Acoustic Species Identification: Inference Conversion

Converting an Inference Script from Rust to Python

Siya Kamboj

Undergraduate Student, University of California: San Diego, sskamboj@ucsd.edu

To monitor biodiversity non-invasively, researchers use acoustic loggers that generate terabytes of audio data. Manually labeling this data is a major bottleneck for environmentalists' ability to track biodiversity. A cross-platform desktop application, with efficient, on-device inference for species identification, can streamline this process. This paper explores converting Python-based inference pipelines into Rust, a language known for performance and safety. Using Burn and ONNX Runtime, we integrate machine learning inference into an Electron-based desktop app. This eliminates the need for a Python runtime or local web server, improving performance, portability, and deployment. I benchmarked Rust inference against Python and observed faster runtime. Hence, these results show Rust's potential for building high-performance, field-ready tools for scalable biodiversity monitoring.

Computing methodologies • Machine learning • Information systems applications • Biodiversity informatics

Rust, ONNX Runtime, Burn, Species Classification, Biodiversity Monitoring, Acoustic Data, Machine Learning Inference

1 INTRODUCTION

1.1 Background

Biodiversity is declining at alarming rates due to climate change, habitat loss, and human interference. Scientists have historically used items such as camera traps, binoculars, and feeding traps to track species population; however, these methods are often invasive, labor intensive, expensive, or not generalizable to all species. To monitor this loss non-invasively, researchers are turning to passive acoustic monitoring, which captures species presence through birdsong. This approach is especially useful for tracking indicator species like birds, whose calls can reveal ecosystem health. At UC San Diego, the Engineers for Exploration (E4E) lab has been working on the Acoustic Species Identification project to automate this monitoring using machine learning.

The collaborators in this initiative, the San Diego Zoo Wildlife Alliance (SDZWA), deploy low-cost Audio moth recorders in remote habitats such as the Peruvian Amazon. These devices collect terabytes of audio data, but analyzing it poses a bottleneck: manually labeling 1,500 hours of audio (approximately 3.9TB) would take nearly 187 days, assuming three seconds to label each second of sound. Hence, a scalable, automated approach is necessary.

While earlier phases of the project focused on training deep learning models in Python to classify bird calls, deploying those models efficiently in a real-world, field-deployable tool is still a major challenge.

1.2 This Work: Converting Python Inference Pipelines to Rust

The primary contribution of this paper is the conversion of a machine learning inference pipeline from Python to Rust to create a fast, reliable desktop application for species identification. Python is commonly used for model development, but it introduces runtime overhead, dependency issues, and platform compatibility limitations, especially when used in standalone desktop apps or offline environments. Rust, on the other hand, offers speed advantages, like C/C++, while also offering memory safety and a wide myriad of developer tools and libraries to assist in the inference model pipeline.

Particularly, this paper explores:

- Converting Python-based inference pipelines to Rust, using the Burn deep learning framework for pre and post processing and ONNX Runtime for model execution.
- Integrating the resulting Rust inference logic into a desktop application built with Electron, enabling full model execution without requiring a Python interpreter or web server.
- Benchmarking the Rust implementation against its Python counterpart, showing improved inference speed.
- Demonstrating that Rust's performance, memory safety, and cross-platform support make it a compelling option for field-ready ML applications, a usage that is not widely aware.

1.3 Context & Broader Applications

This conversion is part of a larger E4E effort to build a desktop app for species identification. The application provides tools for:

- Spectrogram visualization and annotation of bird vocalizations,
- Verification of model predictions through interactive audio and visual inspection,
- Local database integration for organizing audio clips, labels, and results,
- And customizable key bindings to support workflows.

While these features encompass the user-facing utility of the app, the work presented in this paper focuses specifically on the back-end portion: enabling fast and portable Rust-based inference, which enhances the app's performance and makes it suitable for deployment in conservation settings with limited computing resources. The images attached below contain information about some of the pages that have already been implemented in the desktop app.

