International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 11 Issue: 11 | Nov 2024 www.irjet.net p-ISSN: 2395-0072

High-Performance Machine Learning Inference Systems with Rust: A Review of Techniques, Tools, and Optimizations

Rajavi Mishra[1]

[1]*Software Development Engineer, Sunnyvale USA*

Abstract - *Machine learning (ML) inference systems are integral to a wide variety of real-time applications, demanding low-latency, high-throughput, and scalability. However, traditional languages like Python face significant challenges in meeting these demands due to inefficient memory management and limitations with concurrency. Rust, with its strong memory safety, fine-grained control over system resources, and powerful concurrency model, emerges as a promising alternative for high-performance ML inference systems. This paper reviews the techniques, tools, and optimizations in Rust's ecosystem that enable scalable and efficient ML inference. Key features such as Rust's ownership-based memory management, fine-tuned control over CPU and GPU concurrency, and zero-cost abstractions are examined in the context of optimizing ML inference workloads. The paper also explores the integration of Rust-based frameworks, including tch-rs and ONNX Runtime, into real-world ML deployment pipelines, enabling efficient model development and deployment at scale.*

Key Words: *Machine Learning, Model Inference, Memory Management, Parallelism, Concurrency, Rust, Onnx Runtime*

1.INTRODUCTION

Machine learning (ML) inference powers a wide range of real-world applications, from image recognition [1] to natural language processing [2], to autonomous vehicles [3]. While model training is foundational in enabling systems to learn patterns from large datasets, ML inference delivers predictive capabilities in production environments. It enables systems to make predictions on new, unseen data using models that have already been trained and deployed at scale. In contrast to model training that is periodically performed on offline datasets of a predefined size, inference is a continuous process serving real-time predictions.

1.1 Challenges

To deliver a user-friendly experience, any real-time production system must operate on the following tenets: (i) serve accurate predictions, (ii) provide a response with minimal delay (low latency), (iii) process as many requests as possible per unit time (high throughput) in an uninterrupted manner, (iv) optimize resource utilization within cost constraints, and (v) scale across hardware to accommodate variable traffic without compromising system availability and response quality. ML Inference workloads operating in a production environment are no exception to these standards. They may handle massive query volumes, with some systems handling upwards of 200 trillion queries per day [4]. They operate within stringent latency constraints, typically between 100 to 300 milliseconds [5]. Furthermore, they may face fluctuating traffic patterns, including predictable variations such as peak and off-peak usage (e.g., daytime versus nighttime, or seasonal) as well as unpredictable disruptions, including data surges triggered by trending topics, one-off application overload, or system changes [6, 7]. To overcome these varying loads, systems must dynamically scale resources while maintaining efficiency and system stability (availability and response accuracy).

1.2 Approaches and Limitations

1.2.1 Model-Level Optimizations

Model optimization techniques such as pruning, which involves removing less important parameters, and quantization, which lowers the precision of weights to reduce model size, have been extensively studied in academic literature to enhance inference performance [8, 9, 10]. These techniques aim to decrease memory usage and computational costs, making models more suitable for deployment in resource-constrained environments like mobile and IoT devices. However, these techniques primarily focus on the model's architecture and do not address broader system-level challenges [10].

1.2.2 System-Level Optimizations

Software design and algorithmic optimizations in memory management, parallelism, and resource allocation help address the aforementioned system-level challenges. The choice of programming language becomes paramount in implementing these system-level optimizations effectively. Developers must choose languages that allow precise control over memory and concurrency, avoiding performance bottlenecks or runtime overheads. Python is widely adopted for ML workflows due to its ease of use and robust ecosystem of libraries, use cases, and comprehensive tutorials. However, it presents critical performance and memory-management challenges for high-performance, real-time inference systems.

