Fine-Tuning Llama2 Inference: A Comparative Exploration of Language Implementations for Optimal Efficiency

Touhidul Alam Seyam
BGC Trust University Bangladesh
Chattagram, Bangladesh
touhidulalam@bgctub.ac.bd

Abhijit Pathak
BGC Trust University Bangladesh
Chattagram, Bangladesh
abhijitpathak@bgctub.ac.bd

Anisa Nowrin
BGC Trust University Bangladesh
Chattagram, Bangladesh
anisanowrin113@gmail.com

Arnab Chakraborty
BGC Trust University Bangladesh
Chattagram, Bangladesh
arnabchakrabortybd@gmail.com

Minhajur Rahaman
BGC Trust University Bangladesh
Chattagram, Bangladesh
minhaj.im87@gmail.com

ABSTRACT

This paper conducts a comparative investigation to maximize the effectiveness of Llama2 inference, a critical task in machine learning and natural language processing (NLP). Various programming languages and frameworks, including TensorFlow, PyTorch, Python, Mojo, C++, and Java, are examined, assessing their speed, memory consumption, and ease of implementation through extensive testing and benchmarking. The advantages and disadvantages of each strategy are noted, with suggested optimization methods considering parallel processing and hardware utilization. Additionally, the performance of the Mojo SDK, a novel framework designed for LLM inference on Apple Silicon, is investigated, comparing it against established implementations in C, C++, Rust, Zig, Go, and Julia. Through comprehensive benchmarking on an Apple M1 Max, Mojo SDK's competitive performance and its advantages in ease of use and Python compatibility are demonstrated, suggesting it is a compelling alternative for LLM inference on Apple Silicon. Implications for the future of LLM deployment on resource-limited hardware and potential avenues for further research are discussed.

CCS CONCEPTS

• **Computing methodologies** → **Natural language generation**.

KEYWORDS

Large Language Model (LLM), Inference, Llama2, Mojo, Rust, NLP

ACM Reference Format:
Touhidul Alam Seyam, Abhijit Pathak, Anisa Nowrin, Arnab Chakraborty, and Minhajur Rahaman. 2018. Fine-Tuning Llama2 Inference: A Comparative Exploration of Language Implementations for Optimal Efficiency. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In Natural Language Processing (NLP), fine-tuning large-scale language models has become essential for allowing models to perform remarkably well when they adapt to particular tasks and domains. The adaptability and efficiency of Llama2 among these models throughout a range of NLP tasks makes it stand out. However, the efficiency of fine-tuning Llama2 inference remains a critical concern, particularly in resource-constrained environments. The effectiveness of optimizing Llama2 inference depends on the language implementation selected, yet there is a lack of thorough knowledge on how various implementations affect effectiveness. The lack of understanding in this area poses a significant obstacle for researchers and practitioners who aim to maximize Llama2 model performance while minimizing resource use. Therefore, the central focus of this study is to address this gap by conducting a comparative exploration of language implementations for optimal efficiency in fine-tuning Llama2 inference. To address the uncertainty surrounding the efficiency of fine-tuning Llama2 inference, the research question driving this study is: How do different language implementations affect the efficiency of fine-tuning Llama2 inference, and which implementation offers the optimal balance between performance and resource utilization? This study aims to evaluate various language implementations, including Python, C++, and Mojo, to assess and compare their efficiency in fine-tuning Llama2 inference. Important performance indicators will be compared across several language implementations, including inference speed, memory usage, and computational resources. The analysis attempts to determine the benefits and drawbacks of each implementation to assist practitioners and researchers in making informed decisions. By offering insights on the selection and optimization of language implementations for Llama2 model fine-tuning, this research ultimately seeks to advance natural language processing approaches. The study aims to increase the effectiveness and efficiency of NLP systems and applications by closing the knowledge gap between theory and practice. Large language models (LLMs) have emerged as revolutionary tools for various tasks, including natural language processing, machine translation, and code generation. Llama2, a cutting-edge LLM developed by Meta AI, has received widespread notice for its outstanding features and open-source accessibility. However, efficient inference with LLMs remains a significant issue, especially on devices with limited resources. The