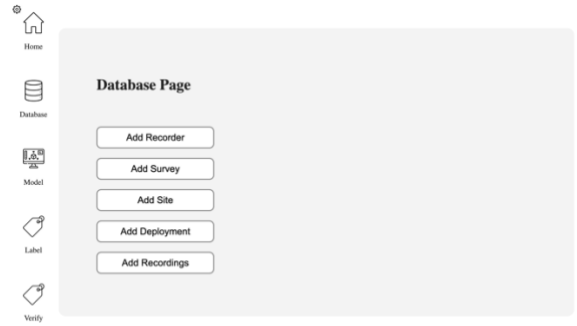

Figure 1: Database Page, where you can insert audio files to the database

Figure 2: Model Page, where the user selects an audio from the database and runs pre-defined model over it. Currently, no model is integrated; in fact, the Rust model from this paper will be integrated into this page.

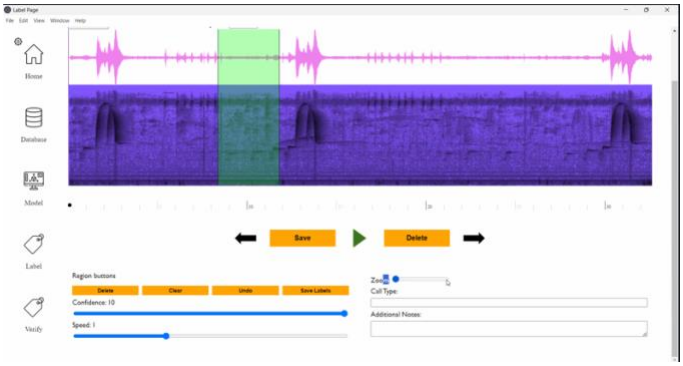

Figure 3: Label page, where the user can manually label an audio file from the database

1.4 Summary of Contributions

- Introduce a practical method to convert ML inference from Python to Rust using Burn and ONNX Runtime

- Propose a method for embedding this Rust-based inference into a desktop app for acoustic species identification
- Show that this transition improves performance and portability
- Lay the groundwork for scalable, field-deployable conservation tools powered by efficient ML

2 RELATED WORKS

These related works highlight that using machine learning for acoustic species is a proven method for tracking biodiversity and that Rust + Onnx offers a strong foundation for ML inference scripts. However, the integration of Rust + Onnx ML frameworks for analyzing birdsong is a largely unexplored field. In fact, integrating such a script into a desktop app with a database backing is also an unexplored field.

2.1 Passive Acoustic Devices for Field Monitoring

1. The Audiomoth, which is the acoustic logger that is utilized in all the avian ML biodiversity tracking efforts in Acoustic Species Identification deployments, is widely adopted for biodiversity surveys. These autonomous recorders collect long-term continuous audio (>1 TB), easing deployment but shifting the burden to efficient post-hoc analysis.
    a. From reference 1
2. Cornell's HaikuBox (2018) integrates BirdNET in a device for backyard bird monitoring, showcasing how embedded ML can scale bioacoustic detection—with over a billion recordings captured.
    a. From reference 2

2.2 Using ML for Acoustic Biodiversity Monitoring & Proving Acoustic Biodiversity Monitoring is a successful measure of biodiversity

1. This article highlights how bioacoustics and AI enable efficient, large-scale ecological monitoring compared to traditional method
    a. From reference 3
2. MDPI systematic review (2023): Surveys AI methods in ecoacoustics, underscoring deep learning's role in interpreting extensive wildlife audio data
    a. From reference 4
3. ArXiv (2024): Introduces real-time, AI-powered sensor networks for biodiversity tracking, tackling noise and overlapping calls
    a. From reference 5
4. MDPI. "A Methodological Literature Review of Acoustic Wildlife Monitoring" (2023)
    a. This survey examines various machine learning approaches for ecoacoustic monitoring, including CNNs, SVMs, and RNNs, and highlights key challenges in deploying systems at scale. It contextualizes our work by showing strong existing interest in ML for biodiversity, but also a notable gap in robust deployment frameworks, which our Rust-based inference engine and desktop app aim to fill.
    b. From reference 6
5. Stowell, D. "Computational Bioacoustics with Deep Learning: A Review and Roadmap" (2021)