Global Interpreter Lock (GIL). Python's concurrency model relies on multithreading within a shared memory space, but its performance is constrained by the Global Interpreter Lock (GIL). The GIL is a mutex that ensures only one thread executes Python bytecode at a time, primarily to prevent data races and memory corruption during object reference count updates. While this approach simplifies memory management and synchronization, it restricts parallelism for CPU-bound tasks, irrespective of the number of processor cores available [11]. This limitation significantly reduces Python's effectiveness in high-performance multithreaded environments. Furthermore, the GIL exacerbates context-switching overhead as threads must repeatedly acquire and release the lock. This is particularly problematic for mixed CPU and I/O-bound workloads, where the "convoy effect" arises – threads handling I/O operations are blocked by CPU-bound threads holding the GIL, introducing unpredictable delays [12, 13]. These inefficiencies degrade performance and scalability, making Python unsuitable for real-time, multi-threaded applications without extensive optimization or reliance on external tools.

Garbage Collection (GC). Python employs a two-tiered garbage collection (GC) mechanism comprising reference counting and cyclic garbage collection. Reference counting ensures that objects are deallocated when their reference count drops to zero. However, this process incurs performance overhead in multithreaded environments where GC needs to acquire the locks to update reference counts, further increasing computational costs. Cyclic GC is designed to reclaim memory from object cycles but operates non-deterministically, leading to unpredictable pauses during program execution that are particularly detrimental in production systems with stringent latency requirements. In long-running applications with high object churn and dynamic memory allocation, memory fragmentation further degrades performance. Even after deallocation, large memory blocks may not be immediately returned to the system, increasing overall memory usage. Moreover, during GL cycles, threads performing memory-intensive operations are forced to wait, exacerbating memory growth and contributing to performance bottlenecks. Similarly, cyclic GL cannot operate concurrently with other threads due to its reliance on acquiring the GIL. Consequently, GC cycles may be deferred, allowing unused memory to accumulate unchecked. As the memory heap grows, GC operations become increasingly computationally intensive, compounding latency and performance issues [14].

In contrast to Python, Rust offers a promising alternative for ML inference due to its safe and stable memory management system, zero-cost abstractions, concurrency model, fine-grained control over system resources, and several other features that address performance requirements of inference systems.

This paper provides an overview of production-grade ML inference systems and explores Rust's key features, such as its memory safety model, concurrency capabilities, and integration with popular ML frameworks, examining how these aspects contribute to building efficient, scalable, and reliable ML inference pipelines.

2. UNDERSTANDING ML INFERENCE PIPELINE

At a high-level, an ML inference pipeline comprises four key stages, each posing distinct challenges [16] and requiring (i) system-level optimizations to meet the demands of a production-grade ML inference system, (ii) monitoring to detect business metrics, prediction quality drifts, performance bottleneck and software failures, and (iii) feedback loops to allow model improvements.

Pre-processing. This stage transforms raw data into a consistent format suitable for feature generation and model inference. Tasks include data cleaning (e.g., handling missing or anomaly values), normalization of numerical data to fit within a defined range, encoding categorical variables, and tokenization of unstructured text. Additionally, text data may be converted into semantic vector representations, such as embeddings, to capture the underlying meaning. Efficient pre-processing is crucial to ensure that input data adheres to the model's requirements and avoids introducing bias or inconsistencies.

Feature generation. This stage drives actionable and measurable signals from the pre-processed data. It selects data parameters and applies transformation techniques like feature scaling, dimensionality reduction, or statistical aggregation to highlight relevant patterns. The quality of generated features is key to enhancing model accuracy and ensuring interpretability.

Model Inference. In this stage, the trained model is loaded into memory and applied to the input features using a defined inference strategy like point-wise, pair-wise, or list-wise, depending on the task. Inference can be executed in various modes with implications on latency: batch processing for parallel inference of smaller batches of data [15], real-time inference for individual predictions, or a hybrid approach that combines the two for optimized performance. The model must be deployed cost-effectively while maintaining accuracy, system performance and scalability. Model deployment can be either remote, exposing an endpoint to serve the model and invoking it using a RESTful API, or embedded, where the model runs directly on the host device. Lightweight models can often run efficiently on the host device, while resource-intensive models may need to be deployed and run remotely. Finally, appropriate hardware needs to be selected for the tasks, with CPUs being more suitable for simpler models and GPUs or TPUs for more computationally-intensive models [15]. Overall, choice of inference strategy,

execution mode, deployment strategy and hardware will all impact and will be driven by expectations of quality, latency and cost.

