# Adapting Mamba Models for Deployment on Microcontrollers

## Enabling Linear-Time Sequence Modeling on Ultra-Low-Power Tiny Devices

**Bartosz Drabiński**[1]

**Supervisor: Bo Yang**[1]
**Responsible Professor: Qing Wang**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Bartosz Drabiński
Final project course: CSE3000 Research Project
Thesis committee: dr. Qing Wang, Mark Neerincx

## Abstract

As machine learning expands into diverse domains, TinyML has emerged as a crucial paradigm for deploying models on highly resource-constrained microcontrollers, which typically feature less than 256 KB of RAM. However, executing complex mathematical operations on these devices remains a significant challenge, necessitating novel model designs and hardware-aware optimization. The Mamba architecture, built around State-Space Model, is a promising candidate due to its compact parameterization and strong performance on long-context tasks. Nevertheless, Mamba was originally designed for highly parallelized GPUs, making its adaptation for TinyML non-trivial. This paper evaluates Mamba deployment strategies on microcontrollers using TensorFlow Lite Micro. We propose architecture modifications and optimization techniques tailored specifically to microcontroller constraints. Our deployment of a quantized Mamba model achieves a 60.4 KB peak RAM footprint on a Keyword Spotting task, a 74% memory reduction compared to state-of-the-art work (MambaLite-Micro). Furthermore, we analyze the trade-offs of quantization, demonstrating that while it substantially reduces memory, it can introduce latency overhead on hardware lacking acceleration of INT8 operations. To mitigate code size and loop-unrolling overheads, we introduce a model-splitting technique that enables the execution of larger models. Our findings demonstrate that while Mamba is a viable architecture for TinyML, further research is required to fully optimize State Space Model implementations for edge hardware.


## 1 Introduction

Edge AI aims to improve latency, privacy and decentralize machine learning inference by running models directly on user devices. TinyML [1] focuses on the extreme end of this spectrum, designing models that run on microcontrollers (MCUs). MCUs are severely resource-constrained: typical devices provide one or two CPU cores (usually clocked below 240 MHz) and often 256 KB or less of RAM, preventing them from hosting large models or full-precision weights. Furthermore, low latency is essential because many TinyML applications, such as audio or video streams, require real-time processing.

The extreme memory limitations of microcontrollers make architectural designs with super-linear memory scaling poorly suited for TinyML applications. Consequently, convolutional neural networks (CNNs) and recurrent neural networks (RNNs) have traditionally remained the dominant choices due to their superior parameter efficiency relative to standard multilayer perceptrons (MLPs) or Transformer-based models [2]. However, the recently proposed Mamba architecture [3] has emerged as a promising alternative to Transformers for long-context tasks, blending convolutional and recurrent design ideas to capture long-range dependencies while maintaining a linear memory footprint. A high-level diagram of this Mamba block architecture is presented in Figure 1.

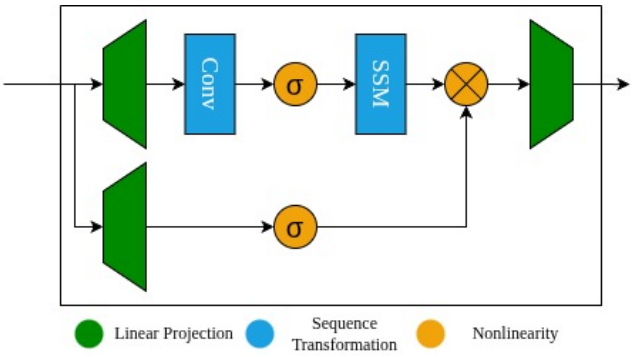



Figure 1: Diagram of the Mamba block.

Mamba's main building block is a Selective State-Space Model (S6) layer, which captures long-range structure while avoiding the quadratic complexity of attention.

In Mamba, the S6 matrices are conditioned on the input, allowing the hidden state dynamics to adapt dynamically over time. S6 layers are typically implemented using efficient scan (parallel-prefix) algorithms that parallelize well on modern hardware. However, their recurrent-style execution can make them highly sensitive to quantization and numerical instability [4; 5].