    a. This paper reviews state-of-the-art deep learning techniques in bioacoustics and identifies technical hurdles such as model portability and real-time inference. It directly supports the existence of the desktop app by highlighting the lack of platform-agnostic, efficient inference tools, which our Rust + ONNX Runtime pipeline addresses.
    b. From reference 7
6. Stowell & Plumbley. “Automatic Large-Scale Classification of Bird Sounds…” (2014)
    a. This study demonstrates that unsupervised feature learning on mel spectrograms significantly improves bird call classification at scale. Their results confirm our use of mel spectrograms as effective features, and justify the need for optimized preprocessing pipelines across languages.
    b. From reference 8
7. Burla Nur Korkmaz et al. “Automated Detection of Dolphin Whistles…” (2022)
    a. The authors apply CNNs to underwater recordings, illustrating that deep learning excels in noisy, real-world acoustic environments. This work gives precedent for applying convolutional ML models in diverse acoustic contexts, reinforcing the value of the EfficientNet-based approach in this project.
    b. From reference 9
8. Gouvêa et al. “Interactive ML Solutions for Acoustic Monitoring…” (2023)
    a. This paper focuses on user-facing interactions with acoustic ML systems in biosphere reserves, emphasizing expert engagement and system usability. This work builds on these principles by creating a desktop app with intuitive annotation, model feedback, and visualization, improving usability for experts.
    b. From reference 10

2.3 Rust in ML and Inference Performance

1. ScienceDirect: “MicroFlow: An Efficient Rust-Based Inference Engine for TinyML” (2025)
    a. MicroFlow demonstrates Rust-based deployment of neural networks on embedded systems, highlighting performance benefits. This aligns closely with our goal to use Rust for efficient inference, validating our choice of Rust + ONNX Runtime for desktop and embedded applications.
    b. From reference 11
2. ArXiv: “Energy Consumption in DL Models & ONNX Runtime” (2024)
    a. This paper evaluates energy and inference efficiency across TensorFlow, PyTorch, and ONNX, noting ONNX’s advantage in hardware-specific execution. It confirms that our choice of ONNX Runtime is optimal for both speed and energy efficiency, crucial for resource-sensitive desktop apps.
    b. From reference 12

3 TECHNICAL INFORMATION

This section outlines the key technical components of the inference pipeline, including the Python-based baseline, the structure of the inference script, the model architecture, and the rationale and structure for porting

the system to Rust. It also defines core tools and frameworks, such as Hugging Face, ONNX Runtime, and Burn, that back the Python and Burn implementations respectively.

3.1 Inference Script Framework

The overall goal of the inference pipeline is to take raw audio recordings and output a probability distribution over possible species present in each clip. The process consists of several key stages:

1. Audio Input: Raw audio files are collected from field-deployed devices such as AudioMoth recorders. These can be of variable length but are sampled at a fixed rate typically. In this implementation, we are using the Birdset audioset, which contains over 1000 hours of audio spanning 8 different locations. In this implementation, we are using the HSN (High Sierra Nevada) training, testing, and validation dataset.
2. Preprocessing: Each audio file is converted into 5-second-long audio clips. Then, they are converted to a mel spectrogram. Mel spectrograms are a time-frequency representation that maps audio across different frequency bands. This transformation is commonly used in audio machine learning because it closely resembles how humans hear audio and makes temporal patterns in calls easier to detect.
3. Model Inference: The spectrogram is passed into a convolutional neural network based on the EfficientNet architecture. EfficientNet is a family of deep learning models that scales depth, width, and resolution in a balanced way to achieve high accuracy with fewer parameters. The model outputs a table of probabilities, one for each species class.
4. Output Representation: The outputs are organized into a probability table, where each row corresponds to an input audio file and each column corresponds to a species. Each cell holds the predicted probability that the given species is present in that audio file.
5. Postprocessing: For each row, the species with the highest probability is selected as the predicted label. These results can then be saved to the database in the Electron desktop app, although this was not implemented in the Rust implementation.

This pipeline was initially implemented in Python using libraries such as PyTorch and librosa. However, Python's interpreter overhead, large dependency footprint, and runtime variability made it suboptimal for deployment in a desktop application.

