Proceedings of 2nd International Conference on Sustainable Computing and Smart Systems (ICSCSS 2024)
IEEE Xplore Part Number: CFP24DJ3-ART; ISBN: 979-8-3503-7999-0

Performance and Metrics Analysis Between Python3 v/s Mojo

Anuj Kumar Aditya Deo
Amity Institute of Information Technology
Amity University Jharkhand
Ranchi, India
anujk866942@gmail.com

Swayam Gupta
Amity Institute of Information Technology
Amity University Jharkhand
Ranchi, India
swayamgupta5698@gmail.com

Roumo Kundu
Amity Institute of Information Technology
Amity University Jharkhand
Ranchi, India
rjroumo@gmail.com

Piyush Jaiswal
Amity Institute of Information Technology
Amity University Jharkhand
Ranchi, India
piyushjaiswal2203@gmail.com

Taha Fatma
Amity Institute of Information Technology
Amity University Jharkhand
Ranchi, India
tahafatma3535@gmail.com

Mohan Kumar Dehury
Amity Institute of Information Technology
Amity University Jharkhand
Ranchi, India
mohankdehury@gmail.com

2024 2nd International Conference on Sustainable Computing and Smart Systems (ICSCSS) | 979-8-3503-7999-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICSCSS60660.2024.10625342

***Abstract* — In the field of programming languages, Mojo and Python have gained significant popularity and recognition among developers. While Mojo, a newly emerging language, and Python, a well-established language, both have their own merits and features that make them suitable for various programming tasks. Recent AI techniques such as transformers in Natural Language Processing (NLP), Reinforcement Learning (RL), and Generative Adversarial Networks (GANs) have shown remarkable advancements. However, these techniques face challenges like high computational costs, scalability issues, and integration complexity. While Python features user-friendly syntax and extensive libraries, its interpreted nature can hinder performance for computationally intensive tasks. This research addresses this limitation by introducing Mojo, a high-performance language specifically designed for AI applications. Mojo leverages compilation and advanced optimization techniques to achieve significantly faster execution speeds compared to Python. The Mojo programming language can address these challenges by offering high-performance computation, efficient memory management, and seamless integration with AI frameworks. This can lead to faster processing times, better scalability, and more streamlined development workflows for advanced AI systems. The paper presents empirical evidence demonstrating the substantial performance gains offered by Mojo. Furthermore, the analysis explores several advantages of Mojo beyond raw speed. These include static typing, which enhances code reliability and maintainability, and built-in support for parallelism, enabling efficient utilization of multi-core processors. Additionally, Mojo's seamless integration with existing Python codebases allows developers to leverage the extensive Python ecosystem while enjoying the performance benefits of Mojo.**

Keywords* — *Mojo, Python, Programming Languages, Performance, AI Applications, Machine Learning, Static Typing, Memory Safety, Multithreading

I. INTRODUCTION

Artificial Intelligence (AI) is the simulation of human intelligence processes by machines, particularly computer systems. It is important in the current world because it enhances efficiency, drives innovation, and enables solutions to complex problems across various sectors, including healthcare, finance, and transportation. As depicted in Fig 1. This growth is demonstrated globally, with a surge of interest evident across numerous countries [1, 50]. This widespread adoption is further illustrated by Fig 2, which depicts a clear upward trend in the use of the term "AI" over time [29]. Python, with its readily readable syntax and extensive ecosystem of libraries like TensorFlow and PyTorch, has become the de facto language for implementing AI and machine learning (ML) models [45, 56]. However, Python's interpreted nature often leads to performance bottlenecks, hindering the efficiency of complex AI applications. This is particularly true for computationally intensive tasks such as deep learning and large-scale data processing [28]. Additionally, Python lacks features crucial for robust software development, such as static typing and strong memory management [34]. To address these limitations, a new language specifically designed for high-performance AI development has emerged: Mojo [26].

Fig 1. Popularity of AI Across Countries

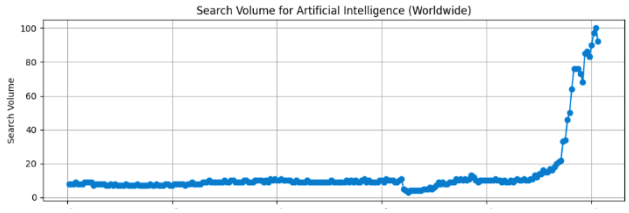

Fig 2. Search Volume For Term AI Over The Years

Authorized licensed use limited to: AMITY University. Downloaded on September 09,2024 at 16:51:47 UTC from IEEE Xplore. Restrictions apply.

