

PolyChirp: Multi-Species Birdsong Classification Using TinyML on Low-Power Acoustic Sensors

Nathan Duboisset

École Polytechnique, Palaiseau, France
Freie Universität Berlin, Berlin, Germany
 nathan.duboisset@polytechnique.edu

Zhaolan Huang

Freie Universität Berlin
 Berlin, Germany
 zhaolan.huang@fu-berlin.de

Felix Bießmann

Berlin University of Applied Sciences (BHT)
 Berlin, Germany
 felix.biessmann@bht-berlin.de

Roudy Dagher

School of Engineering, HES-SO Valais-Wallis
 Sion, Switzerland
 roudy.dagher@hevs.ch

Antoine Lavandier

Inria
 France
 antoine.lavandier@inria.fr

Emmanuel Baccelli

Inria, France
Freie Universität Berlin, Germany
 emmanuel.baccelli@inria.fr

Abstract—Recent progress in the field of TinyML has demonstrated that low-power hardware based on microcontrollers can achieve bird species monitoring in real time based on acoustic sensor data for an entire breeding period on a single battery charge. However, the state of the art on low-power microcontrollers was so far limited to binary classification of a single species. In contrast, real fauna monitoring deployments often target multiple species simultaneously. To address this challenge we develop PolyChirp, an approach combining biological domain expertise, automated dataset curation, neural architecture optimization and novel hardware to achieve multiclass bird species detection in the wild. PolyChirp is based on newly designed tiny multiclass models that leverage recent microcontrollers and hardware acceleration with a neural processing unit (NPU). We evaluate the predictive performance of these models, and we measure their computational performance – memory footprint, latency, energy consumption – on common microcontroller hardware. Our results demonstrate that PolyChirp not only outperforms state-of-the-art on single species binary classification, but also achieves robust classification of up to 10 species simultaneously, while still fitting within the resource envelope of a sensor that must remain operational in the field for a full season on a single battery charge.

Index Terms—TinyML, Edge AI, Acoustic, Sensor, Low-power, Bird, Classification

I. INTRODUCTION

Passive acoustic monitoring of birds usually incurs field-deploying battery-powered sensors as audio recording units (ARU) for a full breeding season, recording continuously. Months later, the devices and their filled memory cards are collected for analysis. In fact, however, only a small fraction of these recordings is of interest. While a campaign targets one or more bird species, recordings end up also including in vast amounts sounds of wind, traffic, other animals etc. Crucially, storing spurious audio depletes the two budgets that constrain an ARU: memory and battery capacities.

One approach to circumvent these bottlenecks is to use TinyML to decide quickly directly on the sensor whether

the current acoustic signal is relevant or not. The difficulty is that the sensor is a low-power microcontroller with a few hundred kilobytes of RAM [1]. Hence, whatever makes that classification has to be very small, and in particular, a big model such as BirdNET [2] does not fit in memory. In contrast, prior work such as TinyChirp [3] demonstrate how microcontroller-based hardware can detect a single target bird species in the field, accurately enough to discard most spurious audio and significantly improve these bottlenecks.

However, such prior work still suffers from limitations in practice. The first is the number of species: a deployment usually follows several species at once, and a single-output detector can neither separate them nor report which one it heard. The second is portability across sites: the species that matter differ from one location to the next, so a model and a dataset built for one species in one place does not carry over, and redoing that data work by hand for every new site is exactly the cost one wants to avoid.

This paper thus introduces PolyChirp, a new multi-species on-device classification mechanism that can run on diverse microcontroller hardware. PolyChirp also provides an automated dataset generation scheme which facilitates re-training for different geographical locations and, accordingly, for different species subsets. In more details, our main contributions are as follows.

- We provide a geographical site-driven dataset builder that turns a location, a radius, and a target species count into a deployment-specific multi-class dataset.
- We design and implement a family of multi-class TinyML models with one sigmoid output per species, such that a single network handles both bird-sound detection and species classification. We publish the open source code of our implementations.
- We provide a comparative performance evaluation of predictive performance of PolyChirp using different multi-

class TinyML models, as well as measurements of Flash, RAM, inference latency, and energy consumption on common microcontroller hardware with and without neural network hardware acceleration (Nordic nRF54LM20 and the Raspberry Pi Pico 2).

- Compared to prior work, we show that PolyChirp not only performs better than TinyChirp in the single-species case, but also that PolyChirp can handle up to 10 species classification with accuracy above 0.98, using a tiny fraction of the memory budget necessary for BirdNet. We also measure the impact of neural network hardware acceleration (NPU) in practice on microcontrollers.

II. DATASETS

Most bird-audio benchmarks use a fixed, global species list. Instead, we automate dataset building based on the place where the device will be deployed.

Dataset Creation Automation – To generate the dataset, we query Xeno-Canto [4] for every recording inside a bounding box, defined by coordinate, radius, and a target species count N . We run BirdNET [2] on a random subsample of those recordings to catch species the metadata misses, and rank species by occurrence. By default, the first N species become the target classes. The remaining species are pooled into a single `non_target` class together with AudioSet [5] clips (speech, traffic, weather, machinery, indoor noise, etc.) and no-bird windows mined from recordings from the same area. A second primary source, the Macaulay Library [6], is queried in parallel to grow the target counts, using the same query system, to add more diverse recordings, instruments, and locations.

As an example, we run the procedure once for Paris with a 50 km radius and $N = 10$. The ten target classes range from 4870 to 10 000 clips. The `non_target` class holds 18 800 clips, split roughly evenly between 8400 other birds, 8400 AudioSet clips and 2000 no-bird clips. Recordings from Xeno-Canto and Macaulay are MP3 formats at mixed sample rates (32k to 48 kHz) and bit-rates (96k to 320 kbps); AudioSet clips are 10 s YouTube segments. We split each class 70/15/15 into train/val/test, stratified at the recording level so clips from the same source recording stay in the same split, to avoid data leakage.

Feature Selection and Pre-processing – Every recording is decoded and resampled to 16 kHz mono; levels are left as recorded. BirdNET then labels it: detections at confidence ≥ 0.92 produce 3 s clips around the detection, and recordings with no detection at all contribute a single random 3 s window to the no-bird pool. Labelling with BirdNET instead of the metadata tag lets us use the whole length of an XC or Macaulay recording while keeping the per-clip class confidence high, which matters because community recordings often spend most of their runtime on background.

For the mel-spectrogram models we use two front-ends, `tinychirp_mel` and `light_mel`. The first, following TinyChirp [3], is a 1024-point Hann window with a 256-sample hop (16 ms) and 80 mel bins over 80–8000 Hz, pro-

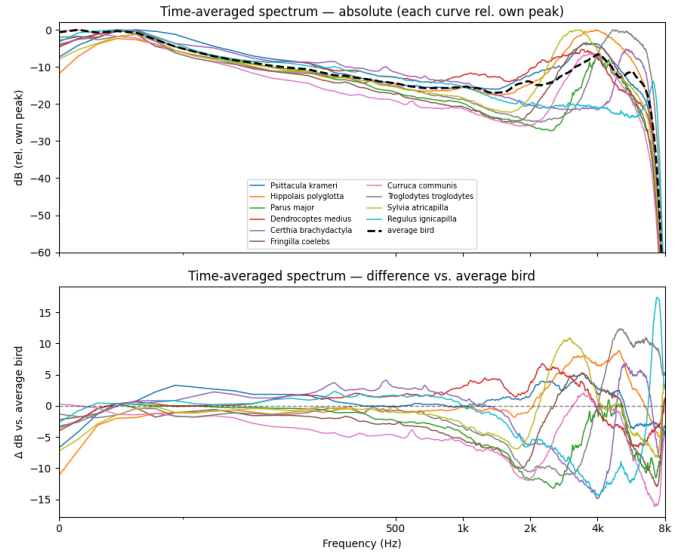


Fig. 1. Time-averaged spectrum of each `paris_10` target species (each curve normalised to its own peak) against the pooled non-target “average bird”. Top: absolute; bottom: difference vs. the average bird. Per-species separation is concentrated above 2 kHz, motivating the `light_mel` 2500–8000 Hz variant.