Post-processing. In this stage, model predictions are processed to fit the requirements of downstream components or systems and align with business objectives. Model output is converted into actionable formats like numerical scores or categorical labels. It may be further processed with techniques like threshold-based filtering or confidence-score bucketing, especially for classification or ranking tasks.

In advanced systems, multiple models are often chained together, each addressing a specific business task and processing the output of previous models. Each model may have its own distinct inference pipeline, including pre-processing, feature generation, inference, and post-processing stages. To optimize efficiency, common pre-processing steps can be shared across models, and feature generation can leverage feature stores for one-time creation. These systems may also incorporate advanced strategies like autoscaling to manage varying traffic loads and model-switching to select model variants based on quality and latency profiles in response to varying resource demands and nature of tasks [17]. This ensures dynamic resource allocation and the deployment of the most appropriate model for the specific workload.

3. KEY RUST FEATURES FOR HIGH-PERFORMANCE

Efficient memory management is a critical requirement in high-performance ML inference systems, where low latency and high throughput are essential. Rust's absence of garbage collection and its fine-grained control over memory and concurrency enable optimal resource utilization and eliminate unpredictable pauses. These capabilities make Rust an ideal choice for scaling inference systems, processing large datasets, generating features from them and meeting real-time demands with consistent quality, reliability and performance.

3.1 Memory Safety Without a Garbage Collector

Rust has a three-tenet ownership model that manages memory without a garbage collector in a performant, deterministic and memory-safe manner.

Ownership. First tenet ensures that every piece of data has a single owner and data ownership can only be transferred, not copied. The memory is freed and de-allocated when the owner goes out of scope [17, 18]. This predictable behavior prevents memory leaks and sudden, unpredictable pauses as seen in Python GC during memory cleanups, which can cause latency spikes in inference systems.

Borrowing. Second tenet allows data to be borrowed and accessed by other parts of a program without actually transferring ownership of the value to another variable. Data can be borrowed in two ways. Immutable borrowing permits multiple references to a data value to co-exist for read-only access. This allows data to be shared across multiple threads for concurrent reads. Mutable borrowing permits only a single reference with write access, preventing data race conditions since multiple threads can safely access and read shared data without another thread modifying it [17, 19]. This enables memory-safe concurrency allowing systems to parallelize tasks like processing large input dataset in batches and running inferences across multiple CPU cores or distributed nodes.

Lifetime. Third tenet ensures that Rust tracks the lifetime of all references to data [17, 20]. This again prevents unsafe and inefficient memory access by avoiding dangling references, null pointer dereferencing, and data races, all of which can cause runtime crashes, unsafe and incorrect memory access, unpredictable behavior. In ML inference systems with large input datasets and complex models, this level of memory safety helps maintain system stability and accuracy.

Overall, these tenets provide an efficient and deterministic memory management system that allow the inference pipeline to efficiently handle multiple concurrent requests while maintaining the accuracy and stability of the model.

3.2 Concurrency and Parallelism

Concurrency refers to managing multiple tasks simultaneously in a thread-safe manner, while parallelism involves actual simultaneous execution of tasks to maximize computational throughput [20]. Both are critical to scalable and efficient ML inference pipelines. As datasets grow and model architectures become more complex, it becomes increasingly important to optimize resource utilization, distribute tasks effectively, and leverage parallel and concurrent execution across CPUs and GPUs. Rust's safety, performance, and control makes it uniquely positioned to address these challenges, offering powerful abstractions for concurrent and parallel programming.

3.2.1 CPU-Level

Rust's borrowing tenet ensures thread-safety during multithreaded execution of tasks without requiring locks [21], mitigating risks like data races. This is advantageous for compute-intensive tasks such as batch inference where large datasets need to be processed concurrently across multiple cores or nodes. Rust also offers several zero-cost abstractions – native tools and libraries with minimal runtime overhead – that enhance its concurrent abilities at a hardware level without having to write low-level code.