Prior work has explored common optimizations for Mamba such as quantization [4], pruning, and knowledge distillation [6]. Notably, MambaLite-Micro [7] implemented PyTorch-style inference for Mamba directly in C and evaluated execution on ESP32-S3 and STM32 microcontrollers. While this verified basic execution feasibility, the implementation is highly model dependent, making deep architectural changes and hardware-specific optimizations cumbersome to explore.

This paper addresses the following overarching research question: ***How can Mamba architectures be adapted and optimized for efficient inference on resource-constrained microcontrollers?***

To address this comprehensively, we formulate three specific sub-questions:

- **RQ1:** What are the primary memory and computational bottlenecks when deploying Mamba models on microcontrollers?
- **RQ2:** How do standard quantization strategies affect Mamba's inference latency, memory footprint, and task accuracy on these edge devices?
- **RQ3:** To what extent can hardware-unfriendly components of the Mamba architecture (e.g., specific activation functions) be simplified or replaced to fit microcontroller constraints, and what are the resulting trade-offs in performance?

To answer these questions, we profile deployment strategies on actual hardware, isolate the underlying hardware bottle-

necks, and evaluate the efficacy of translating typically GPU-centric optimizations to microcontroller architectures.

The remainder of this paper is organized as follows. Section 2 surveys related work and foundational background. Section 3 details our proposed optimizations and architectural modifications, while Section 4 outlines how the experiments were conducted. Experimental results are presented in Section 5. Their discussion and limitations are described in Section 6. Finally, Section 7 addresses ethical considerations, and Section 8 concludes the paper with directions for future work.

## 2 Related Work and Background

This section reviews TinyML constraints and recent Mamba literature relevant to MCU deployment. We summarize common TinyML optimizations—focusing on quantization, pruning, and distillation—and detail prior deployment attempts that motivate our contributions.

### 2.1 TinyML Constraints and Optimization Strategies

Given the limited computational throughput of MCUs, minimizing latency is a critical operational priority for real-time edge processing (e.g., audio or accelerometer streams). Many complex deep learning architectures rely on operations such as exponentiation (e.g., Softmax, SiLU) or trigonometric functions (e.g., Tanh). On microcontrollers, computing these functions non-natively introduces substantial execution overhead. Replacing them with hardware-friendly alternatives like ReLU can provide up to a 25$\times$ computational speedup under resource-constrained conditions [8].

The lack of robust or fast hardware floating-point units (FPUs) on many MCUs necessitates post-training or aware quantization. Quantization reduces the precision of weights and activations from 32-bit floating-point (FP32) formats to lower-bit fixed-point representations, such as 8-bit or 4-bit integers (INT8/INT4). While this strategy substantially reduces the model's memory footprint, these gains often incur an accuracy penalty. Furthermore, the final memory savings are not always directly proportional to the reduced bit-width due to underlying software graph and format conversion overheads [9]. Careless precision reduction can yield severe accuracy drops, particularly for neural network layers that were not structurally designed with quantization bounds in mind [4].

Other optimization mechanisms include network pruning, knowledge distillation, and low-rank factorization [10]. Pruning algorithms eliminate redundant weight channels, whereas knowledge distillation transfers latent representations from a larger teacher network into a compact student model. Although highly effective, applying these methods to alternative structures is often complex, and achieved compression ratios vary significantly depending on the target task and network layout [6].

### 2.2 Mamba Architecture and Edge Deployment Efforts

Most prior literature exploring Mamba model compression has focused on mobile or embedded GPUs rather than MCU-class systems. For example, the Physics-Guided Tiny-Mamba Transformer [11] compresses its topology down to $\sim$0.5M parameters for deployment on the NVIDIA Jetson Nano; Tiny-ViM [12] targets vision tasks at a scale of $\sim$5M parameters; and FEMBA [5] utilizes aggressive quantization targeting a $\sim$7.8M parameter model. While these works demonstrate that standard optimizations like quantization, pruning, and distillation are viable for State-Space Models, the resulting architectures remain orders of magnitude too large for typical microcontroller memory budgets.

Efforts such as LightMamba [13] explore Selective State-Space Models for specialized, resource-constrained tasks like ultra-wideband positioning, but they stop short of evaluating physical microcontroller deployments. MambaLite-Micro [7] is the only previous work that we are aware of that actually achieves successful execution on a microcontroller. It uses ESP32-S3 and STM32 MCUs and a custom C mamba implementation, but its strict adherence to PyTorch-style kernels complicates deep architectural modifications. Consequently, deployment tooling and structural co-design methodologies for Mamba models on MCUs remain immature.