Developed by Chris Lattner, the creator of the popular programming language Swift, Mojo offers a compelling alternative to Python for AI and ML projects [33]. Unlike Python, The Mojo Programming Language, designed for high-performance AI and data science applications, can revolutionize the AI landscape by enabling faster computation, more efficient data handling, and seamless integration with existing AI frameworks, thus accelerating AI development and deployment. Moreover, Mojo boasts several features that enhance the development experience for AI applications. Static typing ensures type safety, preventing runtime errors often encountered in dynamically typed languages like Python [24]. This leads to more robust and maintainable code. Additionally, Mojo offers strong memory management, mitigating the risk of memory leaks and crashes that can plague Python programs [32]. Finally, Mojo supports multithreading, enabling developers to leverage multiple cores or processors for parallelized execution, further boosting performance for computationally intensive tasks [48]. The motive and objective laid down by the amalgamation of the work is to facilitate and validate the features and plethora of the new " Gen-Z " Language Mojo in terms of building "*AI and ML Models Faster"* with better efficiency.

II. LITERATURE REVIEW

Python and C++, two well-known programming languages, are compared in this essay. Operating systems employ C++, a low-level object-oriented language, in contrast to Python, which is a high-level object-oriented language. Memory management strategies, program execution speed, execution time, and memory used by several algorithms in both languages are all analyzed. Data suggests that C++ is faster at execution, but because it's easier to use, Python is a better choice for novices. The study also examines how much time is spent on each language. Although Python interpreters need some time to convert human-readable code into bytecode and machine code, C++ is a statically typed compiled language. Python uses dynamic typing, which quickly resolves type conversion problems that arise during runtime. The sorting, searching, insertion, and deletion algorithms on an array data structure formed the foundation for a comparison between the two languages. The results of the investigation showed that in terms of memory and performance, Python is inferior to C++. Nonetheless, C++ is a better option for bigger projects, while Python is a fantastic tool for beginners and those looking to code quickly. In conclusion, each of Python and C++ has benefits of its own. Python is a better choice for beginners than C++ because of its ease of use, readability, portability, and scalability. C++ is a difficult language to master [10].

The polyhedral loop nest optimizer for the LLVM project, Polly, has suggested user-directed pragmas for loop modification. An MC tree search (MCTS) search method is built to determine the optimal loop optimization strategy. The technique is divided into two stages: first, it goes deep to exploit the loop transformations on the various tree levels, and then it solves the problem by local search process [2,3,4]. In a local solution, a restart mechanism keeps the MCTS from becoming blocked. Experimental results manifest that the MCTS (Monte-Carlo Tree Search) algorithm finds pragma combinations with a speedup of 2.3x over Polly's heuristic optimizations. To correctly find high-performing loop transformation combinations, an autotuning framework was created. This suggested MCTS outperforms 16 PolyBench kernels and three ECP proxy apps' heuristic optimization of Polly. In addition, we want to include the remaining loop transformation that Polly can utilize and reduce the search space by removing transformations that are not viable and identical configurations [11].

The studies that have been described so far have focused on identifying the most painful and descriptive aspects of languages like Java, C, C++, C, Rust, and JavaScript. This effort is distinctive enough to compare the new Language Mojo's design, features, and performance against Python.

III. PYTHON ARCHITECTURE

Since the late 1980s, Guido Van Rossum has been developing Python, a high-level programming language [44]. It was created to take the position of the ABC programming language, which was compatible with the Amoeba OS [38]. Version 1.0 of Python was initially made available in 1994. Versions 2.0 and 3.0 were then published in 2000 and 2008, respectively. The Python interpreter, called CPython, is written in the C programming language and works behind the scenes [16].

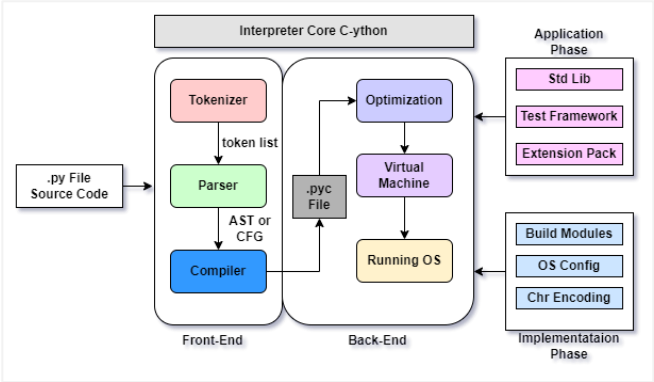

Fig 3. Python's Interpreter Working Mechanism Architecture

Architectural Breakdown

The CPython implementation shown in Fig 3, the most widely used, employs a multi-stage compiler architecture. Here's a breakdown of its key components:

- Tokenizer: The source code is broken down into basic components known as tokens in this first step. Tokens such as def, if, operators (+, -), identifiers like variable names, and delimiters like brackets and commas are represented by these tokens [54].

