the GIL, multiple threads could execute these operations concurrently, potentially speeding up data manipulation tasks. However, we also have many data operations simultaneously, which might slow down GIL-less Python due to the added data accessing overhead.

4.3.4 Input and Output Operations

This benchmark listed in Listing 4 evaluates the performance of concurrent file read and write operations, which are critical for many real-world applications, including database management, logging, and data processing. Since file I/O operations often involve significant overhead due to disk access and operating system interactions, the performance gain will probably not be that high.

```
def write_file(thread_id, chunk_size, iterations):
    data = os.urandom(chunk_size)
    with open(FILE_PATH, "r+b") as f:
        for i in range(iterations):
            offset = thread_id * chunk_size
            f.seek(offset)
            f.write(data)

def read_file(thread_id, chunk_size, iterations):
```

```
    with open(FILE_PATH, "rb") as f:
        for i in range(iterations):
            offset = thread_id * chunk_size
            f.seek(offset)
            _ = f.read(chunk_size)
```

Listing 4: Writes and reades data chunk into a file.

The benchmark first generates a large test file filled with random binary data to simulate realistic workloads. It then measures the execution time of concurrent read and write operations across multiple threads. Each thread operates on a distinct section of the file to minimize contention. The file is accessed using standard Python file handling with seek(), ensuring that threads write to or read from non-overlapping sections. A bounded semaphore limits concurrent file access to prevent excessive contention and simulate real-world constraints.

Removing the GIL is unlikely to improve performance in this benchmark because file I/O operations are primarily limited by disk speed and operating system interactions and not CPU-bound tasks. The GIL's impact should be minimal here since the bottleneck is I/O latency, not Python's threading model.

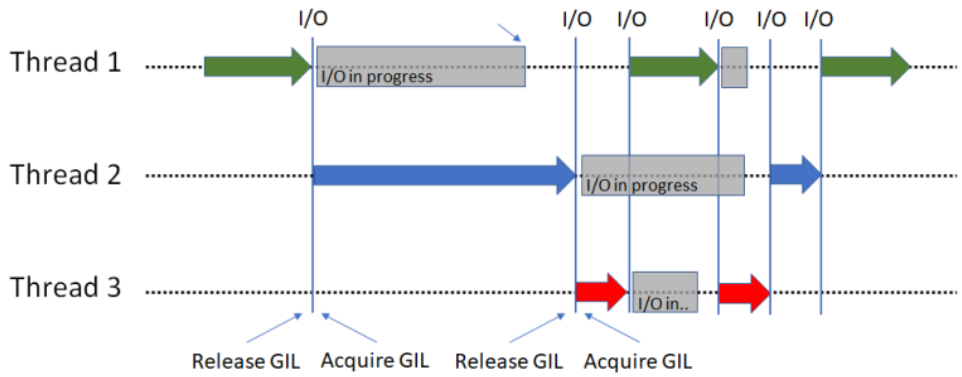


Figure 2: IO operations are faster with multiple threads - even with the GIL.[6]

4.3.5 Pure Python Matrix Multiplication

This benchmark evaluates multi-threaded performance in Python by implementing matrix multiplication in pure Python. While no one realistically performs matrix multiplication this way in real-world applications, since optimized libraries like NumPy handle such operations far more efficiently, it serves as a useful test for evaluating Python's threading behavior and potential improvements in a GIL-free environment.

```
def matrix_multiply(A, B, C, start_row, end_row):
    for i in range(start_row, end_row):
        for j in range(len(B[0])):
            C[i][j] = 0
            for k in range(len(B)):
                C[i][j] += A[i][k] * B[k][j]
```

Listing 5: Multiplies matrix A and B subsected by *start_row* to *end_row*.

This function performs matrix multiplication for a specific range of rows of the result matrix, computing the dot product of the row from the first matrix and columns from

the second. It works in parallel, allowing multiple instances to compute different parts of the result matrix simultaneously. In this benchmark, two randomly generated square matrices are multiplied using threads, with each thread responsible for computing a small subset of the result matrix's rows.

The removal of the GIL might improve performance in this benchmark because matrix multiplication is a CPU-bound task and multiple threads could compute different rows of the result matrix concurrently without the GIL limiting parallelism. This would allow for better utilization of multi-core processors.

4.3.6 Numpy Matrix Computations

This benchmark evaluates multi-threaded numerical computations on large matrices, focusing on common linear algebra operations such as matrix multiplication, element-wise addition, and transposition. The main goal is to test whether NumPy benefits from removing the GIL.

```
def process_matrices(arr, start_idx, end_idx, iterations=1000):
    for _ in range(iterations):
        for i in range(start_idx, end_idx):
            _ = np.dot(arr[i], arr[i])
            _ = arr[i] + arr[i]
            _ = arr[i] * arr[i]
            _ = arr[i].T
            _ = np.mean(arr[i])
            _ = np.sum(arr[i])
```

Listing 6: Performs various operations on subsets of NumPy matrices.

NumPy is designed to optimize performance by offloading heavy computations to compiled C, C++, or Fortran routines. When performing operations such as matrix multiplication (`np.dot`) or element-wise arithmetic, NumPy releases the GIL internally, allowing its underlying native code to run in parallel. Since NumPy already bypasses the GIL in computationally expensive functions, standard multi-threading in Python often does not lead to significant speedups. Instead, threading overhead, context switching, and memory contention between threads can sometimes degrade performance.

In this benchmark, each thread is assigned a subset of the matrix rows to process independently. However, because NumPy efficiently utilizes multiple cores via optimized libraries, launching additional Python threads may not meaningfully accelerate the computation. Instead, performance improvements are more likely when using multiprocessing, which avoids Python's threading model entirely by spawning separate processes with independent memory spaces.