### 2.3 Research Gap

Two primary gaps motivate this work. First, while many TinyML optimization frameworks exist, their algorithmic behavior and stability when applied to Mamba's input-dependent recurrence are not well understood on MCU-class hardware. Second, while initial C-based engines demonstrate basic execution feasibility, they do not explore explicit graph modifications or hardware-aware activation substitutions to mitigate MCU computational bottlenecks. This paper bridges these gaps by profiling physical hardware deployments, isolating structural bottlenecks, and evaluating targeted architectural changes to make Mamba models truly viable on microcontrollers.

## 3 Proposed Mamba-MCU Optimizations

This section details the deployment framework and architectural adaptations developed to enable efficient Mamba inference on microcontrollers. We outline our hardware and toolchain selection criteria, describe the custom model re-implementation pipeline, and introduce three targeted optimization strategies: hardware-aware activation simplification, post-training integer quantization, and a modular loop-partitioning scheme to mitigate compilation unrolling.

### 3.1 Embedded Deployment Toolchain and Hardware Selection

Deploying state-space models on resource-constrained microcontrollers introduces significant engineering trade-offs. While MambaLite-Micro [7] ensures that the output matches the PyTorch model output in 100% of cases by hardcoding entire inference in C, this approach severely restricts flexibility. Any architectural modification or optimization like replacing activation functions or introducing quantization requires manual code modifications to the low-level source code, hindering rapid experimentation.

To overcome this limitation, this work establishes an automated, hardware-agnostic compilation pipeline. We evaluated contemporary edge runtime environments, including TensorFlow Lite for Microcontrollers (TFLM, recently being rebranded into LiteRT) [14] and ExecuTorch [15]. TFLM was selected due to its mature ecosystem, robust documentation, and flexible runtime interpretation. By mapping operations directly to the TFLM runtime, this workflow eliminates intermediary conversion formats (e.g., ONNX), thereby isolating the experimental results from unnecessary intermediary model representations.

The deployment infrastructure leverages several utility tools embedded within the TFLM ecosystem. The compiler utility `tflm_model_transforms` strips redundant metadata and debugging nodes from the intermediate representation graph, isolating the raw architectural parameters and facilitating a rigorous, equitable comparison of binary model sizes. For precision reduction, we deploy the AI Edge Quantizer, which provides fine-grained, programmatic configuration over tensor bit-widths directly on the compiled flatbuffer files. The comprehensive deployment pipeline is structured as illustrated in Figure 2.

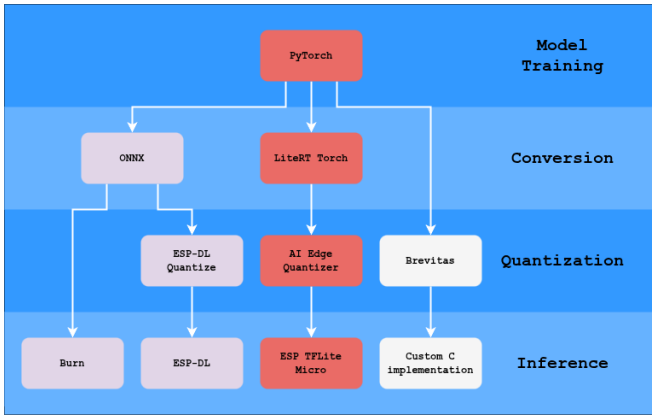



Figure 2: Diagram of the TinyML deployment pipeline. The specific optimization path implemented in this work is highlighted in red.

Alternative model compression libraries were also considered, such as Brevitas [16] (leveraged in FEMBA [5]) and TorchAO [17]. While these frameworks provide robust support for Quantization-Aware Training (QAT) and ultra-low mixed-precision bit-widths, they frequently assume low-level kernel fusion capabilities. Such optimizations are challenging to synthesize efficiently on generic microcontrollers without specialized hardware abstractions.

Physical hardware evaluation is conducted on the Espressif ESP32 family of microcontrollers (Figure 3), specifically the ESP32 and ESP32-S3 modules, using the vendor-optimized `esp-tflite-micro` component library. To ensure the generalizability of our findings across competing hardware platforms, we explicitly avoid vendor-exclusive deployment pipelines such as ESP-DL or STM32Cube.AI.