ducing a 184×80 log-mel spectrogram per 3 s clip. The lighter `light_mel` front-end shrinks every axis at once: a 512-point window with a 480-sample hop and 40 mel bins over 2500–8000 Hz, producing 99×40 . The narrower band comes directly from the data: averaging the spectrum of each target species and comparing it to the pooled “average bird” (Fig. 1), the per-species separation below 1–2 kHz is negligible and almost all of the discriminating energy sits between 2 kHz and 8 kHz, so halving the bin count and dropping the bottom 2.5 kHz costs almost no useful signal. The shorter window and larger hop then cuts the frame count and the FFT size; together with the bin reduction this leaves an input nearly $4 \times$ smaller, a real saving on MCU memory and on the front-end cost (Section V). All log-mels use a $\log \max(x, 10^{-6})$ compression. The time-domain baselines skip this step and read the 16 kHz waveform directly, following end-to-end audio models that classify from raw waveforms and so avoid the spectrogram entirely [7], [8].

III. TINYML MODELS & TRAINING

Every model in this paper has one sigmoid output per target species and is trained with binary cross-entropy. The non-target bucket from Section II is not an extra output channel: a non-target clip is presented to the loss as the all-zeros target vector, and the network has to push every output below threshold. This keeps the head width at exactly N regardless of how the non-target pool is built, and a single model serves both bird-sound detection (any output above threshold) and species classification (argmax over the active outputs, or many birds detected). The width of every convolutional and dense layer in the architectures below scales with a single multiplier m , so the same code defines a one-parameter family of operating points.

A. Mel Frontend

The first TinyChirp [3] architecture used a 184×80 log-mel with a 1024-point Hann window, a 256-sample hop (16 ms at 16 kHz) and 80 mel bins spanning 80–8000 Hz; we keep it as `tinychirp_mel`. Our `light_mel` front-end shrinks every axis at once – a 512-point window, a 480-sample hop (30 ms) and 40 mel bins restricted to 2500–8000 Hz – yielding a 99×40 spectrogram: about $3.7\times$ fewer values and, on-device, 4 to $5\times$ cheaper to compute (Section VI-A).

B. Baseline

The starting point are the three TinyChirp [3] architectures that fit the MCU resource envelope. We leave out its two SqueezeNet variants, which the TinyChirp evaluation reported as out-of-memory on the target microcontroller, and re-implement the rest with two changes from the reference code: the final Softmax becomes a per-class sigmoid, and the channel multiplier m is wired through every channel count and dense width.

CNN-Mel – Two 3×3 Conv2D blocks with ReLU and 2×2 MaxPooling on the `tinychirp_mel` input, flattened into an FC + ReLU head before the N -way sigmoid output. 25 600 parameters at $m=1$.

CNN-Time – Two 1D Conv + ReLU blocks on the 48 000-sample waveform, with 2×1 MaxPooling and SpatialDropout between them, an AveragePooling that collapses the time axis to one, and an FC + ReLU head. 748 parameters at $m=1$.

Transformer-Time – One 1D Conv + ReLU + MaxPooling stem, a global-average pool that emits a single $16m$ -dimensional token, and one pre-norm single-head transformer block (FFN width $32m$), followed by an FC head. 2306 parameters at $m=1$ – slightly above the figure reported in the TinyChirp paper because we match the reference implementation rather than the table in the paper.

C. PolyChirp Birdsong Classification Models

Mel-PolyChirp uses `light_mel`, **DS-CNN** and **WrenNet** reads `tinychirp_mel` with depthwise-separable blocks, and reads the same kind of spectrogram with markedly richer operators; **SincNet** and **LEAF** drop the mel frontend and learn the front-end from the waveform. All five use the same width multiplier m as the baselines.

Mel-PolyChirp – Three 3×3 Conv2D + ReLU blocks ($20m$ channels) with a global average pool and a $32m$ -wide dense head, run on `light_mel`. The goal is to have a model as fast as possible, especially in the case of having the NPU.

DS-CNN [9] – The MLPerf Tiny keyword-spotting reference (Zhang et al.): a stride-2 10×4 Conv2D stem (8 channels) and four depthwise-separable blocks (3×3 depthwise + 1×1 pointwise, batch-norm and ReLU throughout), a global average pool, and the N -way sigmoid. It reads `tinychirp_mel`. Unlike the channel-only architectures it scales compound: depth grows with m and width with $\sqrt{m}$, so parameters still track m^2 .

WrenNet [10] – A recent model for multi-species bird classification on low-power loggers. It reads the

TABLE I
BASELINE ARCHITECTURES. SHAPES ARE REPORTED WITH THE CHANNEL MULTIPLIER m EXPLICIT; N IS THE TARGET SPECIES COUNT.

Layer	Input Shape	Output Shape
CNN-Mel (25 600 params)		
3×3 Conv2D + ReLU	$184 \times 80 \times 1$	$182 \times 78 \times 4m$
MaxPooling	$182 \times 78 \times 4m$	$91 \times 39 \times 4m$
3×3 Conv2D + ReLU	$91 \times 39 \times 4m$	$89 \times 37 \times 4m$
MaxPooling	$89 \times 37 \times 4m$	$44 \times 18 \times 4m$
Flatten	$44 \times 18 \times 4m$	$3168m \times 1$
FC + ReLU	$3168m \times 1$	$8m \times 1$
FC + Sigmoid	$8m \times 1$	$N \times 1$
CNN-Time (748 params)		
3×1 Conv1D + ReLU	$1 \times 48\,000$	$4m \times 48\,000$
MaxPooling	$4m \times 48\,000$	$4m \times 24\,000$
SpatialDropout 0.1	$4m \times 24\,000$	$4m \times 24\,000$
3×1 Conv1D + ReLU	$4m \times 24\,000$	$8m \times 24\,000$
MaxPooling	$8m \times 24\,000$	$8m \times 12\,000$
SpatialDropout 0.1	$8m \times 12\,000$	$8m \times 12\,000$
Average Pooling	$8m \times 12\,000$	$8m \times 1$
FC + ReLU	$8m \times 1$	$64m \times 1$
FC + Sigmoid	$64m \times 1$	$N \times 1$
Transformer-Time (2306 params)		
Conv1D + ReLU	$1 \times 48\,000$	$16m \times 48\,000$
MaxPooling	$16m \times 48\,000$	$16m \times 24\,000$
Dropout 0.25	$16m \times 24\,000$	$16m \times 24\,000$
Global Average Pooling	$16m \times 24\,000$	$16m \times 1$
SingleHeadTransformer	$16m \times 1$	$16m \times 1$
FC + Sigmoid	$16m \times 1$	$N \times 1$

`tinychirp_mel` with the mel bins as channels and is built from PhiNet-style blocks: each block applies a pointwise convolution, a dilated causal depthwise convolution, a squeeze-and-excitation gate, and a pointwise projection, with batch-norm and hard-swish throughout. An initial stage feeds three such blocks and two closing stages, residual connections link the blocks, and a GRU plus a temporal-attention head pool the time axis before the sigmoid output. We keep it to test the limits of the edge path rather than for its footprint. The GRU does not convert to int8 TFLite, where it becomes a custom recurrent op, so the model cannot be baked for microflow or the Axon NPU, and the dilated depthwise convolutions and SE gates lie outside the NPU’s supported CNN operators (Section V). We run it with the GRU replaced by the attention pool, the rest being INT8-compatible.