4.3.7 Pandas (Filter, Mean, Merge, Lambda)

This benchmark evaluates multi-threaded performance when handling large datasets using Pandas, a widely used library for data analysis and manipulation in Python. The test measures operations such as filtering, grouping, merging, and applying transformations, which are common in real-world data processing tasks like natural language processing.

```
def pandas_benchmark_worker(thread_id, df, start_idx, end_idx, iterations):
    for _ in range(iterations):
        sub_df = df.iloc[start_idx:end_idx]
```

```
        filtered_df = sub_df[sub_df["A"] > 50]
        grouped_df = filtered_df.groupby("B")["C"].mean()
        merged_df = pd.merge(filtered_df, sub_df, on="B", suffixes=('_left', '
    _right'))
        transformed_df = merged_df["C_left"].apply(lambda x: x ** 2 + random.
    randint(1, 100))
```

Listing 7: Performs data manipulations on a Pandas dataframe slice.

A dataset is generated with three columns: A, containing random integers; B, a categorical column with repeated values; and C, a floating-point column representing numerical data. The dataset is divided into chunks, with each thread processing a separate portion of the data.

Each thread executes multiple iterations of data transformations, including filtering rows where column A exceeds a threshold, grouping data by column B to compute the mean of C, merging filtered results back with the original dataset, and applying a mathematical transformation to one of the columns. These operations are used in data analysis workflows like financial modeling, machine learning preprocessing, and large-scale analytics.

The removal of the GIL might improve performance in this benchmark because Pandas operations are CPU-bound, and multiple threads could process different chunks of the dataset concurrently without the GIL limiting parallelism.

4.3.8 Pytorch

This benchmark evaluates the impact of GIL removal on deep learning inference workloads using PyTorch. It tests whether multi-threaded deep learning model execution benefits from parallel execution in a GIL-free Python environment.

```
device = "cuda" if torch.cuda.is_available() else "cpu"
resnet = models.resnet18(pretrained=True).to(device).eval()
input_tensor = torch.randn(1, 3, 224, 224).to(device)
def run_inference(model, input_tensor, iterations):
    with torch.no_grad():
        for _ in range(iterations):
            _ = model(input_tensor)
```

Listing 8: Runs inference with a tensor input on a pretrained ResNet model.

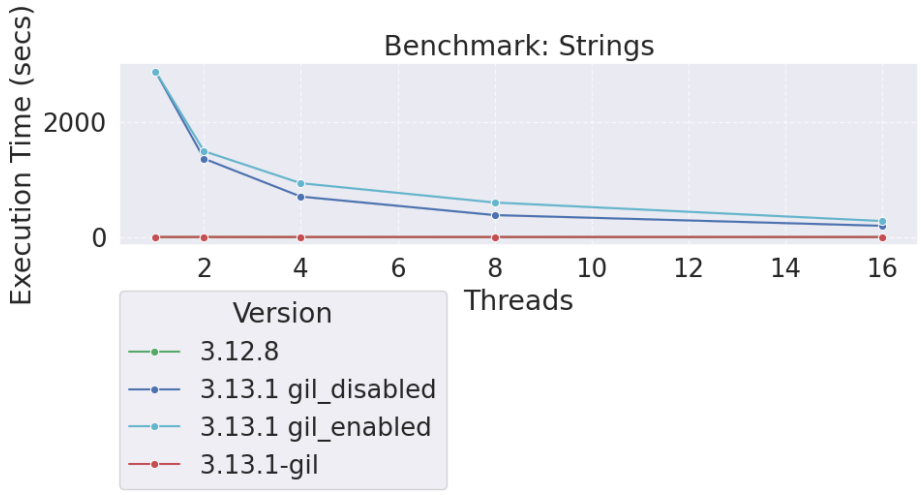
The benchmark loads a pre-trained ResNet-18 model and runs inference on a random input tensor multiple times. To simulate concurrent execution, multiple threads are launched, each performing a fixed number of inference iterations. By distributing the workload across threads, this test examines the efficiency of running deep learning inference in a multi-threaded Python setting. PyTorch's native operations are heavily optimized with multi-threading and GPU acceleration. However, in a GIL-constrained environment, Python threads executing inference may not fully utilize available CPU cores. This benchmark helps determine whether removing the GIL allows for improved concurrency and faster inference times in CPU-bound scenarios.

The paper "Using Python for Model Inference in Deep Learning"[4] suggests that the GIL may be a bottleneck when deploying Artificial Intelligence (AI) models. Therefore, we suspect that we might get some performance improvements.

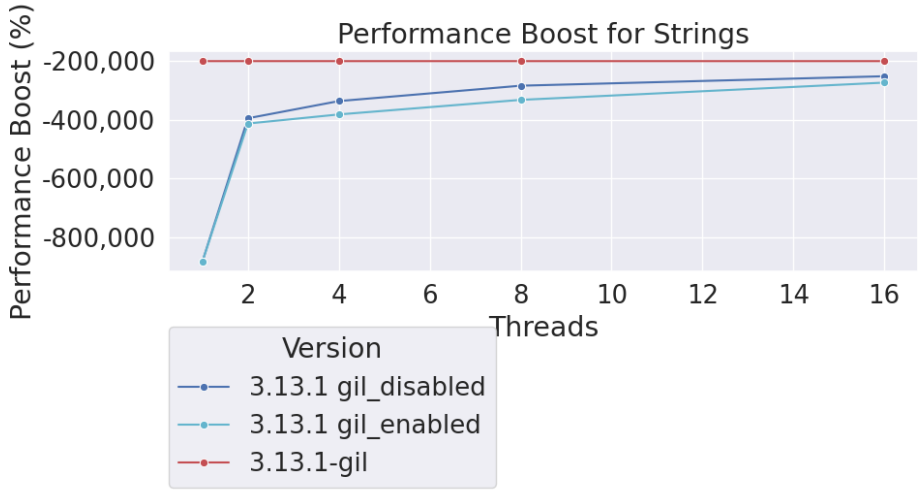
5 Performance Analysis

5.1 Benchmarks

5.1.1 String Operations



(a) Runtime in seconds for the string benchmark.