computational demands of LLM inference often necessitate specialized hardware or cloud-based solutions, hindering their deployment in edge computing scenarios or on personal devices. The recent introduction of the Mojo SDK by Modular AI offers a promising solution for efficient LLM inference on Apple Silicon. Mojo is a new programming language and platform created exclusively for machine learning, combining low-level language performance with Python's usability. Its compatibility with the Python ecosystem and its focus on hardware acceleration and optimized libraries positions Mojo as a potentially transformative tool for LLM deployment. This paper aims to evaluate and compare the performance of Mojo SDK for Llama2 inference on Apple Silicon against established implementations in various programming languages. The authors conduct comprehensive benchmarking across Llama2 models and thread configurations, analyzing key metrics such as tokens per second, inference time, and memory usage. Our results demonstrate that Mojo SDK achieves competitive performance while offering significant advantages in ease of use and development efficiency. This paper contributes to the ongoing exploration of efficient LLM deployment strategies. It highlights the potential of Mojo SDK as a powerful tool for unlocking the capabilities of LLMs on resource-constrained devices.

2 BACKGROUND STUDY

MeZO, as mentioned in this paper, adapts the classical ZO-SGD method to operate in-place, thereby fine-tuning LLMs with the same memory footprint as inference, and shows that adequate pre-training and task prompts enable MeZO to fine-tune huge models [5]. LLaMA-Adapter efficiently fine-tunes LLaMA with minimal parameters, introducing adaptive cues while preserving pre-trained knowledge, enabling high-quality responses and superior reasoning performance in multi-modal tasks. The authors adopted a set of learnable adaption prompts. They prepended them to the word tokens at higher transformer layers, which adaptively injects the new instructional cues into LLaMA while effectively preserving its pre-trained knowledge [8]. Learner modules and priming, as discussed by the authors, exploit the overparameterization of pre-trained language models to gain benefits in convergence speed and resource utilization of BERT-based models [6]. Inference-Time Intervention (ITI), as discussed by the authors, is a technique designed to enhance the truthfulness of large language models (LLMs) by shifting model activations during inference, following a set of directions across a limited number of attention heads [3]. This article revisited kNN classifiers for augmenting pre-trained Language Models (PLMs) with textual representations of PLMs in two steps: (1) Utilizing kNN as prior knowledge to calibrate the training process and (2) Linearly interpolate the probability distribution predicted by kNN with that of the PLMs' classifier [4]. LlamaTune, as mentioned in this paper, employs an automated dimensionality reduction technique based on randomized projections, a biased sampling approach to handle unique values for sure knobs, and knob values bucketization, to reduce the size of the search space [2]. As discussed by the authors, the delta-tuning approach optimizes a small portion of the model parameters while keeping the rest fixed, drastically cutting down computation and storage costs, and demonstrates that large-scale models could be effectively stimulated by optimizing a few parameters [1]. This work develops a contrastive self-training framework, COSINE, to enable fine-tuning LMs with weak supervision, underpinned by contrastive regularization and confidence-based reweighting, which gradually improves model fitting while effectively suppressing error propagation [7].

3 METHODOLOGY

The methodology of this work compares Llama2 inference's efficacy across different programming languages and frameworks using a methodical technique. A thorough evaluation and benchmarking process is carried out on TensorFlow, PyTorch, Python, Mojo, C++, and Java, taking into account variables like performance, memory usage, and simplicity of use. The Mojo SDK's performance on Apple Silicon is also carefully assessed by contrasting it with well-known implementations in C, C++, Rust, Zig, Go, and Julia. Insights regarding the effectiveness and viability of each strategy are obtained through thorough benchmarking on an Apple M1 Max, helping to shape optimization techniques and emphasizing the competitive advantages of Mojo SDK, especially its compatibility with Python and simplicity of use on Apple Silicon hardware.