3.2 Conversion from Python to Rust

To improve performance and portability, we ported the inference logic from Python to Rust, a systems programming language designed for speed, safety, and minimal runtime overhead. Rust offers several advantages over Python in production and embedded environments:

1. Compiled Execution: Rust compiles to native binaries, removing the need for an interpreter.
2. Memory Safety: It prevents null pointer dereferencing and race conditions at compile time, without garbage collection.
3. Cross-Platform Portability: Rust programs can be compiled for Windows, macOS, and Linux with minimal modification.

3.3 Rust Architecture

Two key tools enable machine learning inference in the Rust ecosystem:

- ONNX Runtime: ONNX (Open Neural Network Exchange) is an open-source format for representing ML models trained in different frameworks. I will use ONNX Runtime to convert the Python-trained EfficientNet model into ONNX format. ONNX Runtime in Rust provides efficient, hardware-accelerated inference using these exported models.
- Burn: Burn is a deep learning framework for Rust, similar to PyTorch. It provides abstractions for model building, training, and deployment. In our project, Burn is used primarily to structure the Rust-side application and could eventually support training as well. It is used because it supports a variety of backends, unlike vanilla Rust libraries which are limited.

Together, Burn and ONNX Runtime enable our system to run inference on-device in Rust, without relying on a Python environment or external web server. This should result in faster inference time, reduced memory usage, and greater stability across platforms.

3.4 Application Integration

Currently, the scope of this paper is going to be converting the Python inference script to Rust. However, in the future, the work will be towards integrating the Rust-based inference pipeline into a cross-platform desktop application built using Electron. The Electron frontend handles user interaction—loading audio files, visualizing spectrograms, and showing predictions—while the Rust backend performs efficient, local inference and facilitates communicates amongst the UI layer and database.

The final system will support:

- Batch processing of audio files
- Real-time display of predictions
- Exporting annotated results
- Offline functionality, crucial for use in remote environments

This will be the architecture of a deployable, user-friendly tool for scalable biodiversity monitoring.

4 MILESTONES

4.1 Initial Goals & Planning

At the beginning of the quarter, my primary goal was to convert the existing Python-based inference pipeline into a performant and portable Rust-based implementation, fully integrated into the desktop application. The milestone for the end of the quarter was to have this new inference module embedded within the Electron app, running entirely offline and without any dependency on Python.

To achieve this, I outlined the following deliverables early on:

1. Research and finalize the system architecture.
2. Set up both Python and Rust environments.
3. Implement preprocessing and postprocessing pipelines in Rust.
4. Port the inference logic using ONNX Runtime and Burn.
5. Integrate the Rust backend into the Electron desktop application.

5 PROGRESS BREAKDOWN

5.1.1 Weeks 1&2: Architecture Design & Setup

In the first two weeks, I finalized the core system architecture, including the decision to:

- Use ONNX Runtime to serve a model exported from PyTorch.
- Implement the full preprocessing pipeline natively in Rust.
- Leverage Burn for tensor representation and potential future model-building support.

I also set up and tested the original Python inference script, confirming the behavior and output structure. In parallel, I set up the Rust environment, installed necessary crates (e.g., ndarray, hound, burn, onnxruntime), and began researching audio processing in Rust.

5.1.2 Weeks 3-4: Preprocessing and Postprocessing in Rust

During this phase, I completed postprocessing first, as it was relatively straightforward. After inference, the model outputs a probability vector over all species classes. The highest-probability class is selected and stored as the final prediction for each audio file.

Preprocessing, however, was much more complex. First, I used a python file to extract the dataset from HuggingFace and save the split files to a Json file. Then, I was able to retrieve the data about the audio using Rust's library Serde. The next step was converting all the audio files to a Mel Spectrogram. The original python script accomplished this using lazy transforms, which deferred the creation of the spectrogram until it was needed (in other words, during the inference script). Once created, the spectrogram is not saved or cached anywhere. Rust does not support lazy transforms, so the pipeline revolved around converting the audio to mel spectrograms and saving that information as npy files that will be input to the inference code. This was the pipeline for accomplishing that, entirely in Rust:

1. Decode audio into raw waveform samples
2. Chunk waveform into shorter segments for consistent input size
3. Apply a mel filterbank to map audio data into a mel spectrogram.
4. Normalize values to [0,1] range for consistent scaling
5. Convert to tensor format using Burn to match model input expectations.