(b) Performance boost of string operations with Python 3.12.8 as the baseline.

Figure 3: String Benchmark

The string benchmark presents a dramatic performance discrepancy between Python 3.12.8, Python 3.13.1-gil, and the new threading model in Python 3.13.1.

For Python 3.12.8 and 3.13.1-gil, the runtime remains extremely fast, consistently below one second, regardless of the number of threads used. However, performance completely collapses for Python 3.13.1 with the new threading model (both GIL=0 and

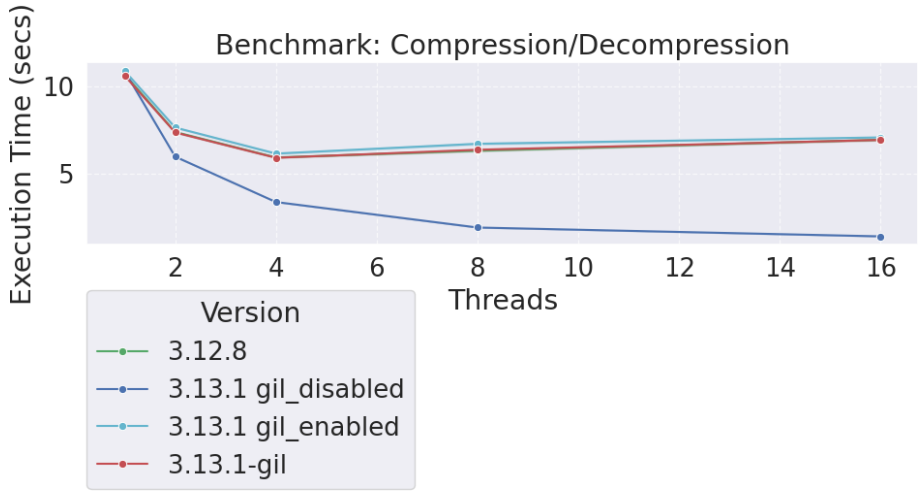
GIL=1). With one thread, the runtime skyrockets to nearly 3000 seconds. While adding more threads does improve the performance, even with 16 threads, the runtime is still around 195 seconds, which is orders of magnitude slower than the other versions.

These results strongly suggest that there may be a significant bug or inefficiency in the new threading model when handling string operations. Since string manipulation is a core feature of Python, this could indicate deeper issues in memory allocation, garbage collection, or internal locking mechanisms introduced in the new model.

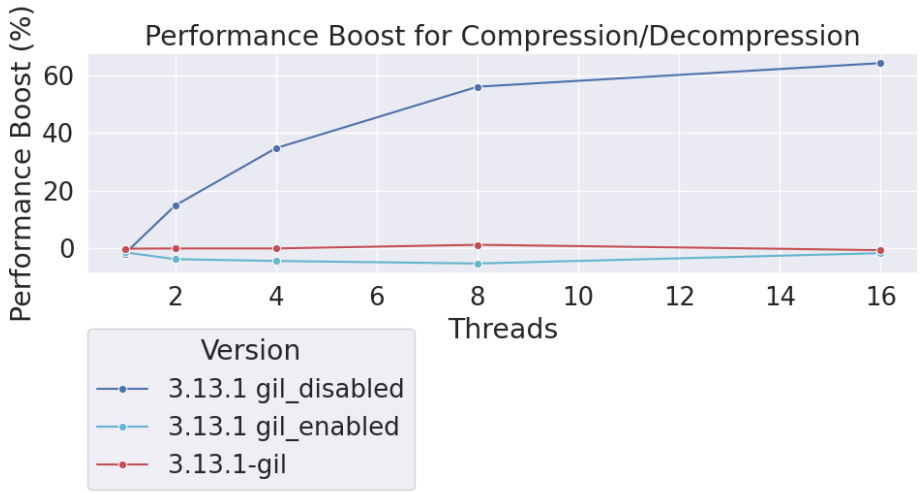
The most current Python version as of writing this report is Python 3.14.0a4[2]. With this version, we get faster performance than our current baseline. But we won't be looking further into this result, as that Python version is only an early developer build.

[2] `https://www.python.org/downloads/release/python-3140a4/`

5.1.2 Compress, Decompress



(a) Runtime in seconds for the compress/decompress benchmark.



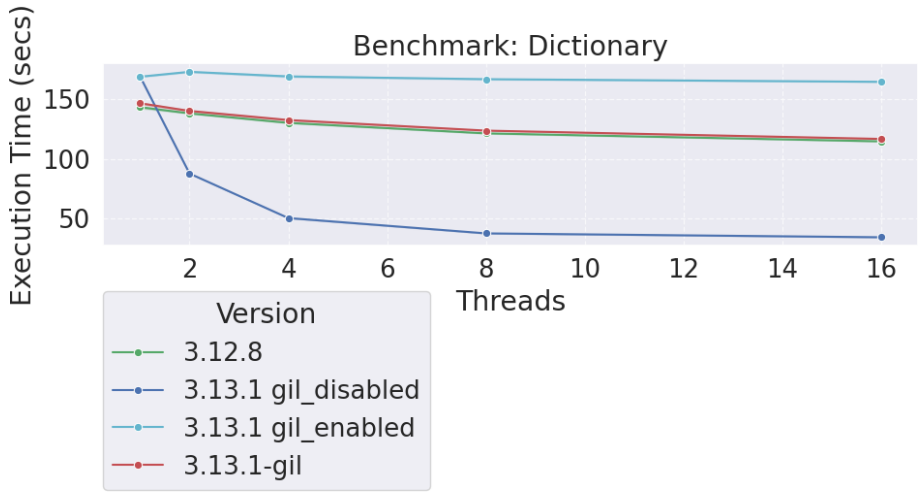
(b) Performance boost of compression/decompression operations with Python 3.12.8 as the baseline.

