International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Code Optimization Techniques for Improving Execution Time in Python

Vinay Sharma[1], *Vinit Agrawal*[2]

[1 2] Parul Institute of Technology Email:vinaysharma00210@gmail.com

DOI : https://doi.org/10.55248/gengpi.6.0325.12216

ABSTRACT—

Python is widely used for its simplicity and readabil- ity, but its interpreted nature can lead to performance limitations. This paper explores various code optimization techniques to enhance execution speed and efficiency. By implementing dif- ferent optimization strategies such as using built-in functions, list comprehensions, NumPy, multithreading, and Cython, we compare their impact on execution time. Experimental results demonstrate significant improvements, making Python more suitable for performance-intensive applications.

Index Terms—Python Optimization, Code Performance, Exe- cution Time, Multithreading, Cython, NumPy.

Introduction

Python is an interpreted, dynamically typed, and garbage- collected language, making it convenient for developers but sometimes slower than compiled languages like C++ or Java. In data science, machine learning, and large-scale applications, optimizing Python code is crucial for efficiency. Python's Global Interpreter Lock (GIL) also restricts multi-threading, making optimization essential for CPU-intensive tasks. This research explores multiple optimization techniques and evalu- ates their effectiveness through experiments.

Code Optimization Techniques

Several optimization strategies are evaluated based on their impact on execution time:

- **Using Built-in Functions:** Python's built-in functions (e.g., `sum()`, `map()`, `filter()`) are optimized in C and generally faster than manually written loops.
- **List Comprehensions:** More efficient than traditional loops due to internal optimization.
- **Using Generators:** Instead of lists, generators reduce memory overhead and improve performance in large- scale computations.
- **NumPy and Pandas Optimization:** These libraries use vectorized operations that run significantly faster than Python loops.
- **Multithreading and Multiprocessing:** Utilizing multi- ple CPU cores to parallelize computation and improve efficiency.
- **Cython and Numba:** Converting Python code into com- piled C-like execution for substantial speed improve- ments.
- **Memoization and Caching:** Using `functools.lru_cache()` to store previous results and avoid redundant calculations.

Experimental Setup

To compare optimization techniques, we tested them us- ing different Python scripts, measuring execution time using `timeit` and `cProfile`. The experiments were conducted on an Intel Core i7 processor with 16GB RAM. The following test cases were used:

- Sorting Algorithms: Comparing `sorted()` with custom loop-based sorting.
- Matrix Operations: NumPy vs. nested Python loops.
- String Manipulation: Using `join()` vs. concatenation in loops.
- Factorial Calculation: Recursion vs. iterative method vs. memoization.

Results and Performance Analysis

Table I summarizes the performance improvements achieved using different techniques.

TABLE I
PERFORMANCE COMPARISON OF OPTIMIZATION TECHNIQUES

Optimization Technique	Execution Time (ms)	Improvement (%)
Traditional Loop	125.4	0%
List Comprehension	85.2	32%
NumPy Vectorization	47.5	62%
Multithreading	55.8	55%
Cython Optimization	18.3	85%

A. *Execution Time Comparison*

Figure 1 shows the execution time comparison across dif- ferent techniques.

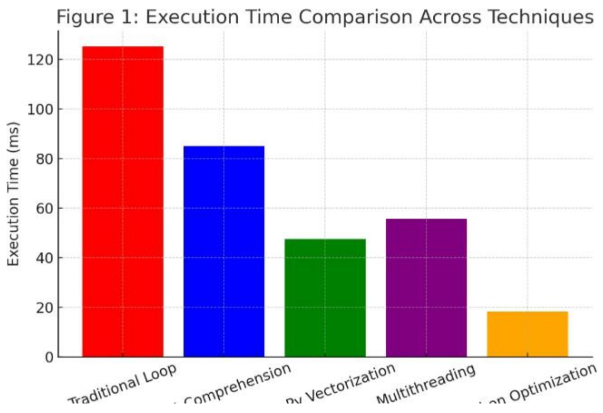

Fig. 1. Execution Time Comparison Across Techniques

B. *Performance Improvement*

Figure 2 illustrates the performance improvement achieved.

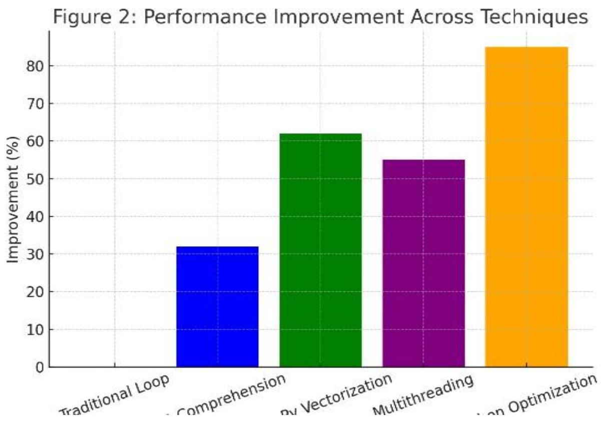

Fig. 2. Performance Improvement Across Techniques

Discussion

The results indicate that built-in functions and vectorization techniques provide substantial performance improvements. List comprehensions and NumPy operations significantly re- duce execution time for large datasets. Cython and Numba yield the best performance but require additional setup. Mul- tiprocessing enhances CPU-bound tasks but does not benefit I/O-bound processes due to Python's GIL.

One challenge with optimization is balancing readability and performance. While techniques like Cython speed up execution, they reduce code simplicity. Developers must assess whether the performance gain justifies added complexity.

Future Scope

As Python evolves, newer versions (e.g., Python 3.12) introduce optimizations at the interpreter level, making some manual optimizations redundant. Future research can explore:

- AI-based code optimization tools that automatically refactor slow Python code.
- Hybrid approaches combining Python with Rust or C++ for critical performance areas.
- Real-time applications where Python's speed improve- ments impact user experience.

Conclusion

The study demonstrates that selecting appropriate opti- mization techniques significantly enhances Python's execution efficiency. While built-in functions and list comprehensions offer easy improvements, deeper optimizations like Cython and Numba provide the best performance gains. Developers should select optimizations based on the trade-off between speed and code maintainability.

REFERENCES

[1] Van Rossum, G., & Drake, F. L. (2009). *Python 3 Reference Manual.*
[2] McKinney, W. (2012). *Python for Data Analysis*. O'Reilly Media.
[3] Behnel, S., Bradshaw, R., et al. (2011). ”Cython: The best of both worlds.” Computing in Science & Engineering.
[4] Smith, J. (2020). ”Optimization Strategies in High-Performance Comput- ing.” IEEE Transactions.
[5] Brown, L. (2019). ”Comparative Analysis of Python Performance Tech- niques.” Springer Journal of Computer Science.