SincNet [11] – Forty trainable band-pass sinc filters (mel-initialised over 2–8 kHz) replace the mel front-end, feeding two strided Conv2D blocks ($24m$ channels) and a flatten + $16m$ -dense head; a pairwise band-overlap penalty keeps the filters from collapsing during training. The sinc front-end is hard to quantize, so we add BatchNormalization after every convolution and use ReLU6 in place of ReLU, keeping the per-tensor dynamic range tight enough for the int8 PTQ calibration [12] of Section V.

LEAF [13] – Learnable Gabor filterbank, Gaussian pooling, and PCEN [14] compression on the raw waveform, feeding two Conv1D + BN + ReLU + MaxPool blocks ($16m$ channels), a global average pool, and a $16m$ -wide dense head.

Table II lists the layer-by-layer shapes with m explicit; WrenNet and DS-CNN are omitted, as WrenNet’s repeated

TABLE II
POLYCHIRP CLASSIFICATION ARCHITECTURES. m : CHANNEL MULTIPLIER; N : TARGET SPECIES COUNT.

Layer	Input Shape	Output Shape
Mel-PolyChirp		
3 × 3 Conv2D + ReLU	$99 \times 40 \times 1$	$97 \times 38 \times 20m$
Average Pooling	$97 \times 38 \times 20m$	$48 \times 19 \times 20m$
3 × 3 Conv2D + ReLU	$48 \times 19 \times 20m$	$46 \times 17 \times 20m$
MaxPooling	$46 \times 17 \times 20m$	$23 \times 8 \times 20m$
3 × 3 Conv2D + ReLU	$23 \times 8 \times 20m$	$21 \times 6 \times 20m$
Global Average Pooling	$21 \times 6 \times 20m$	$20m \times 1$
FC + ReLU	$20m \times 1$	$32m \times 1$
FC + Sigmoid	$32m \times 1$	$N \times 1$
SincNet		
Sinc Conv (40 filt.)	$1 \times 48\,000$	40×2996
BN + ReLU6	40×2996	40×2996
Average Pooling	40×2996	40×749
8 × 1 Conv2D + BN + ReLU6	40×749	$24m \times 375$
Average Pooling	$24m \times 375$	$24m \times 93$
8 × 1 Conv2D + BN + ReLU6	$24m \times 93$	$24m \times 47$
Flatten	$24m \times 47$	$1128m \times 1$
FC + BN + ReLU6	$1128m \times 1$	$16m \times 1$
FC + Sigmoid	$16m \times 1$	$N \times 1$
LEAF		
Gabor Conv1D (40 filt.)	$1 \times 48\,000$	40×3000
Gaussian Pool	40×3000	40×750
PCEN + BN	40×750	40×750
Conv1D + BN + ReLU	40×750	$16m \times 750$
MaxPooling	$16m \times 750$	$16m \times 375$
Conv1D + BN + ReLU	$16m \times 375$	$16m \times 375$
MaxPooling	$16m \times 375$	$16m \times 187$
Global Average Pooling	$16m \times 187$	$16m \times 1$
FC + BN + ReLU	$16m \times 1$	$16m \times 1$
FC + Sigmoid	$16m \times 1$	$N \times 1$

multi-branch blocks and DS-CNN’s compound depth scaling do not fit the flat per-layer format.

D. Training with Augmentation

All models are trained with Adam and binary cross-entropy on the 70/15/15 split described in Section II, sampled at natural class proportions. Sample weights are set so that the non-target bucket carries half of the total weight mass and the other half is split between the target classes at inverse frequency; pure inverse-frequency collapses the non-target weight when that bucket outnumbered any single target, which in real life is most of the recorded time.

Audio augmentation is applied to the training stream only and runs on the raw waveform before any mel computation, so every front-end sees the same perturbations:

- **Shift** – circular shift, up to $\pm 10\%$ of clip length, $p = 0.5$.
- **Gain** – uniform gain in ± 20 dB, $p = 0.5$.
- **Polarity inversion** – sign flip, $p = 0.5$.
- **Noise** – with $p = 0.5$, one or two of {Gaussian noise at amplitude 10^{-4} to 3×10^{-3} ; AudioSet background mixed at 8–25 dB SNR}.
- **Time masking** – zero a single 2–8% band of the clip, $p = 0.25$.
- **Clipping distortion** – soft clip between the 0th and 5th amplitude percentiles, $p = 0.10$.

IV. TINYML DEPLOYMENT PIPELINE

Sections II and III contribute two reusable parts: a builder that turns a deployment site into a labelled multi-class dataset, and a family of models that read either a log-mel spectrogram or the raw 16 kHz waveform. What a sensor runs in the field is a particular composition of the two, and the same parts compose in more than one way. This section fixes the two single-stage pipelines we deploy and states the questions the evaluation (Section VI) answers about them.

Classification Pipeline – The default pipeline is site-driven and species-aware. A campaign is specified once, off-device, by a coordinate, a radius and a target count N – the species of interest. The builder of Section II pulls and labels that site’s audio into N target classes and one pooled non-target bucket, a model from Section III is trained on it, and the quantized network is flashed to the sensor. In the field the network slides over the stream and emits N sigmoid scores per 3 s window. Because the non-target pool was trained as the all-zeros target (Section III), those scores answer both questions a campaign asks at once: the window is worth storing when any score clears the threshold, and the highest active output names the species. Detection and classification are the same forward pass.

Detection Pipeline – Some campaigns only need to know that a bird was heard, not which one (leaving that to biologists); presence alone decides whether a window is worth keeping. For these the labelling collapses to bird-versus-no-bird and reuses the very same site-specific dataset: every bird clip becomes a positive – the N target species and the pooled non-target birds alike – and only the non-bird material, the AudioSet noise and the no-bird windows, stays negative. Within that positive class a few abundant species would otherwise dominate, so before pooling we cap every species at the same maximum number of clips – up to 250 from each source, 500 in all – which balances the bird set across species rather than letting the most-recorded birds swamp it. The result is a one-output bird-presence detector, the multi-species counterpart of the single-species screen of TinyChirp [3], and the cheapest model we deploy: a smaller head, a single threshold, no per-species calibration. This is a strictly broader question than the classification pipeline answers – a non-target bird is a positive here but the all-zeros target there – so the two are distinct models, not two readings of one. The detector stands alone when species identity is not needed.

V. IMPLEMENTATION OVERVIEW

We target two off-the-shelf low-power boards representative of hardware with/without NPU acceleration: respectively the Nordic nRF54LM20 and the Raspberry Pi Pico 2. Both run the models on their Cortex-M core. The nRF54LM20 additionally provides an Axon NPU. The same trained network thus alternatively use two categories of execution paths: portable CPU paths that runs on either board, or an accelerated NPU path (only on the nRF54LM20).