Figure 4: Compress/Decompress Benchmark

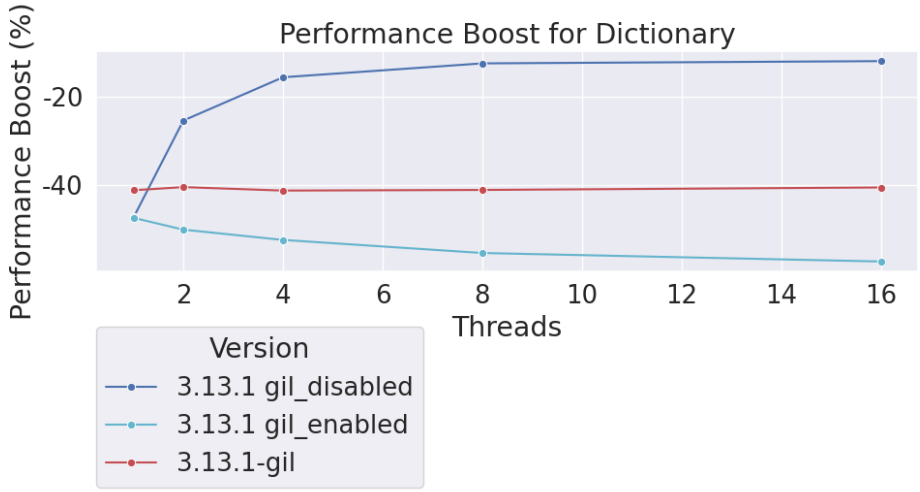
The results in Figure 4 show that when the GIL is enabled, the runtime remains relatively stable across Python 3.12 and 3.13.1, with only minor deviations. However, running Python 3.13.1 with the GIL disabled shows noticeable performance increases. With a single thread, the performance remains effectively unchanged, which aligns with expectations since the GIL does not affect single-threaded execution. We observe a 20% performance improvement with two threads, and with four threads, the improvement grows to over 40%. At eight threads, the performance boost reaches 70%, reducing execution time from 6.43 seconds in Python 3.12 to just 1.89 seconds in Python 3.13.1 with the GIL disabled. With 16 threads, we get around 80% performance improvement.

This result confirms that the GIL is a significant bottleneck in this specific multi-threaded workload. Although relatively lightweight in computation, the zip benchmark involves enough concurrent execution to showcase the impact of removing the GIL. Since Python's standard threading model under the GIL prevents true parallel execution of CPU-bound tasks, this benchmark highlights the benefits of a GIL-free environment where multiple threads can fully utilize available CPU cores. Interestingly, the performance with the GIL enabled remains mostly flat even when the number of threads is increased, showing that traditional multi-threading in Python does not scale well due to GIL constraints.

5.1.3 Dictionary (Intersection, Union)



(a) Runtime in seconds for the dictionary benchmark.



(b) Performance boost of set operations with Python 3.12.8 as the baseline.

Figure 5: Dictionary Benchmark

Looking at the results in Figure 5, Python 3.12.8 and Python 3.13.1 (with the GIL enabled) exhibit very similar performance. In fact, Python 3.12.8 is slightly faster, though the difference is minimal. Additionally, increasing the number of threads in these GIL-enabled versions does lead to small improvements in runtime, but only by a few percentage points per thread. However, when running Python 3.13.1 with the GIL disabled, the runtime performance does change. The single-threaded performance sees a slight decrease compared to Python 3.12.8, which could be due to internal changes in the interpreter or modifications required to support a GIL-free execution model. As soon as we introduce multiple threads, the performance improvements become noticeable.

With two threads, we see nearly a 40% reduction in runtime compared to the single-threaded case. This scales even further, reaching a 60% improvement with four threads and a 70% boost with eight threads, which stays the same for 16 threads. For reference, the execution time drops from around 179.8 seconds (single-threaded) to just 39.5 seconds with eight threads—a clear demonstration of how the removal of the GIL allows for true parallel execution.

One key takeaway from this benchmark is that dictionary operations, which are common in real-world Python applications, significantly benefit from multi-threading when the GIL is removed. Since dictionaries are used extensively in Python for things like object attribute storage, caching, and lookups, these results indicate that a GIL-free Python could offer substantial speedups in applications that rely heavily on dictionaries in concurrent execution scenarios.