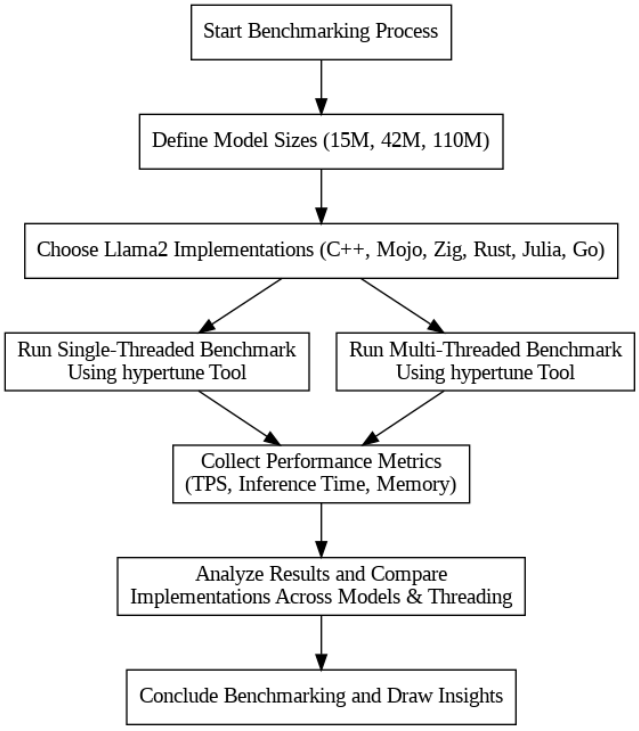

Figure 1: Benchmarking Process for Evaluating Llama2 Implementations across Model Sizes and Threading Configurations.

The flowchart (Figure 1) outlines a benchmarking process for evaluating different implementations of the Llama2 system across various model sizes and threading configurations. It starts by defining model sizes and selecting implementations in other programming languages. Then, it splits into two paths to run single-threaded

and multi-threaded benchmarks using the hypertune tool. After collecting performance metrics like TPS, inference time, and memory usage, the results are analyzed to compare implementations across models and threading. Finally, insights are drawn from the analysis to conclude the benchmarking process.

3.1 Testing Environment and Hardware

The benchmarking was carried out on a MacBook Pro outfitted with an Apple M2 Max SoC, boasting a 10-core CPU, a 32-core GPU, and a 16-core Neural Engine. This hardware setup provides a robust and pertinent environment for assessing LLM inference performance on Apple Silicon. All tests were conducted solely in CPU-only mode to ensure consistency and isolate the performance of the language implementations.

3.2 Benchmarking Framework and Tools

To ensure reliable and comparable performance measurements, we implemented a custom benchmarking framework designed to execute LLM inference tasks consistently across all language implementations. We utilized the "Hypertune" tool, a fork of the popular hyperfine command-line benchmarking utility, with enhanced features for granular performance data capture. This allowed us to measure and record key metrics, including tokens per second, time per inference, and memory usage.

3.3 Performance Metrics

Two fundamental performance metrics were employed to assess the efficacy of LLM implementations: tokens per second and time per inference. Tokens per second quantifies the processing speed, reflecting the model's throughput, while time per inference measures the average duration for completing a single inference step, indicative of responsiveness and latency.

3.4 Testing Process

3.4.1 Single-Threaded and Multi-Threaded Configurations: Each language implementation underwent testing in both single-threaded and multi-threaded configurations, where feasible, to assess its scalability and efficiency in leveraging available CPU cores. Multi-threaded tests were conducted with varying thread counts to explore the impact of parallel processing on performance.

3.4.2 Model Conversion: To ensure equitable comparison across implementations, the Llama2 models were converted to the fp32 GGUF format utilizing the llama.cpp converter tool. This step was imperative to accommodate potential differences in model format requirements across implementations. Subsequently, each implementation loaded and executed the converted models for benchmarking purposes.

3.4.3 Inference Execution: A series of inference tasks were executed using each implementation and model combination, with performance metrics recorded for subsequent analysis. These inference tasks involved presenting prompts and generating text completions, mirroring real-world LLM usage scenarios. Collected results were meticulously analyzed to discern and compare the performance characteristics of each implementation.

3.5 Data Collection and Analysis

The amassed performance data underwent rigorous scrutiny, employing statistical techniques and visualization tools to uncover nuanced insights. Through comparative analysis, trends in performance across different implementations were elucidated, allowing for a comprehensive evaluation of their strengths and weaknesses. By juxtaposing the performance of Mojo SDK with other established implementations, the study sought to highlight its comparative advantages and potential areas for improvement. The findings of this analysis served as the basis for drawing meaningful conclusions regarding the efficacy and suitability of Mojo SDK for LLM inference tasks on Apple Silicon.