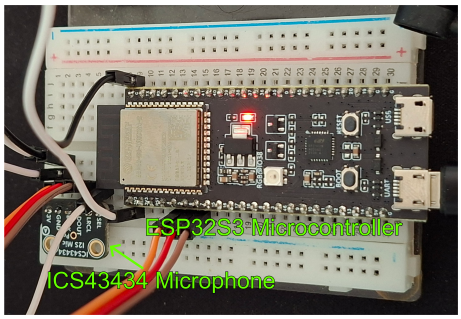



Figure 3: Experimental setup featuring an ESP32-S3 microcontroller coupled to an ICS43434 microphone module.

### 3.2 System Design and Architectural Adaptations

To maintain empirical continuity and isolate the performance impacts of our deployment optimizations, we adopt the core structural dimensions of the MambaLite-Micro baseline [7] rather than performing an exhaustive hyperparameter grid search. The reference network consists of a fully connected projection layer, a single Mamba block ($d_{\text{model}} = 64$, expansion factor $E = 2$), a global average-pooling layer, and a linear output classification head.

Because standard Mamba implementations rely heavily on custom Triton or CUDA GPU kernels that lack native equivalents in edge runtimes, we re-implemented the network graph using standard, modular operators in PyTorch. This re-implementation provides malleable graph structures allowing us to swap mathematically intensive, exponentiation-heavy activation functions (e.g., SiLU) for hardware-friendly primitives (e.g., ReLU) prior to model compilation.

Following initial profiling cycles, the state-space formulation and transcendental activation layers emerged as the dominant latency and memory bottlenecks. In our work we focus on those two main issues, by simplifying the activation functions and suggesting a solution for the SSM layer.

Furthermore, initial deployment trials exposed a systemic limitation within the TFLM translation layer: the absence of native recurrent loop primitives caused the execution graph of the Selective State-Space Model to compile as a completely unrolled sequence. For long-context workloads, such as the Google Speech Commands V2 dataset, this unrolling behavior triggers an exponential explosion in model binary size and completely breaks execution due to constraints on the `CONCATENATE` operator nodes.

To resolve this sequence scaling ceiling, we introduce a modular loop-partitioning design pattern. The model is decoupled into three independent execution stages: pre-SSM projection phase, SSM time-step kernel, post-SSM output phase.

The host microcontroller code executes an explicit iterative loop around the single-step SSM kernel component, sequentially cycling through the inputs while maintaining the hidden state vector in native memory. This approach eliminates graph duplication by recycling a single set of network weights across all time steps. To quantify the execution trade-offs inherent to this approach, we bench-test both the unrolled and loop-partitioned implementations on the Human Activ-

ity Recognition (HAR) dataset [18]. Notably, the unrolled baseline fails to run on the long-sequence Google Speech Commands V2 [19] workload because of the graph size (after 51 sequence step) unrolling being too large. This makes the loop-partitioned paradigm a strict architectural prerequisite for complex audio tasks.

### 3.3 Showcase Demonstration: Keyword Spotting on ESP32-S3

To demonstrate the practical viability of our proposed optimizations, we deploy a quantized Mamba model on an ESP32-S3 microcontroller for a Keyword Spotting task. The model is trained on the Google Speech Commands V2 dataset. It is built on top of the tensorflow micro speech example for efficient preprocessing. The audio gathering is done using an ICS43434 microphone module. Data gathering and inference runs fully on the microcontroller and it is able to process audio samples two times per second.

## 4 Experimental Setup and Evaluation

This section describes datasets, preprocessing, evaluation metrics, and measurement procedures used to quantify accuracy, peak RAM, and latency. We provide enough detail to allow replication of conversion, quantization, and microcontroller benchmarking steps.

### 4.1 Dataset and Preprocessing

Many datasets have been used and demonstrated as viable for TinyML applications; these typically use sensor data such as audio, images, or accelerometer signals. For this work we use two datasets: the HAR dataset and the Google Speech Commands dataset.

These datasets are popular in the TinyML community and represent different data modalities (accelerometer and audio), which enables comparison with prior work such as MambaLite-Micro [7].