CPU Paths – On all the hardware, we use a Rust firmware built on Ariel OS [15]. For transpiling models, we use two

approaches. On the one hand, inference handled by microflow-rs [16], which turns a quantized TFLite file into a model at build time. The network is generated directly from the `.tflite`, so the width-multiplier family of Section III compiles into firmware without any hand-written kernels, and the mel front-end (FFT and filterbank) runs in the same Rust path. On the other hand, for comparison, we also implement an alternative CPU path based on Ariel-ML and the IREE transpiler [17]. We chose IREE as comparison point as prior work [18] measured improvements compared to TinyChirp [3]. Thereafter, such CPU paths are simply marked as "IREE".

NPU Path – On the nRF54LM20, we can leverage the Axon NPU using its Edge AI add-on [19]. The add-on’s compiler turns the int8 model (Section V) into a C header that the Axon driver executes, while the mel spectrogram is still computed on the Cortex-M core and only the resulting feature map is handed to the NPU. Note that not every model can take this path. The Axon compiler caps each tensor dimension at 1024 and restricts convolutions to small kernels (filter width up to 32, stride up to 31), and its operator set covers standard CNN blocks – convolution, pooling, fully connected, and the ReLU family – but none of the learnable audio front-ends. The mel-spectrogram models fit this envelope naturally: the spectrogram is a compact image (184×80 for `tinychirp_mel` models, 99×40 for `light_mel` used in `Mel_PolyChirp`), well inside the size limit, and its convolutional blocks map directly onto the NPU. The time-domain models do not. A 48 000-sample waveform already exceeds the 1024-long input limit, and the band-pass filters of SincNet and the Gabor/PCEN front-end of LEAF are long kernels with no counterpart in the supported operators. These models therefore stay on the CPU. In effect the NPU accelerates the spectrogram branch of the pipeline, while the waveform branch – which exists precisely to skip the mel computation – remains a CPU-only option.

Optimization Techniques – After training in float, we apply post-training quantization (PTQ) [12], [20], mapping weights and activations to int8 with per-tensor scales and zero-points calibrated on a sample of the training set. The Axon NPU executes int8 only, and on the Cortex-M core quantizing to int8 arithmetic keeps weights and activation arena inside the RAM budget. As the network infers on a 3 s sliding window over the audio stream, consecutive windows overlap. Exploiting this fact, we can use partial convolution as described in TinyDjävö [21] to limit peak RAM usage. Note that the time-domain models cannot be executed on the NPU (Section V), hence they only have a CPU path.

Mel Front-end – The mel-spectrogram models rest on a log-mel front-end that, on a microcontroller, is a bottleneck in its own right: Nordby measured mel feature extraction at 60 ms against the 38–81 ms taken by the networks it feeds [22], and TinyChirp cites this very cost when it argues for skipping the spectrogram and reading the waveform directly [3]. We keep the front-end on-device but make it cheap enough not to dominate the inference budget, by precomputing everything that does not depend on the incoming audio and reusing

it across every frame and every sliding window. The Hann window is tabulated once at start-up instead of being re-evaluated for each frame, and the triangular mel filterbank is built once from its band edges and stored sparsely, as a flat list of (FFT bin, mel bin, weight) triples that records only the non-zero filter overlaps. Because each FFT bin falls under at most two adjacent triangles, applying the filterbank then becomes a single pass of about two multiply-accumulates per bin, rather than the dense matrix product across all FFT bins and mel channels that a literal filterbank would run. The FFT is the one step we cannot precompute (a 512-point transform for Mel-Polychirp, 1024 for the heavier CNN-Mel): on the nRF54LM20 it is offloaded to the Axon NPU, leaving the Cortex-M core only the windowing, magnitude, filterbank, and log, while a CMSIS FFT keeps the same pipeline running on boards without the accelerator.

VI. EXPERIMENTAL PERFORMANCE EVALUATION

Across all pipelines and models, our evaluation is organised to tackle four axes defined as follows.

- **Accuracy** – Can a single multi-class network both detect a bird and identify which of up to $N = 10$ target species it is, and how much of that accuracy survives the int8 post-training quantization the hardware forces (Section V)?
- **Front-end** – Does reading the raw waveform, and so skipping the mel computation entirely, trade accuracy for a cheaper pipeline against the mel-spectrogram front-end?
- **Budget and hardware** – Does the multi-class setup fit the flash, RAM, latency and energy envelope of a sensor meant to run for a full season, and what does the on-chip NPU buy across the two boards?
- **Pipeline shape** – When only presence matters, does the species-agnostic detector lower the average on-device cost against always running the full classifier?

We report below on measurements we performed on two low-power boards: the Nordic nRF54LM20 and the Raspberry Pi Pico 2. Note that on-device timings are rounded to the nearest millisecond, since the variation between runs is already larger than that.

A. Front-end Preprocessing Cost

We time the mel front-end per 3 s window on both boards, by step, averaged over ten windows (Table III).

Precomputing everything but the FFT (Section V) makes the front-end 7 to 11 $\times$ faster than TinyChirp’s literal per-frame version [3]; the gain is in framing, where tabulating the Hann window once removes a per-sample cosine that is costly on an MCU. `light_mel` is a further 4 to 5 $\times$ cheaper than `tinychirp_mel`, the FFT and magnitude dominating throughout. The Axon NPU runs the 512-point FFT no faster than the Cortex-M (60 vs 58 ms) and cannot take the 1024-point one (Section V), so the front-end stays on the CPU. That the front-end is a real cost, not just inference, matches earlier keyword-spotting work, which reports roughly one second to compute a much smaller (30×40) spectrogram on a more powerful STM32F7 [23].

TABLE III

MEL FRONT-END COST PER 3 S WINDOW, BY STEP (MS); ROWS WITH +NPU: FFT OFFLOADED TO AXON NPU. COLUMN *Energy*: TOTAL PER-WINDOW FRONT-END COST (MJ), MEASURED ON THE nRF54.

Front-end	Target	Frame	FFT	Mag	Mel	Log	Total	Energy
tinychirp_mel	Pico 2	93	119	76	60	13	366	27.6
tinychirp_mel	nRF54LM20	51	202	118	67	34	484	2.25
light_mel	Pico 2	16	29	21	9	3	80	5.31
light_mel	nRF54LM20	13	54	33	14	9	126	0.587
light_mel	+NPU	6	30	5	5	6	53	0.515

TABLE IV

DETECTION PREDICTIVE PERFORMANCE (INT8, F_2 -TUNED THRESHOLD).

Model	Acc.	Prec.	Rec.	F_1	F_2	AUC
CNN-Time	0.79	0.72	0.97	0.83	0.91	0.90
Transformer-Time	0.81	0.74	0.97	0.84	0.92	0.93
CNN-Mel	0.91	0.85	1.00	0.92	0.96	0.97
Mel-PolyChirp	0.91	0.86	0.99	0.92	0.96	0.97
Mel-PolyChirp×2	0.92	0.88	0.99	0.93	0.97	0.98
Mel-PolyChirp×3	0.93	0.89	0.99	0.94	0.96	0.98
SincNet	0.88	0.81	1.00	0.90	0.95	0.97
SincNet×2	0.88	0.82	0.99	0.90	0.95	0.97
WrenNet	0.93	0.89	0.99	0.94	0.97	0.99
DS-CNN	0.92	0.87	0.99	0.93	0.96	0.97

B. Predictive Performance

Per-class thresholds are tuned on train+val, separately for F_1 and F_2 , and reported on the held-out test split; classification metrics are macro-averaged over the target species, detection metrics are read on the single bird-presence class. Everything is INT8 unless stated, at the F_2 operating point, where recall is favoured over precision.

