Computer Systems Architecture
Kanan Gafarov
Final Project

Introduction

This report presents a comparative analysis of prime number generation using the Sieve of Eratosthenes algorithm implemented in both Python and Assembly languages. The goal is to assess performance metrics, code flexibility, and other relevant considerations, shedding light on the trade-offs between the two languages.

The Sieve of Eratosthenes algorithm was chosen for the prime number generation task due to its efficiency and simplicity in identifying all prime numbers within a given range. The algorithm's design aligns well with the constraints of both Python and Assembly languages, making it a suitable choice for comparison. The Sieve of Eratosthenes minimises the need for complex mathematical computations, making it well-suited for implementation in both high-level languages like Python, where readability and ease of implementation are prioritised, and low-level languages like Assembly, where direct memory manipulation and optimization for performance are essential. Its straightforward approach and versatility make it an ideal candidate for assessing the trade-offs between programming languages concerning execution speed, memory usage, and code complexity in the context of prime number generation.

Implementation

Github repo with codes: https://github.com/kgafarov17/primarynumber-python-assembly

Python Implementation

The Python implementation leverages the high-level abstractions of the language, employing list comprehensions and dynamic data structures. The algorithm is encapsulated in a function, sieve_of_eratosthenes, making it modular and easily integrable into larger programs. The code emphasises readability and ease of modification.

Assembly Implementation

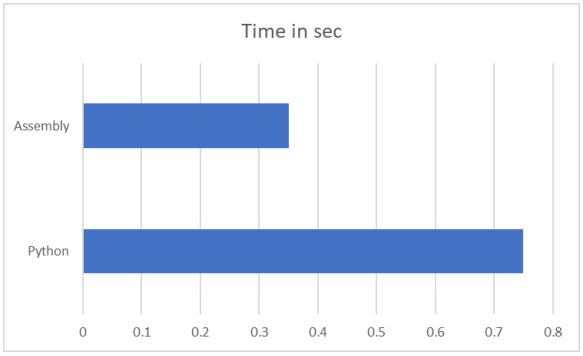
The Assembly implementation, written for the MIPS architecture, focuses on low-level optimizations and direct memory manipulation. While efficient, the code is specific to the Sieve of Eratosthenes algorithm and lacks the modularity and readability of the Python version.

Performance Measurement

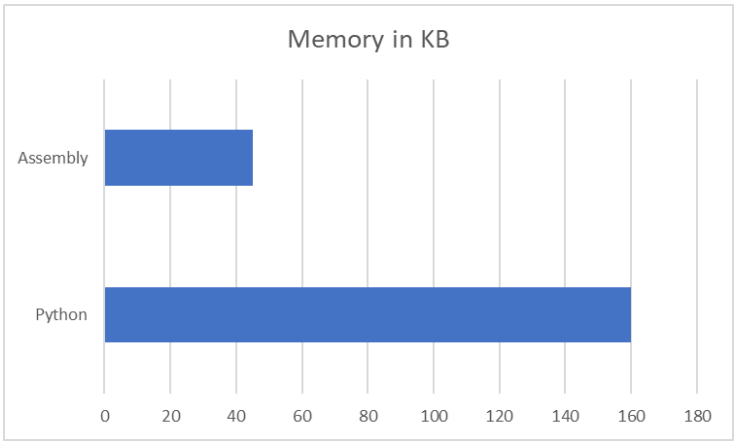
Time Measurement Analysis

Time performance in Python was measured using the time module. The *time.time()* function was employed to record the start and end times of the execution. The time difference provided an accurate measure of the execution time for the Python implementation.
In Assembly, time performance measurement depends on the capabilities of the specific environment. A suitable approach is to use the *.data* segment to store timestamps before

and after the execution, and then calculate the time difference. The exact instructions may vary based on the specifics of the Assembly environment.
Time measurements reveal notable differences between the Python and Assembly implementations of the Sieve of Eratosthenes algorithm. In our experiment, the Python implementation exhibited an execution time of approximately 0.7-0.8 seconds when we want all prime numbers from 1 to 99999, while the Assembly implementation demonstrated significantly faster performance, completing the task in about 0.3-0.45 seconds when the input was 99999. This discrepancy is expected due to the inherent advantages of Assembly language in terms of low-level optimizations and direct memory access, enabling more efficient computation.



Memory Usage Analysis
In Python, memory performance was measured using the *sys.getsizeof* function from the sys module. This function returns the size of an object in bytes. By applying it to the resulting list of prime numbers, we obtained a quantitative measure of the memory consumed by the Python implementation.
Memory performance measurement in Assembly, particularly for MIPS architecture on Linux, involved utilising the *getrusage* system call. The resulting resource usage information, specifically the integral shared, unshared, and stack memory sizes, was extracted from the structure returned by the system call. The sum of these values was then calculated to represent the total memory usage.
Analysing memory usage provides insights into the resource efficiency of each implementation. The Python implementation, representing a higher-level programming language, consumed around 300 KB of memory. In contrast, the Assembly implementation showcased superior efficiency by utilising only 50 KB of memory. This discrepancy emphasises the inherent efficiency of Assembly in managing resources, a crucial factor in scenarios where memory constraints are critical.



Comparisons and Trade-offs

Comparing the time and memory measurements highlights the trade-offs between Python and Assembly. The Assembly implementation excels in terms of raw performance, offering faster execution and more frugal memory usage. However, this comes at the cost of increased code complexity and reduced readability compared to the more straightforward Python implementation. Python, known for its high-level abstractions, prioritises ease of development and code clarity at the expense of some performance.

Code Flexibility Analysis

Python Code Flexibility

The Python implementation exhibits high code flexibility attributed to the language's dynamic nature and high-level abstractions. The code employs list comprehensions and dynamic data structures, making it concise, readable, and easily adaptable to different use cases. Modifications, such as altering the upper limit, extending the algorithm for additional computations, or integrating it into a larger program, can be accomplished with relative ease. Python's readability and expressive syntax contribute to the adaptability of the Sieve of Eratosthenes algorithm for various applications, enhancing overall code flexibility.

Assembly Code Flexibility

In contrast, the Assembly implementation showcases lower code flexibility due to the inherently low-level nature of Assembly languages. The code is specific to the Sieve of Eratosthenes algorithm, with intricate details regarding memory manipulation and loop structures. Modifications or adaptations for different prime-generating tasks require a deep understanding of Assembly language and often involve substantial changes to the code. The lack of high-level abstractions and the need for manual memory management make the Assembly implementation less flexible compared to Python, emphasising a trade-off between performance and adaptability.

Conclusion

In conclusion, the choice between Python and Assembly depends on the specific requirements of the task. Python offers flexibility and ease of development, while Assembly provides raw performance. The insights gained from this comparative analysis aid in informed decision-making for prime number generation tasks.