The HAR dataset consists of recordings of volunteers performing activities of daily living while carrying a waist-mounted smartphone with embedded inertial sensors. The data is preprocessed into 561 features. Following previous work, we pad this input to 570 features and split them into 10 time steps of 57 features each, so that it matches the format used in prior work. State-of-the-art models can achieve nearly 98% accuracy on this dataset [20].

For the Google Speech Commands dataset we preprocess audio into Mel-frequency cepstral coefficients (MFCCs). We extract 40 MFCC features per time step, producing 51 time steps with 40 features each. While some implementations reduce the number of classes from 35 to 10, we kept all 35 classes to exercise long-sequence behavior. State-of-the-art models can achieve around 96% accuracy on this dataset [21].

### 4.2 Model Training and Conversion

The experimental design evaluates parts of the deployment pipeline: model architecture, optimization strategies, and deployment choices. We measure both peak memory usage and latency, which are critical for TinyML.

For both datasets we use the following hyperparameters: Hidden size: 64; Expansion ratio: 2; State size: 16; Batch size: 32; Learning rate: 0.002 (drop to 0.001 after 10 epochs); Epochs: 20.

The size of the model was chosen to be comparable to the MambaLite-Micro implementation. We do not perform extensive hyperparameter search, since our focus is on deployment and optimization rather than maximizing accuracy.

The model is trained using the Adam optimizer with a learning rate of 0.002 and a step learning rate scheduler that reduces the learning rate by half every 10 epochs. We train for 20 epochs, which is sufficient for convergence on these datasets.

We train the model in PyTorch and then convert it to TFLite using the TFLM converter. Then, we apply quantization using the AI Edge Quantizer. We only evaluate static post-training quantization. We do this with 2000 samples from the training sets of the two datasets. We found that increasing the number of samples further did not significantly change quantization results.

### 4.3 Evaluation Metrics and Measurement Procedures

Measuring peak memory and latency on an embedded device is challenging, but TFLM provides tools to help. For memory usage, we use the TFLM memory recorder to measure peak memory and to separate permanently allocated versus temporary buffers. For latency we adapted the default TFLM profiler to measure total inference time and time per operation type, which helps identify bottlenecks and the impact of optimizations.

For accuracy testing we run the full test set on the development PC for both the PyTorch implementation and after conversion to TFLite to ensure conversion does not introduce significant degradation. We also measure accuracy after quantization to evaluate its impact.

For testing on the microcontroller we chose the ESP32 (ESP32-WROOM-32), ESP32S3 (ESP32-S3-DevKitC-1-N32R16V). This allows us to focus on one hardware family while still comparing a basic model (ESP32) to a more powerful one (ESP32-S3). It also allows us to compare with MambaLite-Micro, which also uses ESP32-S3.

On the microcontroller we evaluate accuracy on a small subset (50 samples) from each test set due to resource constraints, which provides confirmation that the final embedded implementation matches expected behavior.

The code for training, conversion, and microcontroller benchmarking is available at https://github.com/drabart/MCUFriendlyMamba.

## 5 Results

This section presents our empirical findings across four primary metrics: classification accuracy, compiled model size, hardware latency, and peak runtime memory consumption. Direct analytical interpretations of these metrics are integrated into each respective results section, followed by a consolidated discussion of architectural trade-offs, system constraints, and broader structural limitations.

### 5.1 Model Accuracy and Storage Fingerprint

We first validated the baseline classification performance on a development PC using our custom PyTorch implementation of the Mamba block. Training achieved validation accuracies of 98.6% on the HAR dataset and 90.09% on the KWS audio task. Following export to the TensorFlow Lite (`.tflite`) flatbuffer format, accuracy was profiled before and after uniform `INT8` post-training quantization across both full (unrolled) and split topologies (Table 1).

Table 1: Accuracy results for full and split models.

| Dataset | Float32 | Int8 |
|---|---|---|
| HAR PyTorch | 93.89% | - |
| HAR TFLite Full | 93.89% | 93.59% |
| HAR TFLite Split | 93.89% | 92.37% |
| KWS PyTorch | 88.96% | - |
| KWS TFLite Full | 88.96% | 85.06% |
| KWS TFLite Split | 88.96% | 84.10% |

The empirical accuracy trade-offs demonstrate distinct, topology-dependent sensitivities to low-bit fields. While the HAR model preserves competitive fidelity under quantization, the KWS models experience a sharper performance drop of nearly 4 percentage points under `INT8` precision. This divergence reinforces known complexities in quantizing SSMs without custom scaling layers [4].