1) *Classification pipeline*: The site-driven classifier grows with the number of target species k , so we report it as a scaling curve. Figure 2 tracks the six TinyChirp [3] metrics – macro F_2 , F_1 , accuracy, precision, recall, AUC – from $k=1$ to $k=10$, each point a mean over resampled species draws.

Quality falls off gradually with k and the models spread apart. The spectrogram models hold up best – wrennet drops only from 0.98 to 0.97 macro F_2 , mel_polychirp_x2 stays above 0.94 – while the time-domain baselines fare worst, CNN-Time and Transformer-Time halving to 0.43 and 0.52. AUC and accuracy stay nearly flat throughout, so the loss is a threshold effect, not lost separability: at the F_2 point precision falls first while recall is held high. The run-to-run spread splits the models the same way: mel_polychirp_x3 and mel_polychirp_x2 hold their macro F_2 standard deviation between 0.001 and 0.006 across $k=1$ to 10, while ds_cnn and mel_cnn vary from 0.013 to 0.200, over 8 resampled species draws per point.

2) *Detection pipeline*: Bird-presence detection is a single binary task, not a scaling one, so we report it once per model (Table IV, INT8 at the F_2 point). Every model clears $F_2 \geq 0.91$ at recall ≥ 0.97 : almost no bird window is dropped. wrennet, the mel_polychirp family and ds_cnn lead, and even the 1–3 KB time-domain baselines reach $F_2 = 0.91$ –0.92. LEAF is excluded (Fig. 3).

TABLE V

CROSS-PLATFORM LATENCY CALIBRATION, CLOCKS: PICO 2 150 MHZ, nRF54 128 MHZ. MF = MICROFLOW TRANSPILER.

Model	Latency (ms)	
	MF Pico2	MF nRF54
CNN-Time	680	2654
CNN-Mel	51	113
Mel-PolyChirp	343	846
Mel-PolyChirp×2	1581	3098
Mel-PolyChirp×3	2115	4948
SincNet	1002	2521
SincNet×2	1710	3235
DS-CNN	599	1566

Quantization to INT8 is important for every model but particularly for the time models : LEAF, CNN-Time, Transformer-Time, and even SincNet even though it is designed to try and counter this. (Fig. 3): float and INT8 test loss differ by under 0.4 everywhere except LEAF, whose loss jumps from 0.19 to 1.33 as the learned PCEN gains do not survive a single activation scale, so it is excluded from the quantized comparisons, since its collapsed network is not useful anymore.

C. Inference Cost and Latency

We deploy every model on the nRF54LM20 and report its inference cost there (Table VI) under each runtime it can target: the microflow INT8 transpiler, with and without its streaming kernel functionality; the IREE ahead-of-time compiler; and the on-chip neural accelerator (NPU). To place those numbers in context we time a handful of models on the RP2350 (Pico 2) as well, giving the cross-platform factors we use to reason about the rest (Table V).

On the models that fit both runtimes, microflow runs about $2.5\times$ faster on the Pico 2 (150 MHz) than on the nRF54 (128 MHz); the $1.17\times$ clock ratio explains only part of this, the rest being a consistent platform factor – memory-access latency, as both boards carry 512 KiB of RAM – that we fold into the calibration.

On the nRF54 itself (Table VI) the NPU dominates – mel_cnn in 2 ms and mel_polychirp_x3 in 39 ms – whereas the transpiler runs from hundreds of milliseconds up to several seconds, and its non-streaming path runs out of memory at the widest width (“OOM”) where streaming, which never materialises the full feature maps, still fits. Note that time-domain models have neither a compiled nor an NPU path (IREE provides no streaming kernel, so the raw 48 000-sample input overflows memory, while the NPU does not take a raw waveform). The LEAF and Transformer-Time models could not be run on any backend : the PCEN mechanism is not runnable on int8 models, while for Transformer time, the first layer (16×48000) is unable to exist without streaming, but microflow currently doesn’t include the attention mechanism. The only time models measured (CNN-Time, SincNet) therefore run under microflow alone – SincNet in both

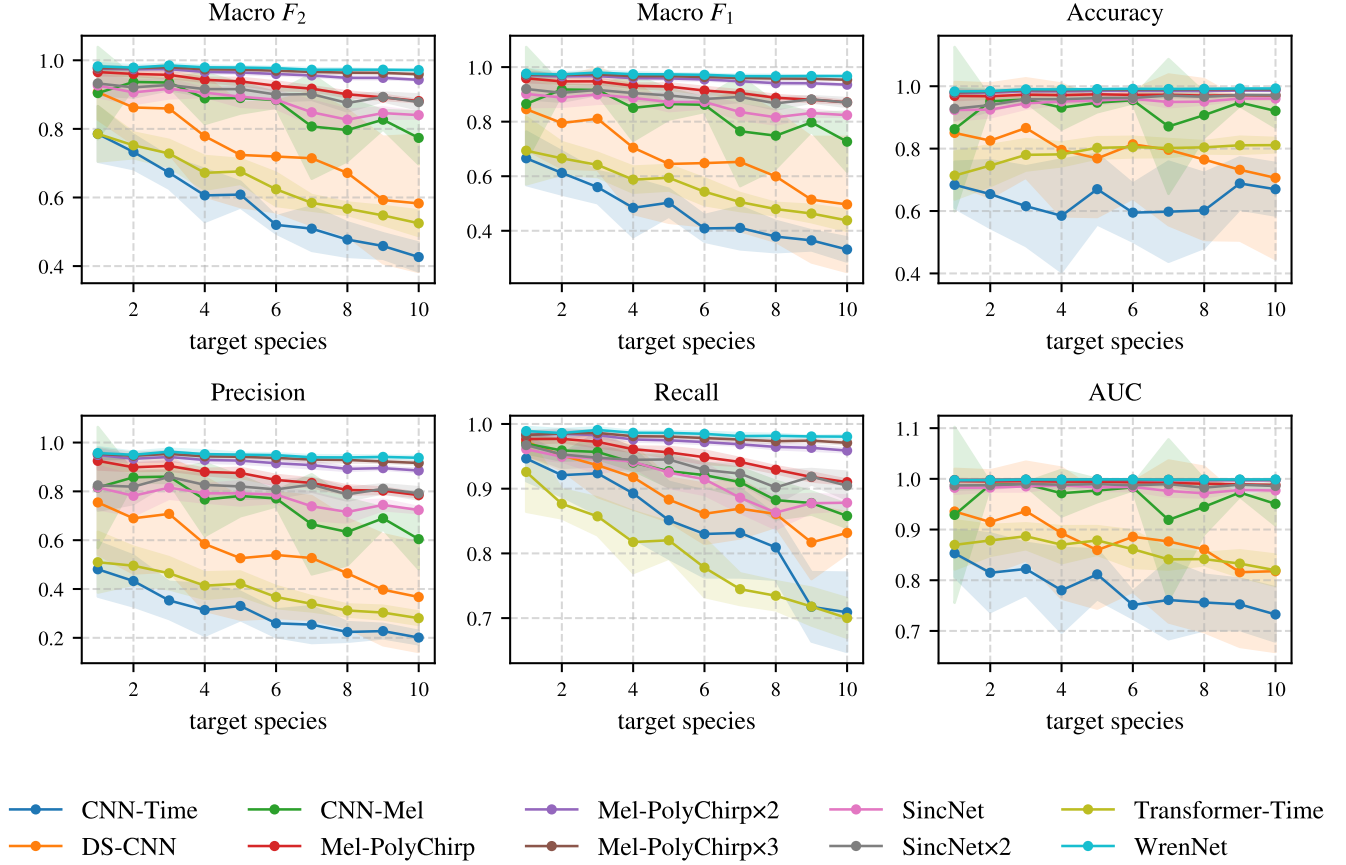


Fig. 2. Classification predictive performance vs. number of target species k (INT8). Precision and recall are read at the F_2 -tuned threshold; each curve is a mean over resampled species draws. LEAF collapses under INT8 (Fig. 3) and is excluded.