5.1.3 Weeks 5-6: Model Inference and Introducing Roadblocks

The final milestone was integrating the model using ONNX Runtime. In theory, this should have been a plug-and-play process. However, I encountered persistent runtime errors during inference, which I traced to two primary causes:

1. **Spectrogram format mismatch**: Rust-generated spectrograms could have differed subtly in format, scale, or channel layout compared to what the model (trained in Python) expected. This could have caused discrepancies in shape and value range.
2. **Python inference script complexity**: The original inference code from E4E included unnecessary features & abstractions and tightly coupled dependencies. In fact, the inference script caused issues not just for me but for the data augmentation team. Hence, it is plausible that the original script that we were attempting to port had issues.

5.1.4 Weeks 7-10: Revising My Implementation and Overcoming Roadblocks

To resolve these issues, I revised the architecture in two ways:

1. Fixing the Preprocessing Mismatch
    a. Used PyO3, a Rust-Python wrapper, to call Python's mel spectrogram generation function directly
    b. Saved the final spectrograms as .npy files (NumPy format), ensuring identical preprocessing as used during training

    This approach guaranteed consistency and eliminated discrepancies in shape and value scale
2. Simplifying the Inference Script
    a. Two of the project leads in Acoustic Species Identification created a new inference code that was minimized and removed unnecessary features from the original Python inference code. Since it was new and in the beta stages, there were some hiccups; however, the code was overall better suited for ONNX Runtime than the previous code.
    b. Re-exported the minimal model to ONNX format
    c. Loaded this simplified model using ONNX Runtime in Rust with successful inference execution
    d. Tweaked my preprocessing and postprocessing logic slightly to accommodate for this shift.

5.2 Final Status & Reflection

While I did not fully achieve my original goal of desktop app integration by the end of the quarter, I did achieve a reliable and testable Rust-based inference script.

5.3 Results & Findings

5.3.1 Strengths of Rust

To evaluate the performance of the Rust-based inference implementation relative to the original Python pipeline, we measured total processing time across multiple iterations of the model. The evaluation accounted for both preprocessing and inference stages.