Crucially, the data reveals that split (looped) configurations suffer higher quantization-induced accuracy drops than their unrolled counterparts. This behavior is structurally determined: because an unrolled computational graph handles each sequential step as an independent tensor node, the static post-training quantization algorithm can compute separate, granular calibration scales and zero-points for individual instances of identical operators. Conversely, our split topology forces every iterative timestep to share a singular, globally calibrated set of quantized parameters, causing rounding errors to accumulate aggressively across the recurrent SSM boundary.

Table 2: Model sizes before and after quantization and optimization.

| Dataset | Original | Quantized | Optimized |
|---|---|---|---|
| HAR Full | 195 KB | 102 KB | 70.5 KB |
| HAR Split* | 160 KB | 62 KB | 50.9 KB |
| KWS Full | 319 KB | 267 KB | –** |
| KWS Split* | 165 KB | 63 KB | 51.9 KB |

*Sizes of the 3 partial models have been added up

**Concatenation operation too long to handle optimization.

Parallel to model precision, we tracked binary storage footprints across individual optimization steps (Table 2). In edge environments, raw weight dimensions alone do not determine true storage profiles; framework metadata and node definitions frequently introduce graph serialization overhead. Stripping redundant debug and name descriptors via `tflm_model_transforms` successfully yielded up to a $\sim$30% reduction in final file size.

### 5.2 Embedded Performance and Structural Bottlenecks

To establish low-level optimization targets, we isolated the execution latency assigned to individual operation types within a standard Mamba layer (Figure 4). Profiling indicated that native exponential instructions impose a severe latency penalty on bare-metal MCUs that lack hardware-level transcendental acceleration blocks.

To mitigate this bottleneck, we structured three comparative floating-point models under varying configurations of transcendental replacement. *float noopt* - preserves all original functions. *float* - replaces the SiLU and Softmax functions with hardware-friendly ReLU primitives. *float opt* - additionally substitutes the analytical expression $\overline{A} = \exp(\Delta_t \cdot A)$ with a first-order Euler discretization $\overline{A} \approx 1.0 + \Delta_t \cdot A$. The resulting latency and accuracy behaviors are detailed in Figure 5 and Table 3, respectively.

Table 3: Accuracy results for float HAR models with varying amount of exponentiation removed.

| Model | Float32 | Int8 |
|---|---|---|
| float noopt | 94.06% | 93.42% |
| float | 94.64% | 94.67% |
| float opt | 93.76% | 89.96% |

The *float* model variant achieved superior classification fidelity while bypassing expensive non-linear calculations. Conversely, the heavily optimized Euler discretization variant (*float opt*) exhibited high variance and severe accuracy degradation when paired with subsequent quantization layers. Consequently, the *float* configuration was chosen as our structural optimization baseline and was used for the remainder of experiments.

A major finding of our on-device profiling is that the `INT8` quantized implementations are significantly slower than their `FLOAT32` equivalents, with integer execution times nearly doubling across identical layers (Figure 6). This counter-intuitive execution behavior is explained by the operation breakdown in Figure 4. The runtime graph is completely dominated by element-wise multiplication (`MUL`) instructions. Although the ESP32-S3 contains hardware-level vector extensions capable of accelerating integer arithmetic, the standard, non-fused TFLM runtime infrastructure introduces extensive quantization/dequantization scaling and memory reindexing overhead around each discrete scalar multiplication node. This software wrapper overhead ultimately overrides the raw clock-cycle efficiency of the integer execution unit.