Rayon. This is a data-parallelism library that abstracts low-level thread management and parallelizes computation tasks across multiple CPU cores. For instance, in ML inference, it can simultaneously process large datasets partitioned into smaller chunks, significantly boosting throughput and reducing latency. Rayon employs work-stealing schedulers and dynamically distributes tasks across threads to maximize CPU utilization and minimize idle time. Its integration with Rust's iterator traits allows complex operations to be parallelized while preserving code simplicity and efficiency [22, 23].

async/await. Rust's async/await paradigm enables asynchronous programming with non-blocking execution of I/O-bound tasks, hence maintaining system responsiveness under high load. Here, asynchronous functions are transformed into state machines that progress through discrete execution states. When an await expression is encountered (for example, a system may be waiting for an external event), the task is suspended, freeing resources for other tasks. Upon completion of the awaited operation, execution resumes from the suspended state. This mechanism eliminates the need for thread-based context switching, which traditionally incurs significant overhead. Async/await decouples compute-bound tasks (for example, ML inference or feature engineering) from I/O-bound tasks (for example, fetching input data), allowing the system to maximize resource utilization [24]. Such parallelism is particularly advantageous in data pipelines involving streaming real-time data inputs(for example, from sensors) or where preprocessing, inference, and postprocessing stages are chained.

Tokio. This provides a runtime infrastructure to execute asynchronous code in Rust. It complements async/await by offering cooperative multitasking, where tasks yield control to ensure optimal resource sharing [25]. Tokio can be useful for I/O-heavy operations, such as retrieving remote data, communicating with distributed nodes, or loading large ML models from the cloud.

Rust's ownership model, combined with zero-cost abstractions like Rayon, async/await, and Tokio, ensures optimal CPU utilization and efficient, thread-safe concurrency.

3.2.2 GPU-Level

CPUs have lower memory bandwidth, higher energy consumption per computation, and fewer cores compared to specialized hardware. As a result, CPUs may struggle to meet the demands of ML inference systems using resource-intensive deep learning models and require low-latency responses. In contrast, GPU acceleration is more efficient for handling these workloads, offering the parallel processing power necessary to meet performance requirements. There are several Rust libraries that provide an ergonomic interface for GPU-based parallelism. CUDA crate provides bindings for NVIDIA CUDA, allowing Rust developers to leverage CUDA's high-performance parallel computation operations. This includes managing memory between host and device, kernel launches, and synchronizations. OpenCL offers hardware-agnostic interface for GPU acceleration, making it suitable for systems requiring portability across heterogeneous devices. It is best-suited for relatively lightweight inference tasks as compared to CUDA. WGPU is primarily designed for graphic rendering in gaming and high-performance computing tasks. It provides a safe, cross-platform abstraction layer compatible with various low-level graphics interfaces like Vulkan, Metal, and DirectX [26, 27, 28, 29].

Complex, production-grade ML inference systems may adopt a hybrid approach, spanning workloads across CPUs and GPUs based on task characteristics, while optimizing for both latency and cost efficiency. Lightweight tasks, such as data preprocessing or feature extraction, may be executed on CPUs, while computationally intensive operations, such as tensor operations or inference on large models, may be offloaded to GPUs. Rust's zero-cost abstractions and inter-device communication capabilities facilitate integration between CPU and GPU operations, enabling systems to balance computational load and minimize latency.

4. ML INFERENCE FRAMEWORKS IN RUST

Rust offers several libraries to optimize model inference and deployment. These frameworks leverage Rust's performance, memory safety, and concurrency features to build efficient and scalable systems. Following are key Rust-supported frameworks for ML inference and model deployment in production environments.

tch-rs. This library provides Rust bindings for PyTorch, enabling access to its tensor operations and pre-trained models. This library supports both CPU and GPU-based inference, making it well-suited for real-time applications that require high throughput and low-latency execution. It offers several key benefits, (i) numerical computations with minimal memory overhead and optimal performance, (ii) integration support for PyTorch-trained models directly into Rust-based systems, (iii) hardware acceleration for GPUs through CUDA, enabling optimized deep learning model inference. This capability is particularly advantageous for tasks like image classification and natural language processing, where the computational demands of deep learning models are high [30].