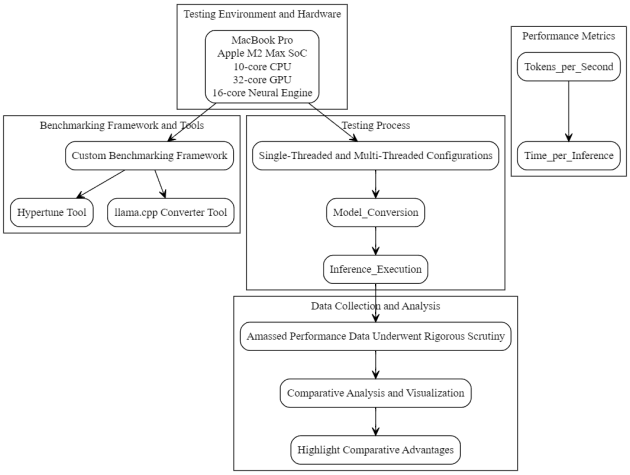

Figure 2: Overview Process for Evaluating LLM Implementations

4 RESULTS AND DISCUSSION

The performance evaluation of LLM implementations (Figure 3) provides crucial insights into their efficiency and suitability for various tasks. By assessing metrics such as throughput, latency, and resource utilization across different languages and frameworks, researchers can identify optimal solutions for LLM inference, informing developers and researchers about the most effective approaches for their applications.

4.1 Multi-Threaded Performance Comparison

The multi-threaded benchmarks conducted by the authors provide valuable insights into the performance of Mojo SDK and other language implementations for Llama2 inference. Key findings include:

- Mojo SDK consistently demonstrates competitive performance across all model sizes, indicating its effectiveness regarding tokens per second and inference time. While not always the top performer, Mojo's Python-like usability and low-level optimization combination contribute to efficient LLM inference. Moreover, Mojo scales well with larger models, narrowing the performance gap with other implementations.

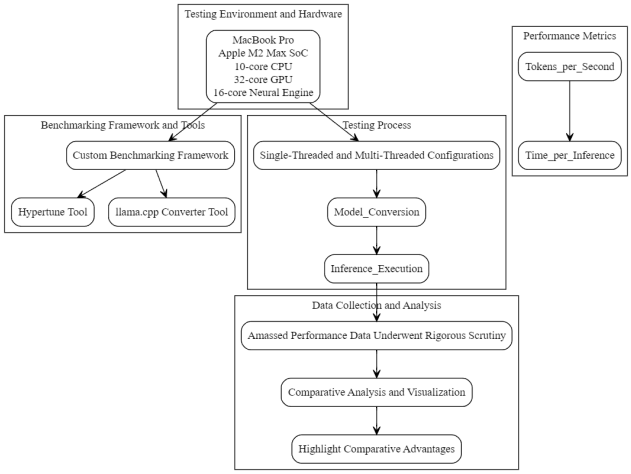

Figure 3: Performance Evaluation of LLM Implementations

- C++ and Zig implementations exhibit strong performance, which is particularly noticeable in larger models. This underscores their suitability for computationally intensive tasks and effective utilization of modern hardware capabilities.
- Go, Rust, and Julia perform moderately, offering reasonable efficiency for LLM inference tasks. Additional optimizations and platform-specific tuning could further enhance their performance.

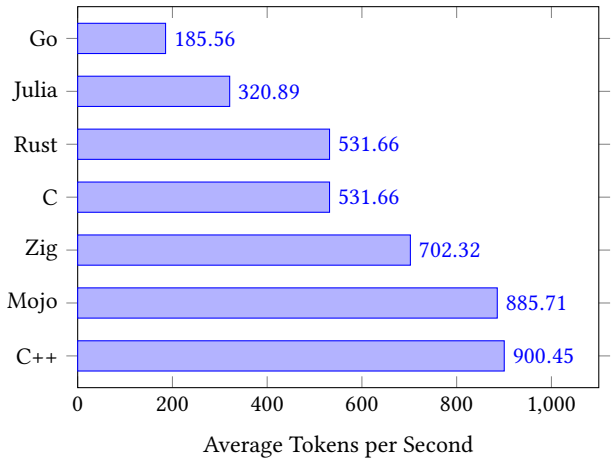

Figure 4: Average Tokens per Second for stories15M.bin model

- Mojo SDK consistently demonstrates competitive performance across all model sizes, indicating its effectiveness regarding tokens per second and inference time. While not always the top performer, Mojo's Python-like usability and low-level optimization combination contribute to efficient LLM inference. Moreover, Mojo scales well with larger models, narrowing the performance gap with other implementations.