Furthermore, platform optimization libraries demonstrated notable instability; activating vendor-specific `ESP_NN` kernel accelerations caused measured accuracy on the HAR task to drop from 96% to 56%, due to a bug in the vendors implementation (we have not been able to find its exact source). Because this issue was strictly isolated to this specific layer graph, a separate baseline code path (INT8 ANSI) was pro-

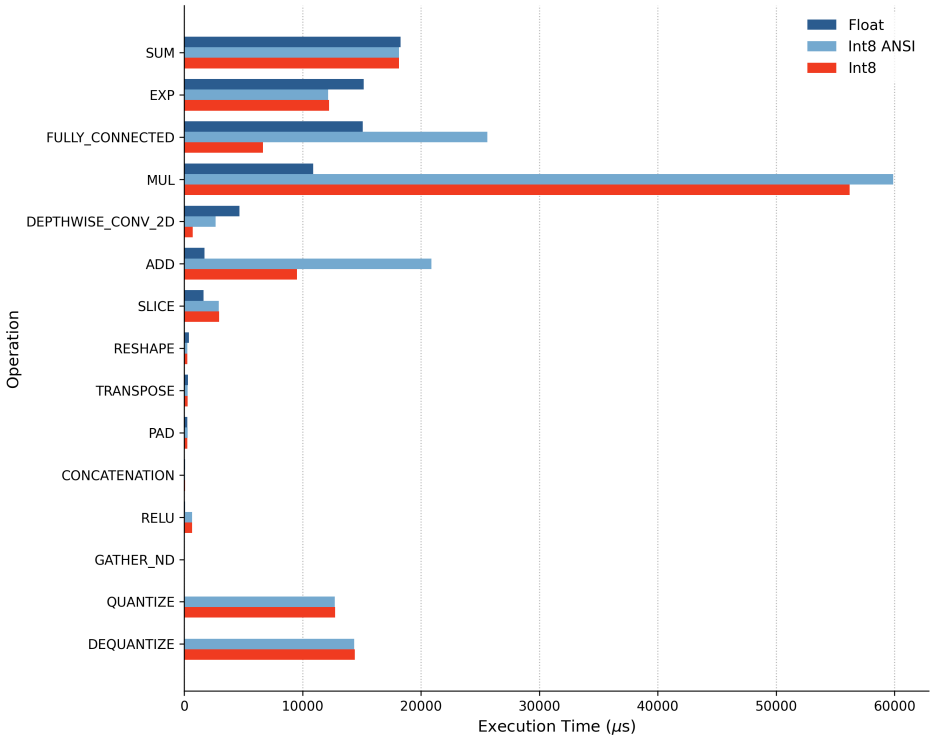



Figure 4: Granular execution impact per instruction type across Float, INT8 ANSI, and INT8 HAR configurations.

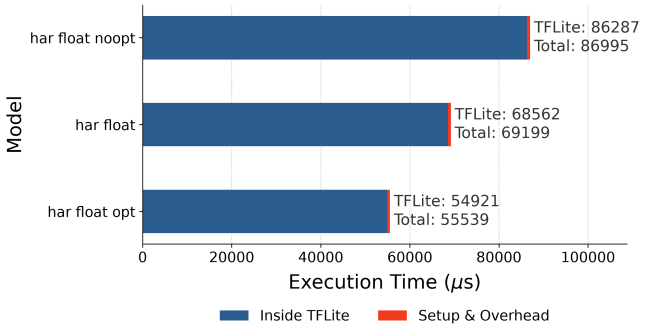



Figure 5: Execution latency across alternative levels of non-linear operation removal within the floating-point HAR architecture.

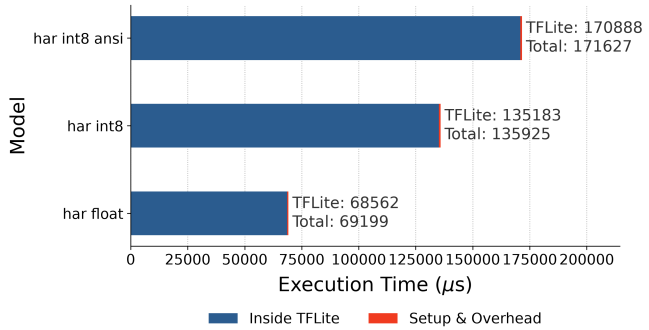



Figure 6: On-device execution latency comparisons for the unrolled HAR topology.

filed to isolate the compiler bug and showcase performance without any hardware acceleration.