ONNX Runtime for Rust (ORT). ORT is a high-performance engine for deploying Open Neural Network Exchange (ONNX) models. ONNX is a standardized format that allows models to be trained in one framework, such

as TensorFlow, and deployed across various environments, promoting interoperability between different ML frameworks. ORT accelerates inference on both CPU and GPU, with optimizations for specific hardware accelerators, including NVIDIA TensorRT, OpenVINO, and CUDA. These optimizations significantly improve inference speed and reduce resource consumption, making ORT suitable for large-scale, production-grade workloads. Additionally, ONNX Runtime's ability to handle models from multiple training frameworks enhances the flexibility of deploying machine learning solutions, allowing for smoother transitions from development to production while ensuring compatibility across various tools and hardware [31].

ndarray. This library handles multi-dimensional arrays for tensor operations. It efficiently supports a range of operations such as element-wise calculations and matrix reductions, all of which are fundamental for pre-processing and feature generation in ML inference pipelines. By leveraging Rust's ownership model, ndarray ensures memory safety during computations, eliminating issues such as memory leaks. This makes it a reliable foundation for many Rust-based ML workflows, particularly for data processing tasks that require high efficiency. Finally, ndarray integrates with other Rust libraries allowing for flexible and scalable pipelines in real-time, production applications.

Linfa. This library is designed to implement common machine learning algorithms, such as support vector machines, k-means clustering, and linear regression. Supporting both supervised and unsupervised learning tasks, It leverages ndarray for memory-efficient, parallelized processing of large datasets. The library can integrate with deep learning frameworks like tch-rs and ORT, enabling the creation of hybrid pipelines that combine traditional machine learning algorithms with deep learning models, creating flexible workflows that can address a broad array of tasks. For instance, linfa can be used for tasks like data pre-processing and feature extraction, while tch-rs and ORT can handle more computationally intensive tasks like model inference and fine-tuning.

Overall, this modular approach allows developers to choose the most appropriate tools for each task, optimizing both performance and efficiency across different stages of the ML workflow.

5. CONCLUSION

This article explored how Rust's memory safety, concurrency, and resource management features make it ideal for high-performance ML inference systems. Its ownership model ensures efficient memory usage, while zero-cost abstractions, like Rayon for parallelism and Tokio for async programming, support low-latency, high-throughput ML pipelines. Libraries such as ONNX Runtime for Rust, tch-rs, linfa, and ndarray further enhance its capabilities in machine learning workflows.

Overcoming the limitations of traditional languages, Rust is becoming a go-to choice for building high-performance, reliable systems. Looking ahead, with growing adoption in major tech companies, its popularity among developers and increasing demand for real-time, resource-intensive services, Rust is well-positioned to support concurrent applications across systems programming, game development, and embedded systems. As tooling and interoperability with other languages improve, Rust can further expand into areas like web development, DevOps, and cloud computing, ensuring a promising future for high-performance, production-grade systems.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proceedings of the Conference on Computer Vision and Pattern Recognition, 2016. doi: 10.1109/cvpr.2016.90.

[2] A. Parikh, O. Täckström, D. Das, and J. Uszkoreit, "A decomposable attention model for natural language inference," in Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2016. doi: 10.18653/v1/d16-1244.

[3] D. Xu, D. Anguelov, and A. Jain, "Pointfusion: Deep sensor fusion for 3D bounding box estimation," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018.

[4] K. Lee, V. Rao, and W. Arnold, "Accelerating Facebook's infrastructure with application-specific hardware," Engineering at Meta, Mar. 14, 2019. [Online]. Available: https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/.

[5] F. Ben, "API Response Time Explained: Wrangling Milliseconds for Peak Performance," Odown.io, 2024. [Online]. Available: https://odown.io/blog/api-response-time-standards. Accessed: Nov. 30, 2024.

[6] G. Vasiliadis, R. Tsirbas, and S. Ioannidis, "The Best of Many Worlds: Scheduling Machine Learning Inference on CPU-GPU Integrated Architectures," in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2022. doi: 10.1109/ipdpsw55747.2022.00017.

[7] B. Li, S. Samsi, V. Gadepally, and D. Tiwari, "Kairos: Building cost-efficient machine learning inference systems with heterogeneous cloud resources," in Proceedings of the 32nd International Symposium on

High-Performance Parallel and Distributed Computing, Aug. 2023, pp. 3–16.

[8] X. Liu, "Model Optimization Techniques for Embedded Artificial Intelligence," in 2021 2nd International Conference on Computing and Data Science (CDS), Jan. 2021, pp. 1–6. IEEE.