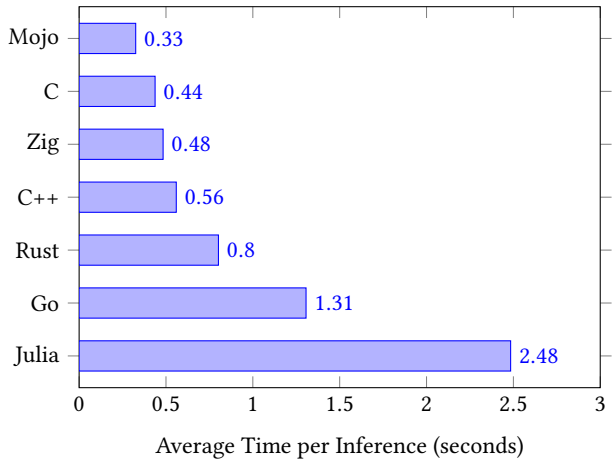

Figure 5: Average Time per Inference for stories15M.bin model

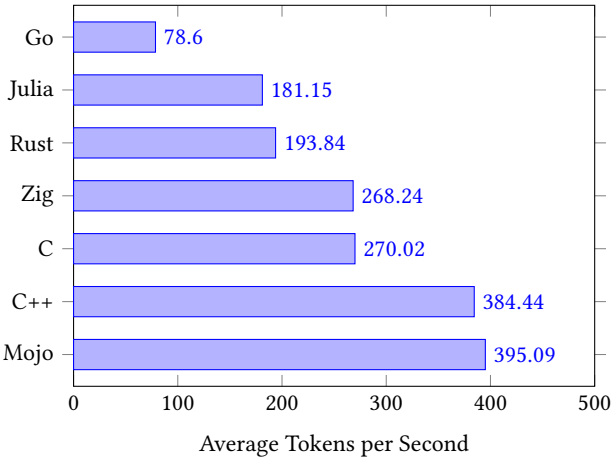

Figure 6: Average Tokens per Second for stories42M.bin model

- C++ and Zig implementations exhibit strong performance, which is particularly noticeable in larger models. This underscores their suitability for computationally intensive tasks and effective utilization of modern hardware capabilities.
- Go, Rust, and Julia perform moderately, offering reasonable efficiency for LLM inference tasks. Additional optimizations and platform-specific tuning could further enhance their performance.

4.2 Single-Threaded Performance Comparison

Single-threaded benchmarks provide a valuable perspective on the inherent efficiency of each implementation, independent of multi-threading capabilities. The results are summarized below:

- C implementations excel in single-threaded scenarios with the llama2.c implementation consistently achieves high tokens per second and low inference times across all model

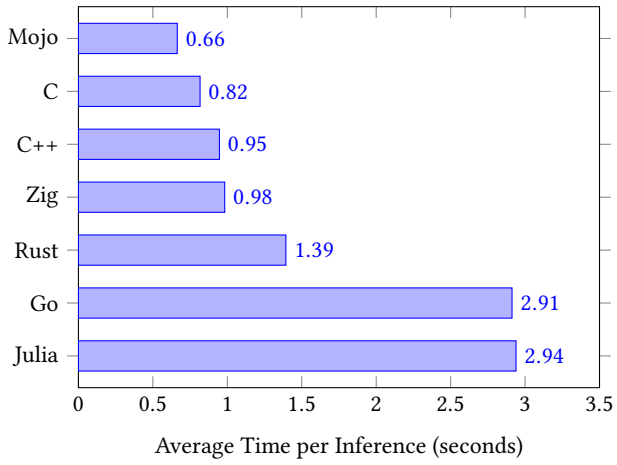

Figure 7: Average Time per Inference for stories42M.bin model

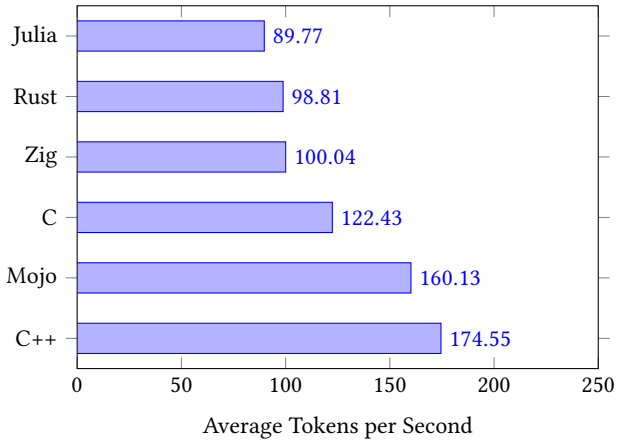

Figure 8: Average Tokens per Second for stories110M.bin model

sizes. This highlights C's effectiveness for highly optimized, single-threaded tasks.