- Parser: The parser takes the token stream and constructs an Abstract Syntax Tree (AST). The AST serves as a hierarchical representation of the program's structure, capturing the relationships between code elements [30].

- Bytecode Generation: Traversing the AST, this stage translates it into bytecode, a low-level instruction set specifically designed for the Python Virtual Machine

Authorized licensed use limited to: AMITY University. Downloaded on September 09,2024 at 16:51:47 UTC from IEEE Xplore. Restrictions apply.

(PVM). Each bytecode instruction corresponds to an operation the PVM can execute [52].

Java Bytecode Vs Python Bytecode

Java and Python are programming languages that use a bytecode as an intermediate representation for their programs [31]. There are several differences between Java bytecode and Python bytecode:

Platform Independence: Java bytecode is platform-independent, allowing it to run seamlessly on various platforms as long as a compatible JVM is present. On the other hand, Python bytecode lacks inherent platform independence, and its execution relies on the Python Virtual Machine (PVM), which is not a platform-independent abstraction layer [36].

Optimization Potential: Java bytecode offers greater optimization potential due to its Just-In-Time (JIT) compiler, which dynamically translates frequently executed bytecode sections into machine code during runtime. However, Python bytecode has less extensive optimization opportunities and is generally considered faster [49].

Development Workflow: Java Bytecode involves a two-stage compilation process, requiring stricter type checking, while Python Bytecode generates bytecode at runtime, offering a streamlined development workflow but potentially causing errors [27].

Optimization: Certain Python implementations incorporate optimization passes at this stage. These passes may analyze the bytecode to identify redundancies or inefficiencies, potentially improving the code's performance [27].

Bytecode Emission: Finally, the optimized bytecode is written to a .pyc file. This file serves as an intermediate representation and can be reused for subsequent executions, provided the source code hasn't changed [43].

IV. MOJO ARCHITECTURE

Mojo (mojo_test.🔥) , a multi-paradigm programming language, was created by Chris Lattner to bridge the gap between the research and production stages of AI development in May 2023 [46]. It offers syntax familiarity, unparalleled performance, Python interoperability, parallel processing, and model extensibility [47]. Mojo borrows heavily from Python's syntax, providing a smoother transition from research to production. It leverages the MLIR compiler infrastructure, allowing for the translation of Mojo code into highly optimized machine code. Mojo also allows for Python interoperability, enabling developers to leverage existing Python libraries within their code. Furthermore, Mojo allows for the modification and extension of existing AI models, providing a powerful tool for advanced AI development. Mojo uses the LLVM compiler infrastructure project for exceptional performance but instead uses MLIR (Multi-Level Intermediate Representation) as a bridge between Mojo's source code and the LLVM toolchain. MLIR offers language independence, rich optimization opportunities, and extensibility, allowing for efficient machine code tailored to specific hardware and domain-specific optimizations relevant to AI applications, enabling further performance enhancements. The next section discusses LLVM and MLIR models in detail.

LLVM

A key characteristic of LLVM is the LLVM Intermediate Representation (IR), a description of the code within the compiler. What is called an optimizer in a compiler is designed to be big enough to have mid-level analysis and transformations. These analyses and transformations will work under the optimizer. Along with its class oil, LLVM IR is said to provide the most aggressive restructure transformation, cross-function and/or interprocedural optimization, full program analysis, and runtime optimizations. Bases of the virtual instruction set such as addition, subtraction, comparison, and branch will perform adding sequentially as in low-level RISC processors. LLVM is tightly typed, hides architectural details under the hood, and is based on a simplified type system. Unlike many others, it has a very modular structure. Three isomorphic forms of LLVM IR are defined: Instead of the textual format, an in-memory data structure that may be examined and edited through optimization can be applied. Moreover, the bitcode format, which is much smaller and denser, is incorporated for the disc storage [6].