5.1.4 Input and Output Operations

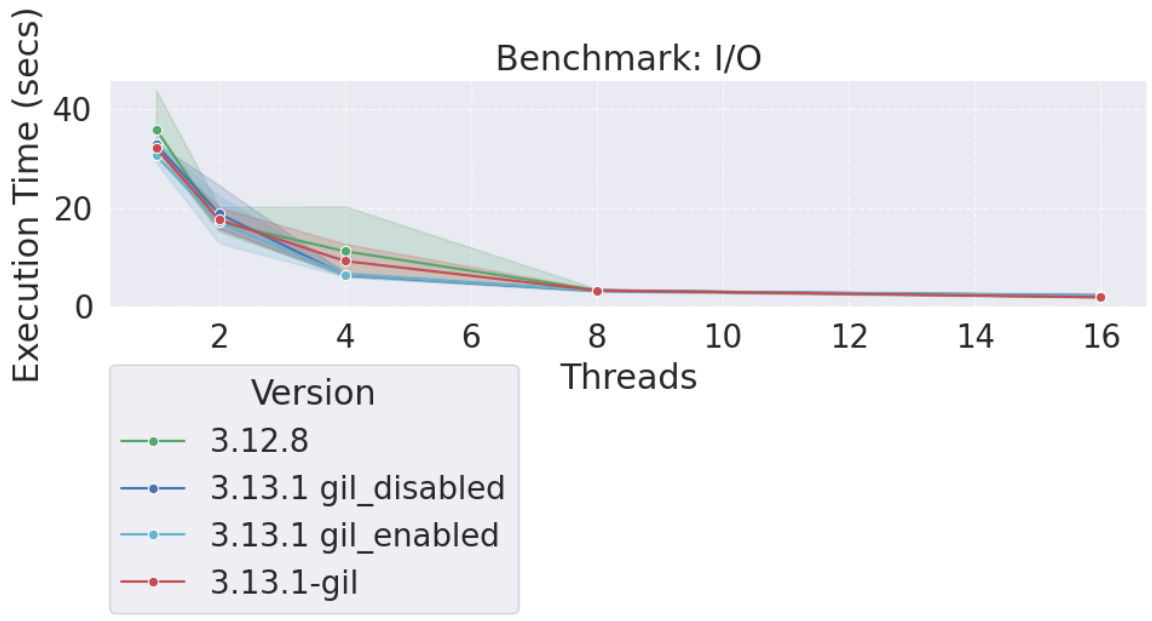


Figure 6: Runtime in seconds for the I/O benchmark.

The I/O benchmark results are straightforward. Regardless of whether the Python version is 3.12.8 or 3.13.1 and whether the GIL is enabled or disabled, the times remain nearly

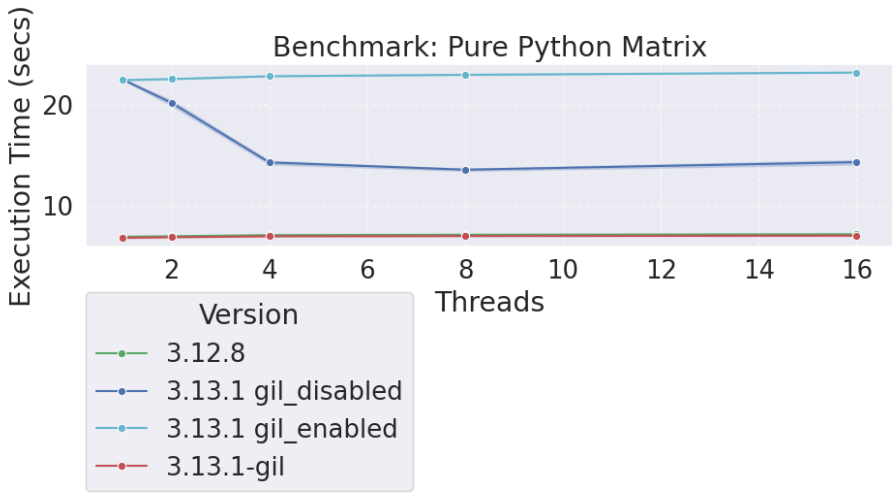
identical. This suggests that the limiting factor in these tests is not the GIL but rather the speed of the storage system.

As expected, the number of threads does improve performance. A single-threaded run takes on average around 32–34 seconds, while using two threads nearly halves the execution time to around 16–19 seconds. Increasing to four threads brings the execution time down to roughly five seconds, and at eight threads, we can see that the runtime is around three to four seconds. With 16 threads, we get the best performance at around two seconds. This behavior aligns with what we would anticipate in an I/O-heavy workload: more threads allow for greater parallelism, but the overall speed is ultimately limited by the hardware's ability to read and write data.

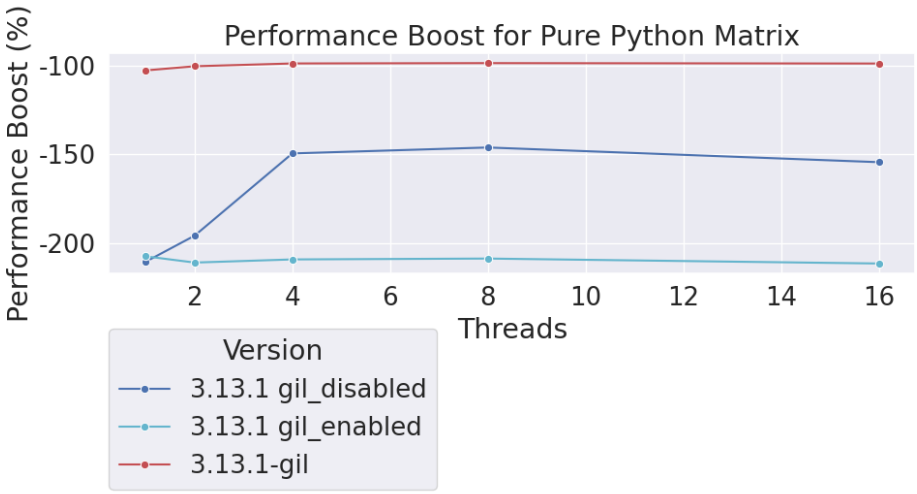
The runtime results occasionally showed slight deviations, which could be attributed to caching mechanisms, disk scheduling variability, or background system activity. In most cases, repeated runs show faster execution times, likely due to the operating system caching frequently accessed data, reducing actual disk read/write operations.