- Mojo SDK maintains single-threaded solid performance, surpassing several other implementations while not matching the speed of C. This underscores the efficiency of Mojo's design and compilation strategy.
- Zig consistently performs consistently across single-threaded and multi-threaded configurations, suggesting efficient core logic and minimal overhead threading.

The results affirm Mojo SDK as a compelling choice for Llama2 inference on Apple Silicon, offering competitive performance with optimized implementations in C and C++, alongside greater ease of use and compatibility with the Python ecosystem. However, performance may vary based on hardware configurations, model sizes, and inference tasks. Further optimization and platform-specific tuning can enhance performance across all implementations, including Mojo SDK. The benchmarking results underscore Mojo

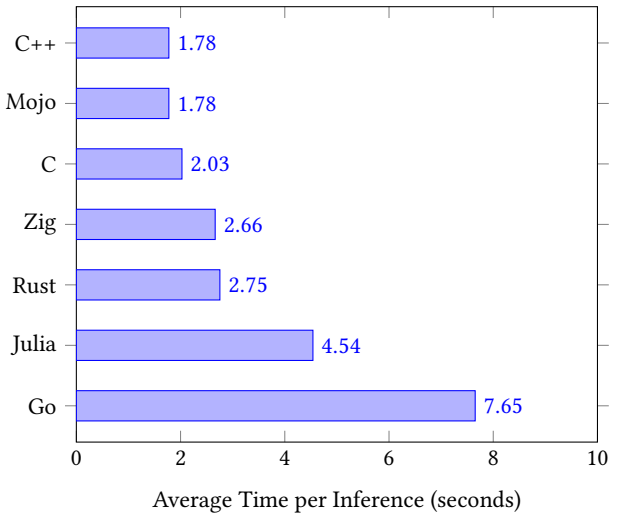

Figure 9: Average Time per Inference for stories110M.bin model.

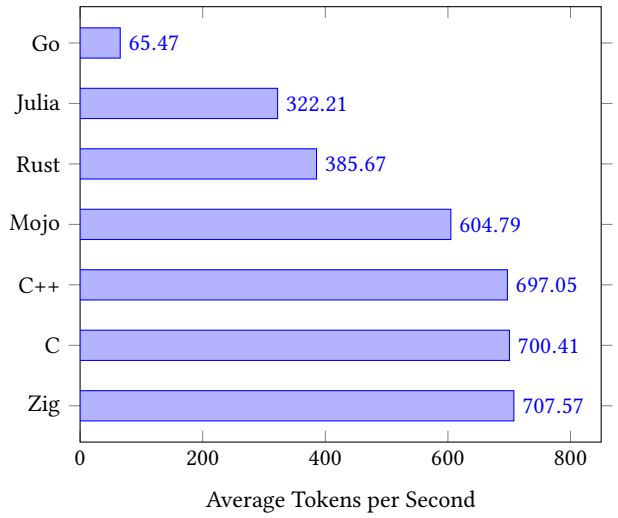

Figure 10: Average Tokens per Second for stories15M.bin model

SDK's effectiveness as an efficient solution for Llama2 inference on Apple Silicon. Its competitive performance, ease of use, and Python compatibility present a unique advantage in the LLM deployment landscape. Leveraging the extensive Python ecosystem for pre-processing, post-processing, and integration with other machine-learning tools enhances Mojo's appeal to developers seeking performance and productivity.