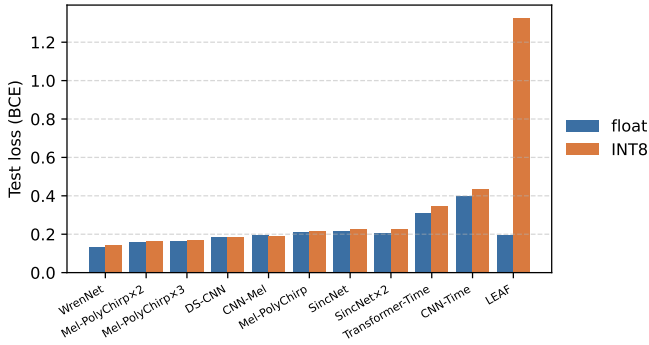


Fig. 3. Float vs. INT8 test loss per model on detection.

modes, while CNN-Time fits only with streaming. Energy per inference (right half of each table) closely tracks latency, along two lines rather than one (Fig. 4): the three runtimes draw comparable power (4.9 mW), so their energy is roughly latency times a shared constant, whereas the NPU sits in a separate regime, drawing about twice that power (9.8 mW) but, finishes inference one to two orders of magnitude faster, thus spends far less energy per inference.

A four-point spot-check on the Pico 2, spanning 74 ms to 1.6 s across inference and preprocessing, finds the effective power (energy over latency) constant at 70 mW to within 14%, so its board-level energy follows a single $E = P \times \text{latency}$ law and needs no per-model table. Measured on the full 3.3 V rail, for the recorded operations the Pico takes 1.3 to $2.2\times$ less time than the nRF54 but draws 6.5 to $12\times$ more energy, making it suitable only where energy is not constrained.

Flash follows the same runtime ordering (Table VI, right): the `microflow` transpiler yields the smallest binaries, while IREE’s ahead-of-time kernels and runtime add roughly 100 kB. All three land above comparable RIOT-OS deployments such as TinyChirp [3], as the footprint is set less by the model than by the Rust-based Ariel OS base firmware, which is heavier than RIOT’s.

VII. DISCUSSION & PERSPECTIVES

Hardware Accelerator (NPU) Considerations – Using an NPU remains somewhat challenging. At the time of writing, this requires using a separate C firmware on the nRF Connect SDK (Zephyr) combined with Nordic’s closed-source Edge AI add-on [19]. The add-on is easy to use, but we experienced downsides. First, this path departs from the single-language

TABLE VI

ON-DEVICE INFERENCE COST ON THE NRF54LM20. ALL MODELS QUANTIZED TO INT8 PRECISION. TIME-DOMAIN MODELS RUN UNDER MICROFLOW ONLY; CNN-TIME FITS ONLY WITH STREAMING. FLASH IS THE BINARY SIZE OF THE DEPLOYABLE FIRMWARE. OOM: MODEL CAN FIT FLASH BUT EXCEEDS AVAILABLE RAM AT RUN TIME; NA: MODEL NOT SUPPORTED BY RUNTIME DUE TO INSUFFICIENT OPERATOR COVERAGE.

<i>Mel-spectrogram models</i>												
Model	Inference Latency (ms)				Energy (mJ)				Flash (kB)			
	IREE	microflow	microflow w/ streaming	NPU	IREE	microflow	microflow w/ streaming	NPU	IREE	microflow	microflow w/ streaming	NPU
CNN-Mel	245	113	536	2	1.12	0.491	2.56	0.0221	187.1	86.9	94.3	182.7
Mel-PolyChirp	671	846	1092	5	3.03	4.28	5.68	0.0469	161.2	55.8	71.4	167.8
Mel-PolyChirp×2	2597	3098	3388	18	12	16	18.5	0.165	183.6	81.8	100.4	198.9
Mel-PolyChirp×3	5483	OOM	4948	39	25.5	OOM	25.8	0.343	226.1	126.0	142.9	245.5
WrenNet	475	NA	NA	NA	2.49	NA	NA	NA	298.3	NA	NA	NA
DS-CNN	628	1566	1963	6	3.08	8.53	10.3	0.0632	182.7	70.6	74.9	157.3

<i>Time-domain models (microflow)</i>						
Model	Inference Latency (ms)		Energy (mJ)		Flash (kB)	
	microflow	microflow w/ streaming	microflow	microflow w/ streaming	microflow	microflow w/ streaming
CNN-Time	OOM	2654	OOM	11.9	OOM	44.9
SincNet	2521	4484	12.1	20.1	64.5	73.9
SincNet×2	3235	5132	15.8	23.5	140.4	151.9

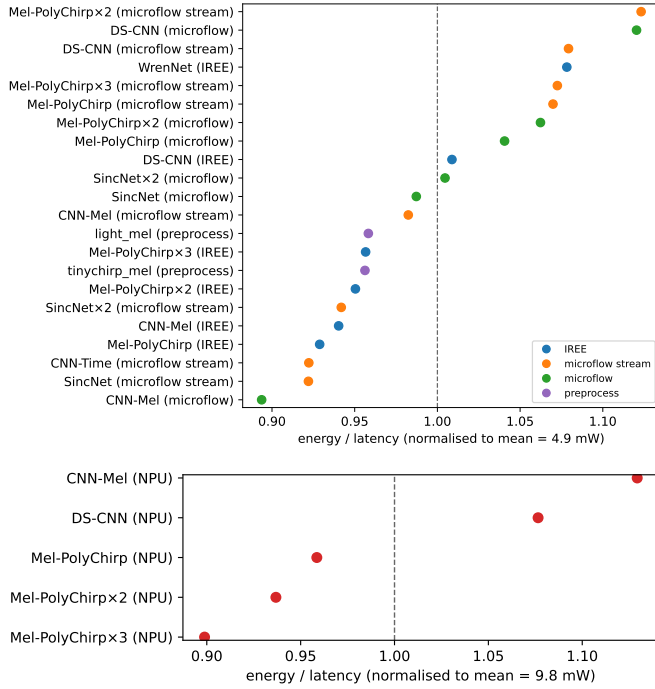


Fig. 4. Effective power (energy over latency) measured on NRF54LM20. Top: on MCU. Bottom: on NPU. Normalised to group mean (dashed line).

(Rust and Ariel OS) stack described in Section V. Second, it ties the deployment to a specific vendor’s hardware. However, the Axon NPU does execute much faster the convolutional part of mel models compared to Cortex-M cores. Inference latency decreases 2 orders of magnitude (from about a second to a few milliseconds, recall Table VI), and inference energy consumption drops proportionally, which is a strong motivation.

TABLE VII

ON-DEVICE INFERENCE LATENCY (MS) ON THE RP2350 (PICO 2). ALL MODELS QUANTIZED TO INT8 PRECISION. OOM: MODEL CAN FIT FLASH BUT EXCEEDS AVAILABLE RAM AT RUN TIME; NA: MODEL SUPPORTED BY RUNTIME DUE TO INSUFFICIENT OPERATOR COVERAGE.