Since file I/Ooperations are inherently limited by disk throughput, changes to Python's threading and GIL behavior have no real impact here. Whether the GIL is enabled or not, the storage systems' performance still constrains Python, and improvements in execution efficiency at the Python level do not make a noticeable difference in I/O-bound tasks.

5.1.5 Pure Python Matrix Multiplication



(a) Runtime in seconds for the matrix multiplication benchmark.



(b) Performance boost of matrix multiplication with Python 3.12.8 as the baseline.

Figure 7: Math Benchmark

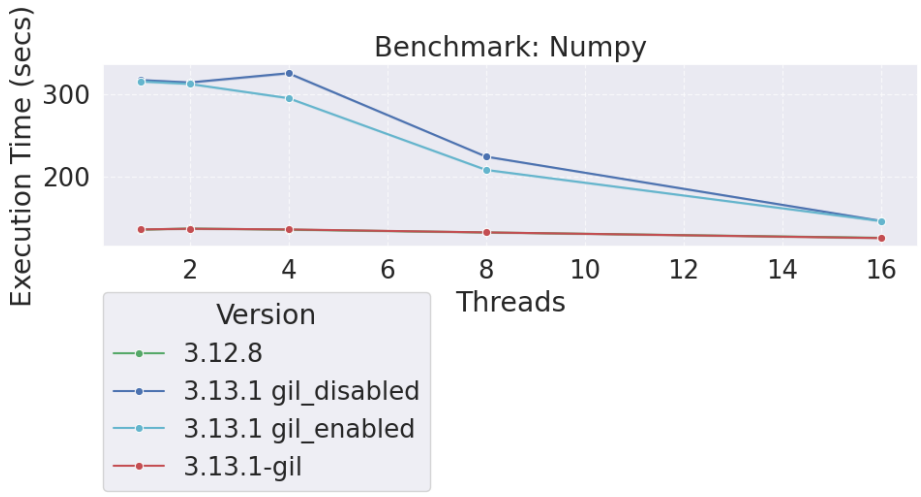
The results of the pure Python matrix multiplication benchmark can be seen in Figure 7. First, Python 3.12.8 and Python 3.13.1 with the GIL show nearly identical performance across all thread counts. Increasing the number of threads has almost no effect on execution time, suggesting that the GIL prevents any meaningful parallel execution in this particular workload.

Python 3.13.1 with GIL disabled shows significantly worse performance in single-threaded mode, taking around 22.5 seconds, over three times slower than the baseline. However, we see some performance improvement with more threads: 20.98 seconds with two threads, 14.97 seconds with four, 13.92 seconds with eight, and 13.81 seconds with 16 threads. This suggests that while Python 3.13.1 (with GIL disabled) can leverage multiple

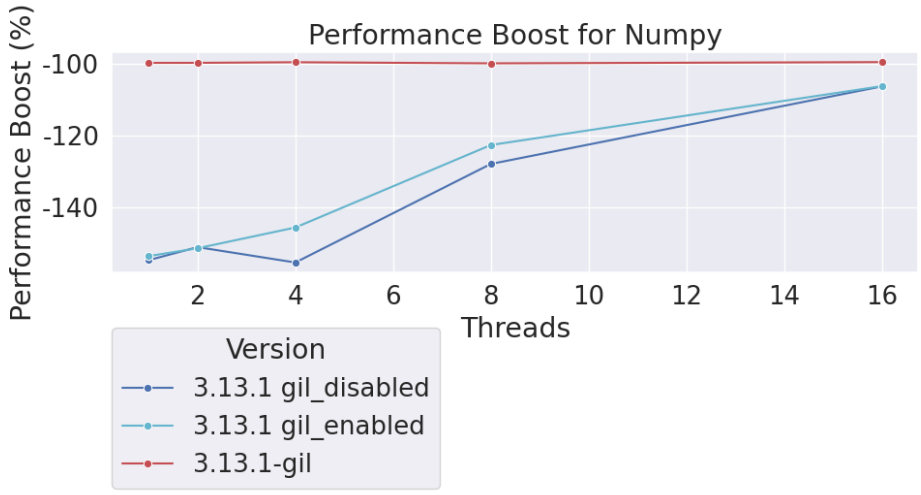
threads for performance gains, it starts from a much worse baseline performance, which prevents it from catching up to Python 3.12.8 or 3.13.1-gil.

Python 3.13.1 with GIL enabled but running multiple threads sees no such improvements. The execution time remains nearly constant across all thread counts, lingering around 22–23 seconds no matter how many threads are used.

5.1.6 NumPy Matrix Multiplication



(a) Runtime in seconds for the NumPy benchmark.



(b) Performance boost of NumPy operations with Python 3.12.8 as the baseline.