The on-disk format can be changed from text to binary using tools from the LLVM Project. Because it is intended to be both expressive enough to provide significant optimizations for actual targets and simple enough for a front end to create, an intermediate representation of a compiler can serve as the optimizer's "perfect world". A front end is used by an LLVM-based compiler to read, verify, and identify mistakes in input code before translating it into LLVM IR. After going through analysis and optimization stages to make the code better, this IR is fed into a code generator to create native machine code. LLVM IR is the sole well-defined interface to the optimizer and the source of power and flexibility for the LLVM architecture [7]. The three-phase architecture model is shown in figure 4. Its success in many applications can be attributed in large part to this feature. This feature is absent from the GCC compiler, though, because its GIMPLE mid-level representation is not self-contained. To create a GCC front end, front-end authors must thus be familiar with GIMPLE and the tree data structures of GCC. Furthermore, GCC is hard to experiment with since it doesn't allow you to dump out "everything representing my code" or read and write GIMPLE in text form. Instead of being a single command line compiler or opaque virtual machine, LLVM is an architecture built as a set of libraries. It is a framework that may be applied to specific tasks, such as developing a C compiler or refining an optimizer within a pipeline for special effects [41]. The optimizer receives LLVM IR, processes it for a while, and then outputs LLVM IR, which should result in speedier execution. The optimizer is set up as a series of discrete optimization runs, each of which is executed on the input and allowed to act. Expression reassociation, loop invariant code mobility, and the inliner are common instances of passes [42]. An indirect descendant of the Pass class, each LLVM pass is expressed as

Authorized licensed use limited to: AMITY University. Downloaded on September 09,2024 at 16:51:47 UTC from IEEE Xplore. Restrictions apply.

a C++ class. A single.cpp file contains the majority of passes, and an anonymous namespace defines their subclass of the Pass class. Numerous passes are available from the LLVM optimizer, and each one is compiled into one or more files that are then assembled into a collection of archive libraries (.a files on Unix systems). Because of the loose coupling between the passes, these libraries offer a variety of analysis and transformation capabilities. The implementation may select which passes are appropriate for the image processing domain as well as the sequence in which they run thanks to LLVM's library-based design [40, 55]. This uncomplicated design strategy enables LLVM to offer a great deal of functionality without penalizing library customers who only wish to do basic tasks. To generate the best code feasible for each target, the LLVM code generator converts LLVM IR into target-specific machine code. Instruction selection, register allocation, scheduling, code layout optimization, and assembly emission are some of the distinct stages that it divides the code generation problem into. The target author has the option to apply custom passes that are unique to their needs or select from these predefined passes. Target authors may develop excellent code thanks to this flexibility since it eliminates the need to completely create a code generator for their target.

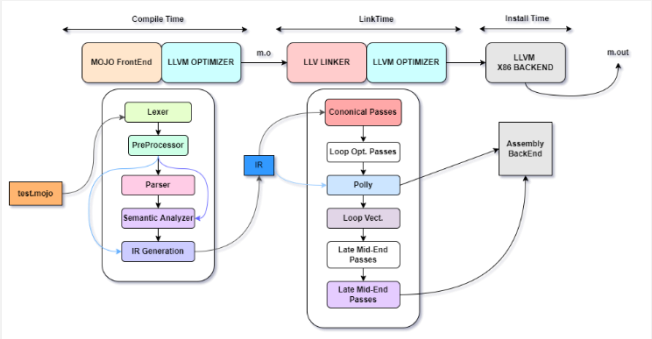

Fig 4. Comprehensive LLVM's Three-Phase Architecture Model

MLIR

Building reusable and extendable compiler infrastructure is being tackled in a new way with the MLIR project [39]. It attempts to alleviate the fragmentation of software, enhance compilation for heterogeneous hardware, lower the cost of developing domain-specific compilers, and provide interoperability amongst current compilers. The comprehensive model is shown in figure 5. Target-specific operations, polyhedral primitives, dataflow graphs, optimizations, loop optimizations, code generation "lowering" transformations, and hardware synthesis tools are just a few of the needs that MLIR, a hybrid IR, covers in a single infrastructure [5]. Despite being a sophisticated representation, MLIR does not enable source language writing for end users or low-level machine code-generating methods. Current best practices, such as creating and maintaining an IR standard, creating an IR verifier, and creating modular libraries, are encouraged by the MLIR framework. Additional modules, including restricting the scope of SSA to minimize use-def chains and substituting explicit symbol references for cross-function references, have been integrated into the design [40].

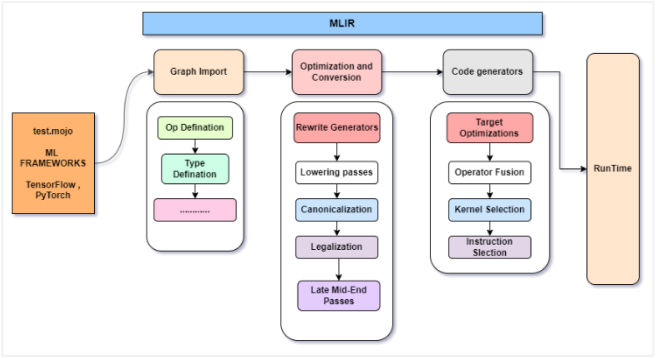

Fig 5. Comprehensive MLIR's Modular Phase Architecture Model as an Extension to the LLVM Model