[9] C. Surianarayanan, J. J. Lawrence, P. R. Chelliah, E. Prakash, and C. Hewage, "A survey on optimization techniques for edge artificial intelligence (AI)," Sensors, vol. 23, no. 3, p. 1279, 2023.

[10] F. Sabah, Y. Chen, Z. Yang, M. Azam, N. Ahmad, and R. Sarwar, "Model optimization techniques in personalized federated learning: A survey," Expert Systems with Applications, 2023, p. 122874.

[11] M. Mohan, "Codedamn," Codedamn, Mar. 24, 2023. [Online]. Available: https://codedamn.com/news/python/demystifying-python-gil-concurrency-performance. Accessed: Nov. 30, 2024.

[12] V. Skvortsov, "Python behind the Scenes #13: The GIL and Its Effects on Python Multithreading," Tenthousandmeters.com, Sep. 22, 2021. [Online]. Available: https://tenthousandmeters.com/blog/python-behind-the-scenes-13-the-gil-and-its-effects-on-python-multithreading/.

[13] I. Turner-Trauring, "When Python can't thread: a deep-dive into the GIL's impact," PythonSpeed, Apr. 28, 2022. [Online]. Available: https://pythonspeed.com/articles/python-gil/.

[14] S. K. Sahoo, "Python's Garbage Collection Explained," Soumendra's Blog, Mar. 31, 2023. [Online]. Available: https://blog.soumendrak.com/garbage-collection-in-python. Accessed: Nov. 30, 2024.

[15] D. Crankshaw et al., "InferLine: ML prediction pipeline provisioning and management for tight latency objectives," arXiv preprint, arXiv:1812.01776, 2018.

[16] S. Ghafouri et al., "IPA: Inference Pipeline Adaptation to Achieve High Accuracy and Cost-Efficiency," arXiv preprint, arXiv:2308.12871, 2023.

[17] R. Jung, "Understanding and evolving the Rust programming language," Ph.D. dissertation, Faculty of Mathematics and Computer Science, Saarland University, Germany, 2020.

[18] S. Klabnik, and C. Nichols, "What Is Ownership?" The Rust Programming Language. [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[19] S. Klabnik, and C. Nichols, "References and Borrowing," The Rust Programming Language. [Online]. Available: https://doc.rust-lang.org/1.8.0/book/references-andborrowing.html.

[20] S. Klabnik, and C. Nichols, "Lifetimes," The Rust Programming Language. [Online]. Available: https://doc.rust-lang.org/rust-by-example/scope/Lifetime.html.

[21] "Concurrency in .NET: Modern patterns of concurrent and parallel programming," [Online].

[22] J. Stone, and N. Matsakis, "Rayon: A data parallelism library for rust," GitHub, Mar. 8, 2023 [Online]. Available: https://www.github.com/rayon-rs/rayon.

[23] "Rayon 2022: A data parallelism library for the Rust programming language," [Online]. Available: https://docs.rs/rayon/latest/rayon/.

[24] Z. Cutner and N. Yoshida, "Safe session-based asynchronous coordination in rust," in Coordination Models and Languages: 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Springer International Publishing, 2021.

[25] "Rust Documentation: Crate tokio," [Online]. Available: https://docs.rs/tokio/latest/tokio/. Accessed: Aug. 2022.

[26] K. Anderson, "What hardware should you use for ML inference?" Telnyx, Aug. 29, 2024. [Online]. Available: https://telnyx.com/resources/hardware-machine-learning.

[27] Nvidia, "What is CUDA?" Nvidia. [Online]. Available: https://nvidia.com. Accessed: Mar. 21, 2024.

[28] P. Zunitch, "CUDA vs. OpenCL vs. OpenGL," Videomaker, Jan. 24, 2018. Accessed: Sep. 16, 2018.

[29] gfx-rs, "wgpu," GitHub, Mar. 8, 2023. [Online]. Available: https://github.com/gfx-rs/WGPU/blob/trunk/etc/big-picture.png.

[30] L. Mazare, "tch-rs," GitHub, Mar. 11, 2023. [Online]. Available: https://github.com/LaurentMazare/tch-rs.

[31] "Introduction | ort," Pyke.io, 2024. [Online]. Available: https://ort.pyke.io/. Accessed: Nov. 30, 2024.