Figure 8: NumPy Benchmark

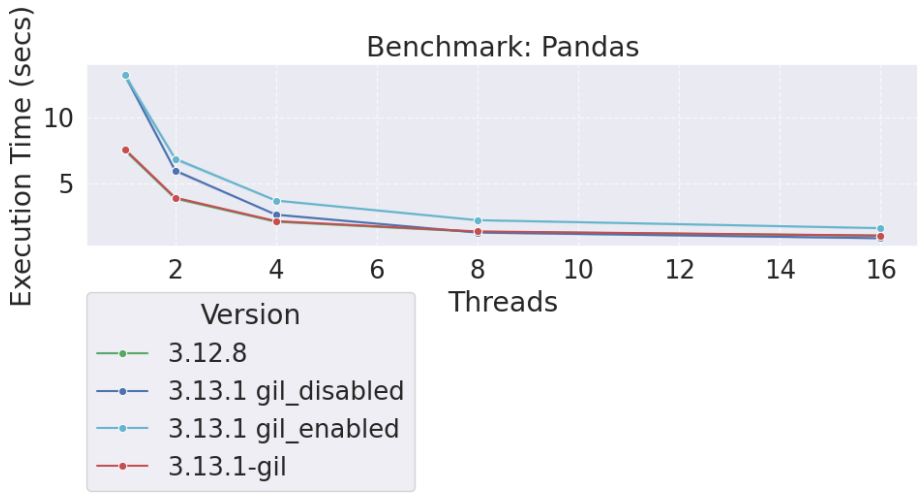
The NumPy benchmark, which can be seen in Figure 8, shows an interesting case where Python 3.12.8 outperforms all versions of Python 3.13.1, regardless of whether the GIL is enabled or not. In Python 3.12.8, the execution hovers around 70-75 seconds across different thread counts. There is a slight performance increase with threads, but the

improvement is minimal, suggesting that NumPy is already handling multi-threading efficiently.

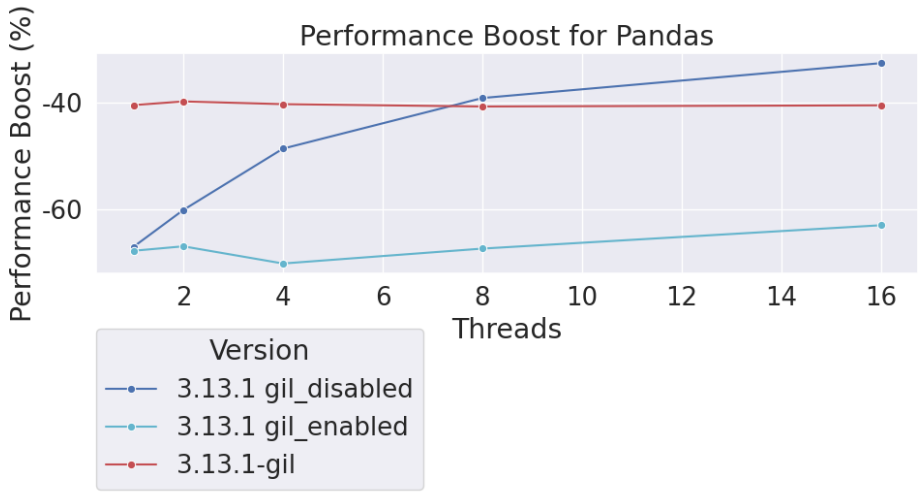
In contrast, Python 3.13.1, both with and without the GIL, shows worse performance. The single-threaded performance of Python 3.13.1 (with GIL enabled) is much slower at around 315 seconds, more than double the time of Python 3.12.8. Even with multiple threads, the performance does not scale well. The benchmark with 16 threads brings the execution time down to around 145 seconds, which still does not match the baseline performance of Python 3.12.8.

This performance degradation suggests that NumPy, at least currently, is not fully optimized for Python 3.13.1. Since NumPy is designed to work efficiently with Python's existing GIL-based threading model, it is possible that the changes in Python 3.13.1's memory management or threading behavior introduce inefficiencies when working with NumPy.

5.1.7 Pandas (Filter, Mean, Merge, Lambda)



(a) Runtime in seconds for the Pandas benchmark.



(b) Performance boost of Pandas operations with Python 3.12.8 as the baseline.

Figure 9: Pandas Benchmark

The Pandas benchmark tells a different story compared to previous tests, highlighting how Python's GIL removal does not always lead to performance gains. Here, Python 3.12.8 and Python 3.13.1-gil (both with the GIL enabled) perform almost identically.

For Python 3.12.8, the single-threaded execution time is around 30 seconds. This time is halved when running with two threads (15 seconds), and it further decreases to approximately 8 seconds with four threads and 4.24 seconds with eight threads.

However, the performance behavior changes when running Python 3.13.1 with GIL=0 or even GIL=1. In both cases, the single-threaded performance is worse than Python 3.12.8, with runtimes of around 13 seconds. The performance decrease could be due to changes in memory management or additional synchronization overhead introduced in

Python 3.13's new threading model. The performance does improve as more threads are added, with times decreasing to six seconds for two threads, 2.6 seconds for four threads, and 1.2 seconds for eight threads. With 16 threads, we get around 0.8 seconds. Notably, the only time the GIL-free version of Python matches the baseline performance of Python 3.12.8 is when using eight or 16 threads. This result is important: removing the GIL is not always beneficial. Running Pandas on a GIL-free Python interpreter introduces an overhead that slows down execution, making it clear that the GIL was not the bottleneck for Pandas.

This benchmark shows that while Python's GIL removal unlocks true parallelism for many workloads, Pandas users might not see substantial benefits.