V. PERFORMANCE ANALYSIS

As termed by the documentation of Modular and Mojo, which states that it supports high-performance speed. To prove justification for that statement comparative testing was done. The three most famous data structure-based algorithms were used for the performance analysis. Namely Linear Search, Insertion Sort, and Bubble Sort. Recursion-based algorithms were not tested as Python has a fixed maximum recursion depth. This means there's a limit of 1000 to how many times a function can call itself recursively before an error occurs [23]. To avoid reaching the maximum recursion depth in Python, it is important to optimize recursive functions and make sure they have a base case that terminates the recursion [22]. By optimizing recursive functions and ensuring a base case, you can prevent the program from reaching the maximum recursion depth in Python. Additionally, it is possible to customize the recursion depth limit in Python using the sys module.

The architectures considered for the task are Google Colab (a), Windows 10 (b), and Modular Playground (c) described in Table 1.

TABLE 1. TESTING PLATFORM'S ARCHITECTURE SPECIFICATIONS

a	CPU Model: x86_64 Operating System: Linux-6.1.58+-x86_64-with-glibc2.35 Physical Cores: 1 Logical Cores: 2 RAM: 12.67 GB Architecture: 64bit
b	CPU Model: AMD64 Family 23 Model 113 Stepping 0, AuthenticAMD Operating System: Windows-10-10.0.19045-SP0 Physical Cores: 6 Logical Cores: 12 RAM: 15.93 GB Architecture: 64bit
c	OS : linux CPU : cascadelake

Authorized licensed use limited to: AMITY University. Downloaded on September 09,2024 at 16:51:47 UTC from IEEE Xplore. Restrictions apply.

	Arch : x86_64-unknown-linux-gnu Physical Cores : 4 Logical Cores : 8 CPU Features : sse4 avx avx2 avx512f avx512_vnni

The information on the Modular Playground has been obtained from the official documentation which states " With 3rd Generation Intel Xeon Scalable processors powering them, Amazon EC2 C6i instances offer up to 15% better pricing performance than C5 instances for a variety of applications. With a RAM-to-virtual CPU ratio of 2:1, they support up to 128 virtual instances per instance, which is an increase of 33% when compared to C5 instances. Examples of these instances include video encoding, ad serving, high-speed computing, distributed analytics, batch processing, and multiplayer gaming that require high computing power. Besides being ideal for low-latency, high-speed applications, they also feature local NVMe-based SSD block-level storage. In terms of cost per TB, the C6id instances are 56% cheaper than C5d instances and would have up to 138% more storage capacity per virtual CPU. The C6in versions provide for two times the packet performance of the C5n instances and up to 200 Gbps of network capacity. Besides, they can deliver 400K I/O operations per second (IOPS) and up to 100 Gbps of Amazon Elastic Block Store (EBS) performance which is perfect for applications with a huge network demand." [21]

The RAM specifications for Amazon EC2 C6i instances, with the text provided, about the RAM specifications for Amazon EC2 C6i instances:

- C6i instances feature a 2:1 vCPU to Memory ratio, as is the case for C5 instances.
- They may be running up to 128 virtual CPUs, which is an increment of 33% in comparison to C5 instances.

Based on this information, we can deduce that the RAM specifications for C6i instances depend on the number of vCPUs allocated - Based on this information, we can deduce that the RAM specifications for C6i instances depend on the number of vCPUs allocated:

- The 2:1 vCPU to M implies that for every two vCPUs, there is an M of memory. The calculating ratio is the same as that of C5 machines.
- Upto 128 vCPUs of processing capability is supported by C6i instances, the translation is that the memory is also up to 64 units (128 vCPU ÷ 2 memory units per vCPU = the value of A).

Though the text does not show the exact value of RAM per vCPU or instance in GB or TB, the given text shows the number of vCPUs or seeds that can use the same resources. It is the RAM allocation that will correspond with the instance type chosen from among the C6i family and with a specific configuration during the instance creation.