<i>Mel-spectrogram models</i>			
Model	IREE	microflow	microflow w/ streaming
CNN-Mel	107	51	219
Mel-PolyChirp	275	343	453
Mel-PolyChirp×2	883	1581	1392
Mel-PolyChirp×3	3831	OOM	2115
WrenNet	188	NA	NA
DS-CNN	245	599	779

<i>Time-domain models (microflow)</i>		
Model	microflow	microflow w/ streaming
CNN-Time	OOM	680
SincNet	1002	1909
SincNet×2	1710	2161

We observe furthermore how the front-end benefits from the NPU. The accelerator supports ML layers, but also direct calls to operations like FFT. With the network offloaded, the mel computation takes most of the time spent on each window. Waveform models would gain the most from NPU, since they are almost all convolution. However, the NPU cannot run these models, as a 48000-sample input is over the 1024 limit on tensor dimensions, and the Gabor front-end is not supported. Hence, NPU hardware limitations would have to be lifted in next-generation NPUs.

Model Transpilers – The choice of model transpiler (we studied microflow and IREE) is a system-level engineering decision involving trade-offs among inference latency, memory

usage, operator coverage, power efficiency, etc. For instance, compared with microflow, we measured IREE incurs a substantially ($> 2\times$) larger Flash runtime footprint and less memory-safety guarantees, but uses less RAM, achieves less latency and provides support for a broader range of model operators. Orthogonally, given the significant acceleration capability and power efficiency of NPUs we measured, future work should further explore adding NPU support to both IREE and microflow. Future work should also develop further operator support for microflow. Overall, the benchmarks presented in this work may provide useful reference points for practitioners aiming to select a suitable transpiler strategy.

Dataset Limitations – A clip becomes a target only when BirdNET is at least 0.92 confident, which is an arbitrary threshold set from previous work [3]. The models therefore learn to copy BirdNET – labelling mistakes included. The high threshold keeps the labels clean but narrows the data. The clips that pass are the loud, clear ones, not the faint or overlapping calls that fill much of a real-life recording. Distilling from BirdNET’s confidence scores, as WrenNet does [10] would keep some of that information and let us reuse the audio we currently drop. Moreover, on the data side, some target species are under-represented due to lack of available data. Thus, the rarer classes were trained on less data than the common ones. More sound sources, and testing on audio records from the target sensor itself in the field, would help close the gap between this curated set and what the device hears.

Recall versus Precision Trade-offs – A false negative drops a window for good and loses a real detection. A false positive only keeps a spurious clip, and we can filter it out later when the SD card is read. Hence we favour recall over precision, and tune per-class thresholds for F_2 as well as F_1 (see Section VI-B). Quantization complicates this. PTQ to int8 shifts the scores, so a threshold set on the float model must be “retuned” per backend (see Section V). There is no single right threshold, since it depends on how fast the SD card fills and how badly a campaign can afford to miss a species. The detection pipeline uses the cheapest case, with one threshold set for recall and the species left to the offline pass. The harder problem comes earlier. We weight the training set so the loss does not collapse onto the large non-target class (Section III-D), but the field distribution is not the one we trained on, so a threshold that looks right on the test set can behave differently in deployment.

Batching Inference Under a Larger RAM Budget – The limiting factor is often RAM, and the streaming scheme of Section V is there to keep it low. If a board has more RAM to spare, we could instead run several 3 s windows through the network at once. Reusing each weight across a batch, while in cache, would lower the per-call overhead and the energy spent per window. This seems like a low-hanging fruit on larger boards where RAM is not the limiting factor (as this costs extra activation memory).

VIII. RELATED WORK

Birdsong recognition on edge & server hardware –

The main reference for automatic bird-sound recognition is BirdNET [2], a residual network that identifies several thousand species from continuous soundscape audio. Other work on bird audio detection include [24] or [25] using fully convolutional networks trained for multi-species recognition. BirdNET-Pi [26] runs a quantized BirdNET continuously on a Raspberry Pi to log detections in real time, and Bird@Edge [27] streams audio from wireless microphones to a local edge node where a convolutional model performs the recognition. More recent work such as Perch [28] sharpens the underlying representation with general-purpose birdsong embeddings that transfer well across tasks. On the other hand, AudioMoth [29] autonomous recording units records to an SD card, but offloads species recognition to an offline step. Across this domain, the classification runs on microprocessor- or GPU-class hardware.

Birdsong recognition on microcontrollers – Early on-device detectors pair a mel front-end with a compact CNN: Disabato et al. [30] detect birdsong at the edge on a Cortex-M, TinyBird-ML [31] runs vocalization analysis and syllable classification on an animal-borne ultra-low-power node, and Miquel et al. [32] study the energy cost of edge audio for biologging. WrenNet [10] distills BirdNET into a small recurrent network. TinyChirp [3] uses a Cortex-M to screen a single target species in the field while TinyDéjàVu [21] lowers the RAM of streaming inference for such sensors.

TinyML pipelines – TensorFlow Lite Micro [33] is the common C++ interpreter, usually paired on Arm Cortex-M with the CMSIS-NN kernel library [34]. A second approach compiles the model ahead of time: transpilers such as μ TVM [35] or IREE [17] lower a network from the major training frameworks to low-level code for a wide range of microcontrollers. MCUNet [36] combines such an engine with neural architecture search to fit the model and runtime to a given memory budget. Building on these transpilers, U-TOE [37], RIOT-ML [38] or Ariel-ML [18] add embedded operating-system integration, so that arbitrary models can be flashed, benchmarked, and updated over low-power links on commodity boards. Vendor pipelines also include STM32Cube.AI and Edge Impulse. More recent proposals include microflow-rs [16] which compiles a quantized model into memory-safe Rust at build time. Orthogonally, techniques are developed to lower peak RAM usage, for instance msf-CNN [39] which leverages patch-based layer-fusion.

Microcontroller hardware & neural accelerators – The cost and the performance of an ARU depends on the silicon it uses. Popular microcontrollers are based on 32-bit instruction set architectures such as Arm Cortex-M, RISC-V 32 or Xtensa. Alongside such architectures, a class of microcontroller-scale neural accelerators (μ NPUs) has appeared: Arm’s Ethos-U55 [40], the Maxim MAX78000 [41], ST’s STM32N6 with its Neural-ART engine [42], and the Nordic Axon we use on the nRF54LM20 [19]. These NPUs cut inference time and energy by large factors, but run int8 only, cover a restricted set of CNN operators, and so far depend on vendor toolchains, which limits both model coverage and portability

across boards. Standardized TinyML benchmark suites such as MLPerf Tiny [9] target plain MCU cores, while Millar et al. [43] recently provided cross-vendor benchmarks of several μ NPUs and find measured performance often departs from the data sheets.

IX. CONCLUSION

Bio-acoustic monitoring leveraging TinyML pipelines is an approach that is appealing and which has recently become practical. In this paper, we have designed, implemented and evaluated PolyChirp, a low-power embedded software framework which enables training and baking different TinyML models in the firmware of various microcontroller-based hardware, with or without NPU hardware acceleration. Compared to prior work which either required microprocessor-class resources or performed only single-species detection, PolyChirp can classify with high accuracy up to 10 species simultaneously on common microcontroller hardware. We also measure how available NPU hardware acceleration is best leveraged on such hardware. Combined with the portable open source code implementations we published, PolyChirp provides a solid base for both practical deployments and further research in this field.

Code Availability – The pipeline for model benchmarking on Ariel OS is published at github.com/ariel-os/ariel-microflow-ml. The reproducible benchmark code is published at github.com/NathanDuboisset/polychirp.