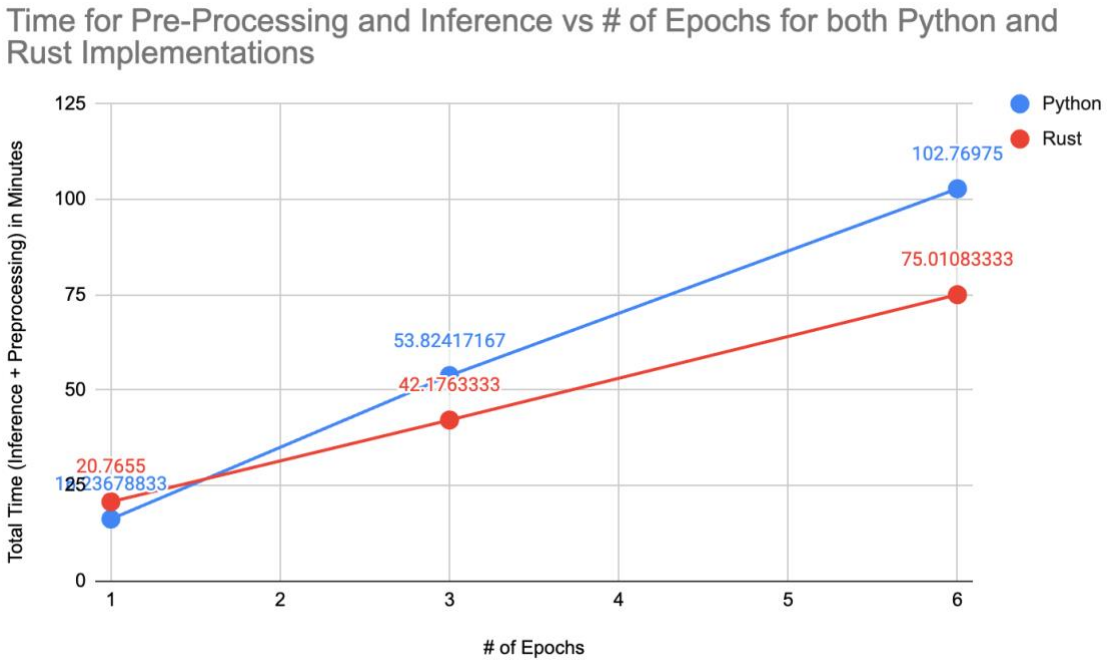

Figure 4: A graph plotting time for inference + pre-processing for both Rust and Python implementations. It was run 3 times: with 1 epoch, 3 epoch's and 6 epochs respectively. A linear trend line was fit on top of the points. It also should be noted that, for the Rust implementation, a one-time preprocessing cost of 11.588 minutes was added to each total runtime to simulate a first-time setup scenario. In contrast, the Python implementation includes preprocessing inline during inference, so no adjustment was made

The results, visualized in the figure above, demonstrate that the Rust implementation is consistently ~2× faster than Python, even when run without GPU acceleration. This trend holds across multiple inference runs after 1.5 epochs (specifically 3 and 6 epochs), with Rust showing reduced total runtime at each point. For example, at 6 epochs, Python required approximately 103 minutes, while Rust completed the same workload in just 75 minutes.

This performance gap is attributed to three main factors:

- No interpreter overhead in Rust, as it is a compiled systems language.
- Precomputed mel spectrograms in Rust, eliminating repeated transformations.
- Native ONNX Runtime optimizations

The root cause of this efficiency difference lies in computation strategy.

Python's inference pipeline, as explained earlier, relies on lazy transformations, meaning it computes mel spectrograms on-the-fly during each inference pass. While this design simplifies the code, it leads to repeated, redundant computation of the same spectrograms across epochs. Additionally, since the trend line for the Python inference is roughly linear, that demonstrates that no caching or memoization is taking place in the Python pipeline. As epochs increase, the time taken grows proportionally. This further supports the

idea that Python recomputes data on each pass, while Rust amortizes preprocessing costs over multiple inferences.

In contrast, the Rust pipeline precomputes and caches mel spectrograms ahead of time. Although this results in a higher up-front memory cost, the downstream inference process is significantly faster and more consistent across runs. This decoupling of preprocessing and inference is a key architectural benefit of the Rust implementation.

5.3.2 Tradeoffs of Rust

The only notable tradeoff in the Rust approach is increased memory usage due to the storage of spectrograms as precomputed float arrays. However:

- Memory is not currently a bottleneck, as the spectrogram data fits comfortably in available RAM during inference.
- The spectrograms are saved in .npy format using 32-bit floats; thus, there is room for further optimization via quantization. For example, reducing precision to 16-bit or 8-bit values, or applying ONNX quantization to the model itself, could further reduce storage footprint and improve inference speed

5.4 Integration Planning for Electron App

To prepare for full integration over the summer, I researched how to embed Rust logic in an Electron app. There are two common approaches:

1. Node Native Modules (via N-API or napi-rs)
    a. Compile Rust code into a Node-compatible module
    b. Electron frontend can call Rust functions directly
2. Command-Line Interface (CLI) Wrapper
    a. Run the Rust binary as a child process from Electron
    b. Exchange data via files or IPC (inter-process communication)

I will first choose to pursue option 1, Node Native Modules, for several reasons:

1. Performance: Native modules communicate in-memory, avoiding file I/O or process startup cost
2. User Experience: Native integration avoids jarring delays or external windows/processes, which can break the app's flow
3. Maintainability: The module is part of the application bundle, making version control, updates, and packaging easier.

However, this path also does require that all the Rust crates and runtime behaviors are compatible with N-API or napi-rs. Since this app is integrated with Burn, which is relatively new and untested, there is a chance that those crates may not integrate well. Hence, I may fall back to the CLI approach, which is less performance efficient but simpler to execute.