OS The performance of Programming languages varies with different variations of Operating Systems. Some operating systems are more optimized for certain programming languages, leading to better performance and efficiency [20]. The snippets of the algos and implementation of the codes of MOJO in the playground have been described here as follows.

```
from random import random_float64
from python.object import PythonObject
from collections import List
import benchmark

fn birth() -> None:
  #LIST CREATION
  var py_list: PythonObject = []
  try:
    for i in range(0, 100000):
      py_list.append(random_float64(0, 10))
  except:
      print("Meow")

fn main():
    birth()
    var report = benchmark.run[birth]()
    report.print()

```

Fig 6. Snippet for Building List in Mojo for Testing

```
fn test():
    # Insertion Sort
    var n: Int64 = len(arr)
    var temp: Float64 = 0
    for i in range(1, len(arr)):
        var key: Float64 = arr[i]
        var j: Int = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j = j - 1
            arr[j + 1] = key

```

Fig 7. Insertion Sort Algorithm Implementation in Mojo

```
fn test() :
    # BUBBLE SORT
      var n: Int64 = len(arr)
      var temp :Float64 = 0
      for i in range(0, n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                temp = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = temp

```

Fig 8. Bubble Sort Algorithm Implementation in Mojo

```
fn test() -> None:
    # LINEAR SEARCH
    try:
      for i in py_list:
        if(i == 2.135678):
          print("Found")
    except:
      print("Error")
```

Fig 9. Linear Search Algorithm Implementation in Mojo

As described in the code snippets in figures 6 to 9, the List Creation has been done with the data points of 10,0000 points

Authorized licensed use limited to: AMITY University. Downloaded on September 09,2024 at 16:51:47 UTC from IEEE Xplore. Restrictions apply.

the test analysis has been the creation of random 10,000 data points then implementing the algorithm as a whole.

Table 2. Performance Results Comparison of Different Algorithms with MOJO v/s Python3

Algo	Colab	Windows	PlayGround
Bubble Sort	17.0 mins	10.45 mins	2.2613 secs
Insertion Sort	7.50 mins	4.49 mins	21.010 secs
Linear Search	0.0118040 ms	0.0150 ms	0.8004 secs

The performance metrics displayed in Table 2 for each algorithm in all the environments give a wide range of execution times. Bubble Sort $O(n^2)$ [17], one of the basic algorithms for sorting, runs more quickly in PlayGround compared to Colab and Windows, possibly due to different resource allocation and optimization approaches. Besides the Insertion Sort $O(n^2)$ [18], an elementary sorting technique, illustrates that Windows and PlayGround are faster than Colab in terms of execution time as well. Compared to this, Linear Search O(n) [19], a very simple search algorithm operates very fast in Colab and Windows but is a little slower in PlayGround. Thus, the variations demonstrate the influence of system resources, optimization strategies, and environmental configurations on algorithms efficiency pointing out that these factors should be taken into consideration when comparing algorithms in various contexts.

VI. CONCLUSION

Finally, the study focused on the programming language landscape as a part of AI and machine learning exploration with a comparative analysis of Mojo and Python. AI development has adopted Python as a staple, due to its clear and high-level syntax as well as a wide range of libraries; however, some performance limitations, if applied to difficult calculating tasks, are evident. The characteristics of Mojo, which is designed for AI applications and involves features like compilation, static typing, memory safety, and built-in support for parallelism, have proven to be limitless. The study underlines the fundamental divergencies between Python and C++ concerning the storage taking in mind and the speed of work as well as highlighting the ease of use of Python for beginners and C++ for larger projects. It also touches on the architecture of Python which involves a multi-stage compiler and bytecode generation. In comparison, Mojo has an architecture that communicates with the MLIR compiler infrastructure for compiled machine code. The performance analysis piece tests the performance of algorithms against different platform environments, illustrating the implications that system resources, optimization strategies, and environment configurations have on the efficiency of a given algorithm. The outcomes emphasize the need for such areas to be factored in when selecting programming languages for particular projects. Overall, the research provides its context to the ongoing discussion of the listing of programming languages for AI development, which underlines their benefits and drawbacks and discovers Mojo as a groundbreaking supporter of performance in terms of fluency and compatibility with the existing Python codebase.

ACKNOWLEDGMENTS

This is to acknowledge the production and publication of the manuscript under the mandate of “Innovate Xplore” the Research and Development Club of “Amity Institute of Information and Technology (AIIT)” at “Amity University Jharkhand (AUJ)” in the year 2024 odd Semester.

REFERENCES

[1] "Google Trends".https://trends.google.com/trends/explore?date=all&q=%2Fm%2F0mkz&hl=en

[2] "docs.modular".https://docs.modular.com/mojo/why-mojo#why-we-chose-python

[3] "python-problems".https://docs.modular.com/mojo/why-mojo#pythons-problems

[4] "python-supersets-with-c-compatibility".https://docs.modular.com/mojo/why-mojo#python-supersets-with-c-compatibility

[5] "why-mojo#mlir".https://docs.modular.com/mojo/why-mojo#mlir

[6] "aosabook".https://aosabook.org/en/v1/llvm.html

[7] "MLIR".https://mlir.llvm.org/

[8] "mojo-speed-over-python".https://www.modular.com/blog/how-mojo-gets-a-35-000x-speedup-over-python-part-1

[9] "Matrix multiplication with Mojo".https://docs.modular.com/mojo/notebooks/Matmul

[10] Zehra, Farzeen & Javed, Maha & Khan, Darakhshan & Pasha, Maria. (2020). Comparative Analysis of C++ and Python in Terms of Memory and Time. 10.20944/preprints202012.0516.v1.