REFERENCES

- [1] C. Bormann, M. Ersue, and A. Keränen, “Terminology for Constrained-Node Networks,” RFC 7228, 2014.
- [2] S. Kahl et al., “BirdNET: A deep learning solution for avian diversity monitoring,” *Ecological Informatics*, vol. 61, p. 101236, Mar. 2021.
- [3] Z. Huang et al., “TinyChirp: Bird song recognition using TinyML models on low-power wireless acoustic sensors,” in *Proc. IEEE 5th International Symposium on the Internet of Sounds (IS2)*, 2024.
- [4] Xeno-canto Foundation, “xeno-canto: Sharing wildlife sounds from around the world,” <https://xeno-canto.org>, accessed: 2026-06-22.
- [5] J. F. Gemmeke et al., “Audio set: An ontology and human-labeled dataset for audio events,” in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, pp. 776–780.
- [6] Cornell Lab of Ornithology, “Macaulay library,” <https://www.macaulaylibrary.org>, accessed: 2026-06-22.
- [7] S. Abdoli, P. Cardinal, and A. Lameiras Koerich, “End-to-end environmental sound classification using a 1D convolutional neural network,” *Expert Systems with Applications*, vol. 136, pp. 252–263, 2019.
- [8] J. Sang et al., “Convolutional recurrent neural networks for urban sound classification using raw waveforms,” in *Proc. EUSIPCO*, 2018.
- [9] C. Banbury et al., “MLPerf tiny benchmark,” 2021, arXiv:2106.07597.
- [10] S. Ciapponi et al., “Enabling multi-species bird classification on low-power bioacoustic loggers,” 2025, arXiv:2509.20103.
- [11] M. Ravanelli and Y. Bengio, “Speaker recognition from raw waveform with SincNet,” in *Proc. IEEE Spoken Language Technology Workshop (SLT)*, 2018, pp. 1021–1028.
- [12] B. Jacob et al., “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proc. IEEE CVPR*, 2018.
- [13] N. Zeghidour et al., “LEAF: A learnable frontend for audio classification,” in *Proc. ICLR*, 2021.
- [14] Y. Wang et al., “Trainable frontend for robust and far-field keyword spotting,” in *Proc. IEEE ICASSP*, 2017, pp. 5670–5674.
- [15] E. Frank et al., “Ariel OS: An embedded Rust operating system for networked sensors & multi-core microcontrollers,” in *Proc. IEEE DCSS-IoT*, Jun. 2025, arXiv:2504.19662.
- [16] M. Carnelos et al., “MicroFlow: An efficient Rust-based inference engine for TinyML,” *Internet of Things*, vol. 30, p. 101498, 2025.
- [17] IREE, “Intermediate Representation Execution Environment,” <https://github.com/iree-org/iree>, accessed: 2026-06-22.
- [18] Z. Huang et al., “Ariel-ML: Computing parallelization with embedded Rust for neural networks on heterogeneous multi-core microcontrollers,” in *Proc. 8th International Workshop on Intelligent Systems for the Internet of Things (ISIoT)*, 2026.
- [19] Nordic Semiconductor, “Edge AI add-on for nRF connect SDK,” https://docs.nordicsemi.com/bundle/addon-edge-ai_latest/page/index.html, accessed: 2026-06-22.
- [20] M. Nagel et al., “A white paper on neural network quantization,” *arXiv preprint arXiv:2106.08295*, 2021.
- [21] Z. Huang and E. Baccelli, “TinyDéjàVu: Smaller RAM and faster inference with neural networks on MCUs for sensor data streams,” in *Proc. 8th International Workshop on Intelligent Systems for the Internet of Things (ISIoT)*, 2026.
- [22] J. Nordby, “Environmental sound classification on microcontrollers using convolutional neural networks,” Master’s thesis, NMBU, 2019, <http://hdl.handle.net/11250/2611624>.
- [23] J. Wang and S. Li, “Keyword spotting system and evaluation of pruning and quantization methods on low-power edge microcontrollers,” 2022, arXiv:2208.02765.
- [24] D. Stowell et al., “Automatic acoustic detection of birds through deep learning: The first Bird Audio Detection challenge,” *Methods in Ecology and Evolution*, 2019.
- [25] García-Ordás et al., “Multispecies bird sound recognition using a fully convolutional neural network,” *Applied Intelligence*, 2023.
- [26] P. McGuire, “BirdNET-Pi,” <https://github.com/mcguirepr89/BirdNET-Pi>, 2021, accessed: 2026-06-22.
- [27] J. Höchst et al., “Bird@Edge: Bird species recognition at the edge,” in *Proc. International Conference on Networked Systems (NETYS)*, ser. LNCS, vol. 13464. Springer, 2022, pp. 69–86.
- [28] B. Ghani, T. Denton, S. Kahl, and H. Klinck, “Global birdsong embeddings enable superior transfer learning for bioacoustic classification,” *Scientific Reports*, vol. 13, p. 22876, 2023.
- [29] A. P. Hill et al., “AudioMoth: Evaluation of a smart open acoustic device for monitoring biodiversity and the environment,” *Methods in Ecology and Evolution*, vol. 9, no. 5, pp. 1199–1211, 2018.
- [30] S. Disabato et al., “Birdsong detection at the edge with deep learning,” in *Proc. IEEE SMARTCOMP*, 2021.
- [31] L. Schulthess et al., “TinyBird-ML: An ultra-low power smart sensor node for bird vocalization analysis and syllable classification,” in *Proc. IEEE ISCAS*, 2023.
- [32] J. Miquel, L. Latorre, and S. Chamailé-Jammes, “Energy-efficient audio processing at the edge for biollogging applications,” *Journal of Low Power Electronics and Applications*, vol. 13, no. 2, p. 30, 2023.
- [33] R. David et al., “TensorFlow Lite Micro: Embedded machine learning for TinyML systems,” in *Proc. Machine Learning and Systems (MLSys)*, vol. 3, 2021, pp. 800–811.
- [34] L. Lai et al., “CMSIS-NN: Efficient neural network kernels for Arm Cortex-M cpus,” *arXiv preprint arXiv:1801.06601*, 2018.
- [35] T. Chen et al., “TVM: An automated end-to-end optimizing compiler for deep learning,” in *Proc. USENIX OSDI*, 2018.
- [36] J. Lin et al., “MCUNet: Tiny deep learning on IoT devices,” in *NeurIPS*, 2020.
- [37] Z. Huang et al., “U-TOE: Universal TinyML on-board evaluation toolkit for low-power IoT,” in *Proc. IFIP/IEEE PEMWN*, 2023.
- [38] Z. Huang, K. Zandberg, K. Schleiser, and E. Baccelli, “RIOT-ML: Toolkit for over-the-air secure updates and performance evaluation of TinyML models,” *Annals of Telecommunications*, 2025.
- [39] Z. Huang and E. Baccelli, “msf-CNN: Patch-based multi-stage fusion with convolutional neural networks for TinyML,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2025, arXiv:2505.11483.
- [40] Arm, “Arm Ethos-U55: microNPU for embedded machine learning,” <https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u55>, accessed: 2026-06-22.
- [41] Analog Devices, “MAX78000,” <https://www.analog.com/en/products/max78000.html>, accessed: 2026-06-22.
- [42] STMicroelectronics, “STM32N6 series,” <https://www.st.com/en/microcontrollers-microprocessors/stm32n6-series.html>, accessed: 2026-06-22.
- [43] J. Millar et al., “Benchmarking ultra-low-power μ NPUs,” 2025, arXiv:2503.22567.