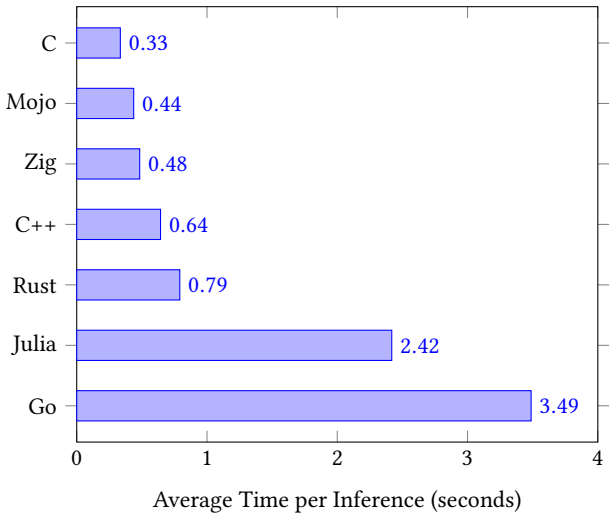

Figure 11: Average Time per Inference for stories15M.bin model

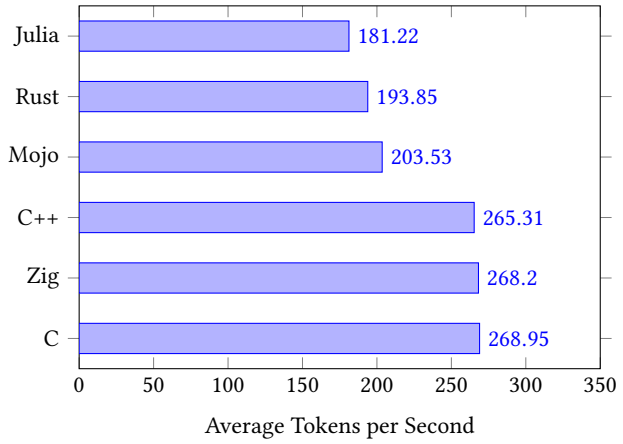

Figure 12: Average Tokens per Second for stories42M.bin model

4.3 Advantages of Mojo SDK

Mojo SDK streamlines LLM development with its Python-like syntax and seamless Python library integration, simplifying complexities inherent in lower-level languages like C or C++. This abstraction lets developers focus on model integration and application logic rather than grapple with memory management or performance optimization intricacies. Despite its high-level approach, Mojo's compiler infrastructure ensures optimal performance, often rivaling or surpassing established implementations. Its accessibility to Python developers broadens its user base, while an active community fosters collaboration and accelerates technology adoption.

4.4 Limitations and Future Work

While this study focused on CPU-based inference on Apple Silicon, future research could explore the impact of GPU acceleration on performance, particularly considering Mojo's adaptability to diverse hardware. Additionally, investigating Mojo SDK's performance and optimization across different hardware platforms with varying architectures or resource constraints would enhance understanding its capabilities and limitations. Moreover, the emergence of efficient and user-friendly LLM inference solutions like Mojo SDK has significant implications for the future of LLM deployment. Lowering the barrier to entry and enabling efficient execution on resource-constrained devices, Mojo empowers a broader range of developers and researchers to explore the potential of LLMs in various applications. This could lead to accelerated innovation in edge computing, personalized AI assistants, and LLM-powered mobile applications.