[11] J. Koo, P. Balaprakash, M. Kruse, X. Wu, P. Hovland and M. Hall, "Customized Monte Carlo Tree Search for LLVM/Polly's Composable Loop Optimization Transformations," 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), St. Louis, MO, USA, 2021, pp. 82-93, doi: 10.1109/PMBS54543.2021.00015.

[12] Antonios Tsigkanos , “A virtual machine and runtime framework targeting Heterogeneous embedded systems”

[13] Elias Athanasopoulos,” Lecture 18 Low Level Virtual Machine (LLVM)”

[14] S. P. Bejo, B. Kumar, P. Banerjee, P. Jha, A. N. Singh and M. K. Dehury, "Design, Analysis and Implementation of an Advanced Keylogger to Defend Cyber Threats," 2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, 2023, pp. 2269-2274, doi: 10.1109/ICACCS57279.2023.10112977.

[15] Al Sukhni, B. Kumar Mohanta, M. Kumar Dehury and A. Kumar Tripathy, "A Novel Approach for Detecting and Preventing Security attacks using Machine Learning in IoT," 2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT), Delhi, India, 2023, pp. 1-6, doi: 10.1109/ICCCNT56998.2023.10307883..

[16] Eberl, S. (2010, June). DGNB vs. LEED: A comparative analysis. In Conference on Central Europe towards Sustainable Building (pp. 1-5).

[17] Hsu, A. (2022, February 22). Comparison of Selection Sort, Insertion Sort and Bubble Sort. https://medium.com/coder-life/comparison-of-selection-sort-insertion-sort-and-bubble-sort-7dc1d971a136

[18] Insertion sort. (2023, January 23). https://www.wikiwand.com/en/Insertion_sort

[19] Linear search. (2001, September 27). https://en.wikipedia.org/wiki/Linear_search

[20] Bukie, P., Udeze, C., Obono, I., & Edim, E. (2019). Comparative Analysis of Compiler Performances and Program Efficiency. . https://doi.org/10.20944/preprints201909.0322.v1.

[21] “Amazon EC2 C6i Instances”. https://aws.amazon.com/ec2/instance-types/c6i/

Authorized licensed use limited to: AMITY University. Downloaded on September 09,2024 at 16:51:47 UTC from IEEE Xplore. Restrictions apply.

[22] Rajnis. (2019, August 5). sys.setrecursionlimit() method. https://www.geeksforgeeks.org/python-sys-setrecursionlimit-method

[23] What Is the Maximum Recursion Depth in Python. (2021, October 7). https://www.codingem.com/python-maximum-recursion-depth

[24] N. (2023, November 7). Mojo - A New Programming Language for AI - DEV Community. https://dev.to/refine/mojo-a-new-programming-language-for-ai-16n5

[25] N. (2023, November 7). Mojo - A New Programming Language for AI. https://dev.to/refine/mojo-a-new-programming-language-for-ai-16n5

[26] u00d6zmen, N. (2023, November 7). Mojo - A New Programming Language for AI. https://dev.to/refine/mojo-a-new-programming-language-for-ai-16n5

[27] Bennett, J. (2018, April 23). An introduction to Python bytecode. https://opensource.com/article/18/4/introduction-python-bytecode

[28] Byun, C., Arcand, W., Bestor, D., Bergeron, B., Gadepally, V., Houle, M., Hubbell, M., Hayden, J., Klein, J., Michaleas, A., Milechin, P., Morales, L., Mullen, G., Prout, J., Reuther, A., Rosa, A., Samsi, A., Yee, S., Kepner, C., & Jeremy, J. (2023, September 7). Python Performance Study. https://arxiv.org/abs/2309.03931

[29] Concepcion, R., Bedruz, R., Culaba, A., Dadios, E., & Pascua, A. (2019, November 1). The Technology Adoption and Governance of Artificial Intelligence in the Philippines. https://doi.org/10.1109/hnicem48295.2019.9072725

[30] Decompilation at Runtime and the Design of a Decompiler for Dynamic Interpreted Languages. (n.d). https://rocky.github.io/Deparsing-Paper.pdf