5.1.8 PyTorch

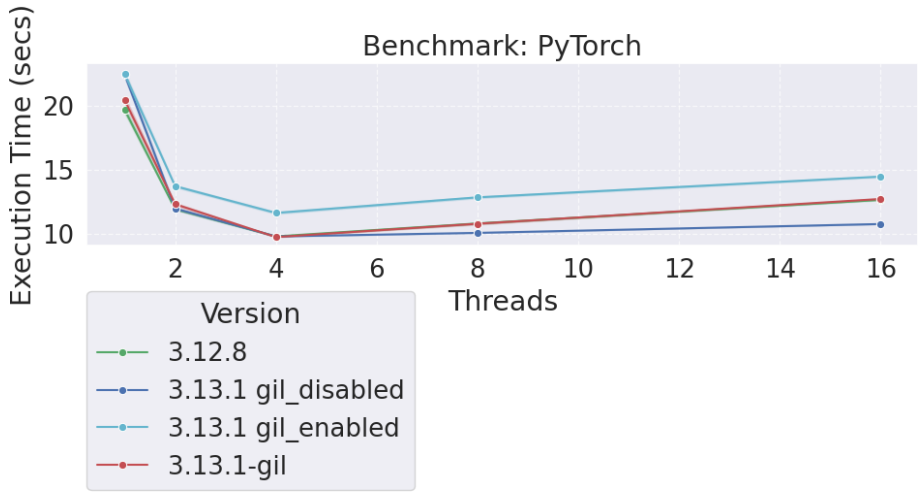


Figure 10: Runtime in seconds for the PyTorch benchmark.

The PyTorch benchmark results show that Python 3.12.8 and 3.13.1, whether the GIL is enabled or disabled, all behave similarly. The initial runtime with one thread is around 20–23 seconds across all versions. As more threads are added, performance improves, reaching the best execution time at four threads, hovering around 10 seconds. Performance only slightly worsens with more threads each time, but remains within a small range of deviation. The performance decrease is the smallest with the GIL disabled.

Originally, it was expected that the performance would remain the same regardless of thread count since PyTorch primarily offloads computations to the GPU. However, the observed improvement with more threads suggests that a single model did not fully utilize the GPU and could handle multiple concurrent workloads. This contradicts the hypothesis from the paper "Using Python for Model Inference in Deep Learning"[4], which suggested that the GIL might be a bottleneck for AI workloads. The different results might be due to different hardware, as DeVito et. al. were using a multi-GPU setup. The results here indicate that PyTorch already works efficiently with multiple threads, making GIL removal largely irrelevant in this specific benchmark.

6 Conclusion

6.1 Summary of Findings

The benchmarks we conducted in this study show a detailed analysis of Python 3.13 without the GIL. The results confirm that removing the GIL can substantially improve certain workloads, but it is not always beneficial.

For CPU-bound multi-threaded workloads, such as dictionary operations and compression, removing the GIL resulted in substantial speedups, with execution times improving by up to 70% when using multiple threads. This shows that the GIL is a bottleneck for these parallel tasks, and removing it allows the programs to utilize modern processors fully.

However, some workloads saw no meaningful improvement or even performance degradation. The Numpy and Pandas benchmarks demonstrated that these libraries, which already work around the GIL, did not benefit from GIL-less Python. In fact, Python 3.13, with the new threading model, introduced overhead leading to performance regressions, especially in single-threaded execution. This suggests that some optimization still needs to be done for these libraries.

The string benchmark revealed severe performance issues with Python 3.13's new threading model. These benchmarks saw severe slowdown with single-threaded performance. Even with multiple threads, execution times remained slower than in Python 3.12.8. This suggests that the new memory model may have critical inefficiencies or bugs.

Overall, while the removal of the GIL offers substantial performance benefits for specific multi-threaded workloads, it also introduces overhead and regressions in others. Adoption will require careful consideration depending on the workload and existing library dependencies.

6.2 Future Work and Outlook

Future work on this topic could investigate why workloads such as NumPy and Pandas suffer in Python 3.13. Further testing in the future could also test whether the overhead of the new threading model gets reduced.

Lastly, future work should also examine real-world applications beyond micro benchmarks. Web frameworks, data pipelines, and more testing for AI workloads could provide practical insights into how the new threading model performs under actual production conditions.

Additionally, Python 3.14 introduces a new interpreter design that uses tail calls between small C functions implementing individual Python opcodes, resulting in performance improvements of up to 30% in certain scenarios, with a geometric mean speedup of 3-5% on the pyperformance benchmark suite[9]. Future research should benchmark these improvements against Python 3.13 to assess whether they alleviate the overheads introduced by the GIL-less execution model and provide meaningful speedups for multi-threaded workloads.

References

[1] 2025. URL: https://docs.python.org/3.13/library/multiprocessing.html.

[2] David Beazley. "Understanding the python gil". In: *PyCON Python Conference. Atlanta, Georgia*. 2010, pp. 1–62.

[3] Carl Friedrich Bolz et al. "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICOOOLPS '09. Genova, Italy: Association for Computing Machinery, 2009, pp. 18–25. ISBN: 9781605585413. DOI: 10.1145/1565824.1565827. URL: https://doi.org/10.1145/1565824.1565827.

[4] Zachary DeVito et al. *Using Python for Model Inference in Deep Learning*. 2021. arXiv: 2104.00254 [cs.LG]. URL: https://arxiv.org/abs/2104.00254.

[5] *GlobalInterpreterLock*. 2020. URL: https://wiki.python.org/moin/GlobalInterpreterLock (visited on 03/01/2025).

[6] Eoin Malins. *Author*. 2019. URL: https://www.blopig.com/blog/2019/01/making-the-most-of-your-cpus-when-using-python/ (visited on 03/27/2025).

[7] Dorian Ouakli. "Native Implementation of OpenMP For Python". Available at https://fosdem.org/2025/events/attachments/fosdem-2025-4034-multithreading-in-python-using-openmp-/slides/238770/OUAKLI_TF_ltmi5ei.pdf. MA thesis. Santiago de Compostela, Universidade de Santiago de Compostela, July 2024.

[8] *PEP 703 – Making the Global Interpreter Lock Optional in CPython | peps.python.org*. en. URL: https://peps.python.org/pep-0703/ (visited on 03/28/2025).

[9] *What's new in Python 3.14*. en. URL: https://docs.python.org/3/whatsnew/3.14.html (visited on 03/27/2025).