6 CONCLUSION

This project addresses a significant bottleneck in biodiversity monitoring: the need to process and label terabytes of acoustic data collected from field environments. While machine learning models have improved the accuracy of species identification, deploying these models efficiently in real-world applications remains a

challenge—especially when performance, portability, and ease of use are critical. To tackle this, we re-engineered the inference pipeline originally written in Python and ported it to Rust.

Our primary contributions include the successful implementation of a Rust-based inference engine using ONNX Runtime, the design of a preprocessing pipeline compatible with existing models, and a thorough comparison against the original Python implementation. Our results demonstrate that Rust significantly improves runtime efficiency—often by a factor of two. This positions Rust as a viable and scalable choice for machine learning deployment in field-ready applications.

Looking ahead, the immediate next step is to integrate the Rust inference engine into the existing Electron-based desktop app, enabling real-time, offline prediction and annotation. After completing several feedback iterations with end-users, I hope to revisit the mel spectrogram conversion logic and explore whether it can be fully re-implemented in native Rust. Doing so would reduce reliance on Python wrappers and improve maintainability and performance.

There is also significant opportunity for performance optimization. Currently, both the Python and Rust implementations use only 5–10% of the CPU, with minimal GPU utilization. Rust is not yet configured for GPU support, and multithreading has not been implemented. By adding GPU acceleration and multi-core parallelism, we can further reduce inference time and improve scalability for larger datasets.

Finally, a long-term goal is to mature the desktop application into a tool that can be used not just by UC San Diego researchers, but also by conservationists and ecologists around the world. With further testing and iteration, this work can contribute meaningfully to the development of accessible, high-performance tools for global biodiversity monitoring.

REFERENCES

[1] Lapp, S., Stahlman, N., & Kitzes, J. (2023). A Quantitative Evaluation of the Performance of the Low-Cost AudioMoth Acoustic Recording Unit. Sensors, 23(11), 5254. https://doi.org/10.3390/s23115254

[2] Heath, B., Orme, D., Picinali, L., & Ewers, R. (2023). The assessment and development of methods in (spatial) sound ecology (Doctoral dissertation, Ph. D. Thesis. Imperial College London. https://core. ac. uk/downl oad/pdf/5884 1 1731. pdf).

[3] JeppeH.Rasmussen et al. ,Sound evidence for biodiversity monitoring.Science385,138-140(2024).DOI:10.1126/science.adh2716

[4] D.A. Nieto-Mora, Susana Rodríguez-Buritica, Paula Rodríguez-Marín, J.D. Martínez-Vargaz, Claudia Isaza-Narváez, Systematic review of machine learning methods applied to ecoacoustics and soundscape monitoring, Heliyon, Volume 9, Issue 10, 2023, e20275, ISSN 2405-8440, https://doi.org/10.1016/j.heliyon.2023.e20275 (https://www.sciencedirect.com/science/article/pii/S2405844023074832)

[5] https://arxiv.org/html/2410.12897v1

[6] Sharma S, Sato K, Gautam BP. A Methodological Literature Review of Acoustic Wildlife Monitoring Using Artificial Intelligence Tools and Techniques. *Sustainability*. 2023; 15(9):7128. https://doi.org/10.3390/su15097128

[7] https://arxiv.org/abs/2112.06725

[8] https://arxiv.org/abs/1405.6524

[9] https://doi.org/10.48550/arXiv.2211.15406

[10] Gouvêa, T. S., Kath, H., Troshani, I., Luers, B., Serafini, P. P., Campos, I. B., ... & Sonntag, D. (2023). Interactive machine learning solutions for acoustic monitoring of animal wildlife in biosphere reserves : https://www.ijcai.org/proceedings/2023/0711.pdf

[11] Matteo Carnelos, Francesco Pasti, Nicola Bellotto, MicroFlow: An Efficient Rust-Based Inference Engine for TinyML, Internet of Things, Volume 30, 2025, 101498, ISSN 2542-6605, https://doi.org/10.1016/j.iot.2025.101498

[12] https://arxiv.org/html/2402.13640v1