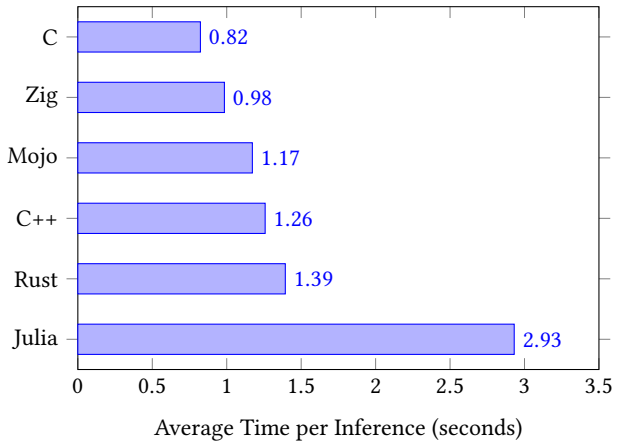

Figure 13: Average Time per Inference for stories42M.bin model

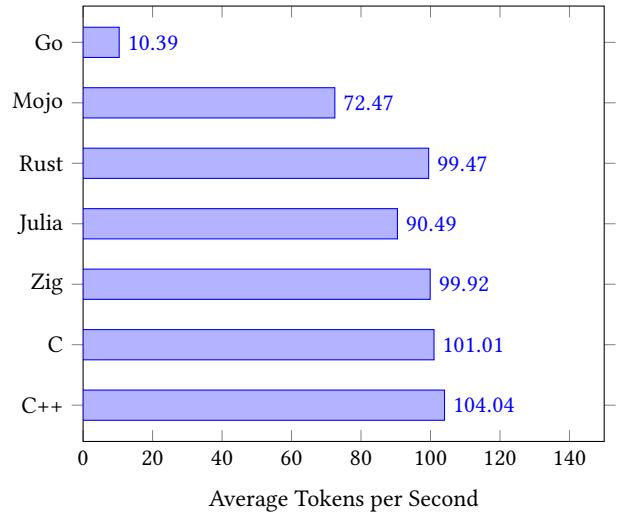

Figure 14: Average Tokens per Second for stories110M.bin model

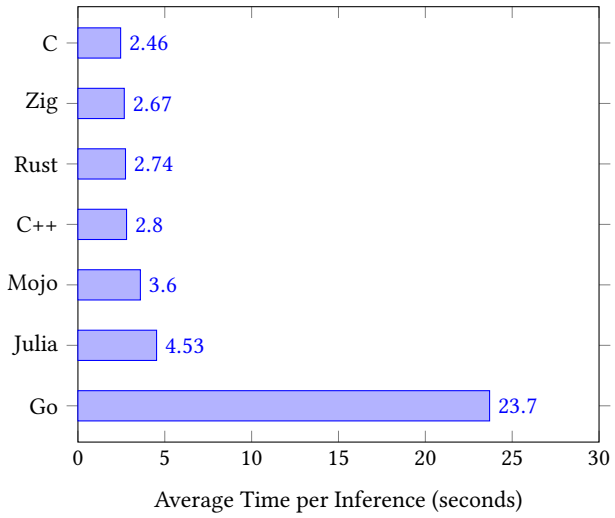

Figure 15: Average Time per Inference for stories110M.bin model

5 CONCLUSION

In conclusion, this study highlights Mojo SDK as a compelling solution for efficient and accessible Llama2 inference on Apple Silicon. Its competitive performance, ease of use, and Python compatibility position it as a valuable tool for developers and researchers alike. By simplifying development processes and enabling efficient execution on resource-constrained devices, Mojo SDK has the potential to democratize access to powerful language models, fostering innovation across various domains. Looking ahead, further exploration of Mojo's capabilities with GPU acceleration, diverse hardware platforms, and evolving LLM architectures holds promise for the future of LLM deployment. As the LLM landscape evolves, Mojo SDK emerges as a promising framework that empowers a broader range of users to harness the power of LLMs and unlock their transformative potential, driving innovation and advancements in the field.

REFERENCES

Richard Antonello, Nicole Beckage, Javier Turek, and Alexander Huth. 2020. Selecting informative contexts improves language model finetuning. *arXiv preprint arXiv:2005.00175* (2020).

Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-efficient DBMS configuration tuning. *arXiv preprint arXiv:2203.05128* (2022).

Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. 2023b. Inference-time intervention: Eliciting truthful answers from a language model, july 2023. *URL http://arxiv. org/abs/2306.03341* (2023).

Lei Li, Jing Chen, Botzhong Tian, and Ningyu Zhang. 2023a. Revisiting k-NN for Fine-Tuning Pre-trained Language Models. In *China National Conference on Chinese Computational Linguistics*. Springer, 327–338.

Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. 2024. Fine-tuning language models with just forward passes. *Advances in Neural Information Processing Systems* 36 (2024).

Danilo Vucetic, Mohammadreza Tayaranian, Maryam Ziaeefard, James J Clark, Brett H Meyer, and Warren J Gross. 2022. Efficient fine-tuning of compressed language models with learners. *arXiv preprint arXiv:2208.02070* (2022).

Yue Yu, Simiao Zuo, Haoming Jiang, Wendi Ren, Tuo Zhao, and Chao Zhang. 2020. Fine-tuning pre-trained language model with weak supervision: A contrastive-regularized self-training approach. *arXiv preprint arXiv:2010.07835* (2020).

Renrui Zhang, Jiaming Han, Chris Liu, Peng Gao, Aojun Zhou, Xiangfei Hu, Shilin Yan, Pan Lu, Hongsheng Li, and Yu Qiao. 2023. Llama-adapter: Efficient fine-tuning of language models with zero-init attention. *arXiv preprint arXiv:2303.16199* (2023).