[31] Edge, J. (2013, April 3). PyCon: Peering in on bytecodes. https://lwn.net/Articles/544787/

[32] Fulton, N., Omar, C., & Aldrich, J. (2014, January 1). Statically typed string sanitation inside a python. https://doi.org/10.1145/2687148.2687152

[33] Gallagher, W. (2023, October 19). Swift creator brings new AI programming language to the Mac. https://appleinsider.com/articles/23/10/19/swift-creator-brings-new-ai-programming-language-to-the-mac?utm_medium=rss

[34] Geller, A. (2021, January 9). Is Python Really a Bottleneck?. https://towardsdatascience.com/is-python-really-a-bottleneck-786d063e2921?gi=5df1662aa2d5

[35] Hajkowicz, S., Sanderson, C., Karimi, S., Bratanova, A., & Naughtin, C. (2023, August 1). Artificial intelligence adoption in the physical sciences, natural sciences, life sciences, social sciences and the arts and humanities: A bibliometric analysis of research publications from 1960-2021. https://doi.org/10.1016/j.techsoc.2023.102260

[36] History of Python. (2009, February 1). https://en.wikipedia.org/wiki/History_of_Python

[37] Introduction to Mojo Programming Language. (2023, July 19). https://www.infoq.com/news/2023/07/mojo-programming-language/

[38] Ivanovs, A. (2023, October 14). The history of Python, the most popular programming language. https://stackdiary.com/the-history-of-python/

[39] Lattner, C., & Adve, V. (2005, January 1). The LLVM Compiler Framework and Infrastructure Tutorial. https://doi.org/10.1007/11532378_2

[40] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., & Zinenko, O. (2021, February 27). MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. https://research.google/pubs/pub49988/

[41] LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. (n.d). https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf

[42] optimization. (2011, May 1). https://blog.llvm.org/tags/optimization/

[43] PEP 488 – Elimination of PYO files. (2022, March 9). https://peps.python.org/pep-0488/

[44] Pramanick, S. (2019, May 2). History of Python. https://www.geeksforgeeks.org/history-of-python/

[45] Python is becoming the world's most popular coding language. (2018, July 26). https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language

[46] Savage, N. (2023, January 1). Revamping Python for an AI World. https://cacm.acm.org/magazines/2023/12/278143-revamping-python-for-an-ai-world/fulltext

[47] Savage, N. (2023, November 17). Revamping Python for an AI World. https://doi.org/10.1145/3624987

[48] Schoenborn, O. (2005, March 1). Resource Management in Python. https://www.drdobbs.com/web-development/resource-management-in-python/184405999

[49] Sewe, A., Mezini, M., Sarimbekov, A., Ansaloni, D., Binder, W., Ricci, N., & Guyer, S. (2012, June 15). new Scala() instance of Java. https://doi.org/10.1145/2258996.2259010

[50] Simeng, Z., Yizhou, Z., & Raj, V. (2020, August 20). Explore the Improvement of the Management of China's International Film Festivals Based on Artificial Intelligence. https://doi.org/10.1145/3407703.3407716

[51] Smith, D. (2023, November 19). How Mojo Hopes to Revamp Python for an AI World. https://developers.slashdot.org/story/23/11/18/2128233/how-mojo-hopes-to-revamp-python-for-an-ai-world?utm_source=rss1.0mainlinkanon&utm_medium=feed

[52] The bytecode interpreter. (2023, January 1). https://devguide.python.org/internals/interpreter/

[53] Wang, S., Huang, Y., & Wang, C. (2020, April 7). A Model of Consumer Perception and Behavioral Intention for AI Service. https://doi.org/10.1145/3396743.3396791

[54] Zehra, F., Javed, M., Khan, D., & Pasha, M. (2020, December 21). Comparative Analysis of C++ and Python in Terms of Memory and Time. https://doi.org/10.20944/preprints202012.0516.v1

[55] G. Savithri, B. K. Mohanta and M. Kumar Dehury, "A Brief Overview on Security Challenges and Protocols in Internet of Things Application," 2022 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS), Toronto, ON, Canada, 2022, pp. 1-7, doi: 10.1109/IEMTRONICS55184.2022.9795794.

[56] B. K. Mohanta, S. Chedup and M. K. Dehury, "Secure Trust Model Based on Blockchain for Internet of Things Enable Smart Agriculture," 2021 19th OITS International Conference on Information Technology (OCIT), Bhubaneswar, India, 2021, pp. 410-415, doi: 10.1109/OCIT53463.2021.00086.

Authorized licensed use limited to: AMITY University. Downloaded on September 09,2024 at 16:51:47 UTC from IEEE Xplore. Restrictions apply.