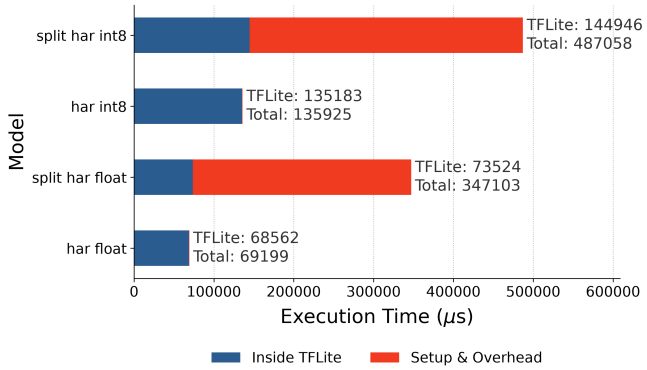
Finally, we evaluated the performance profiles of our proposed model-splitting design pattern against the standard unrolled paradigm (Figure 7). Splitting the model introduces a massive latency penalty, directly multiplying overall execution duration. This overhead is fundamentally external to the TFLM interpreter kernel itself; it is driven by host-side memory orchestration, where data pointers must be manually copied, reshuffled, and verified across three separate subgraph interfaces at every consecutive time step.

While splitting the graph incurs a substantial speed penalty, it successfully addresses memory bottlenecks. Volatile memory usage behaves predictably: INT8 variants demand significantly smaller heap allocations (Figure 8).

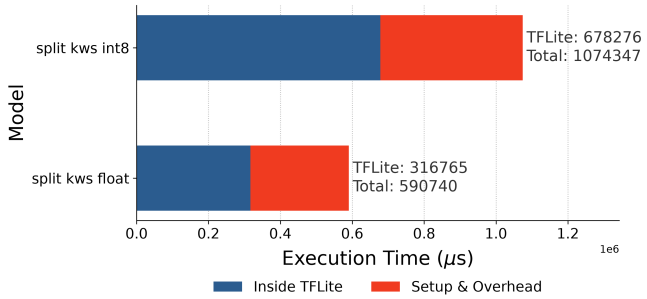
Crucially, the loop-partitioned model reduces peak working RAM allocations relative to the unrolled network, despite requiring additional static data buffers to manually bridge the sub-graph interfaces. The specific scaling dynamics of these intermediate transfer matrices vary linearly based on input tensor dimensions (Figure 9).

## 6 Discussion

The empirical evaluation proves that while Mamba configurations can deliver high classification performance within tight microcontroller constraints, yielding ∼94% accuracy on HAR and ∼89% on KWS, their viability remains tightly coupled to specific compiler behaviors.




(a) Latency comparison between unrolled and split HAR configurations.




(b) Latency profile of the partitioned KWS architecture.

Figure 7: Detailed latency benchmarks across isolated microcontroller execution strategies.

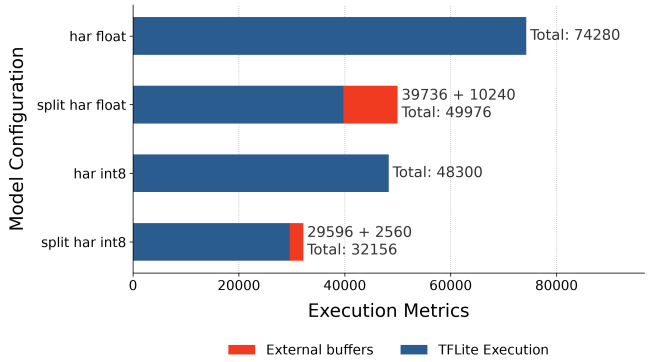



Figure 8: Peak working RAM consumption across alternative HAR model formats.

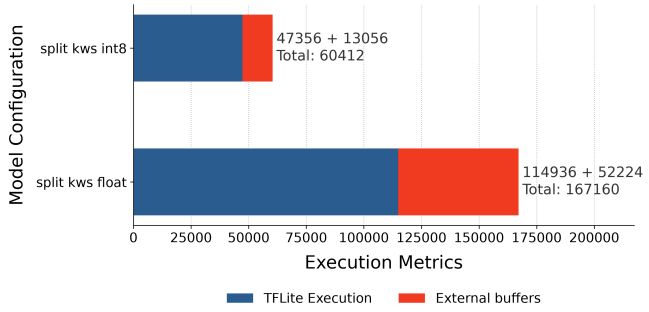



Figure 9: Peak working RAM allocations mapped across varying KWS sequence dimensions.

Our finding that model segmentation reduces peak working memory points toward an important architectural insight. We hypothesize that this memory reduction is an artifact of TFLM's internal arena allocation mechanism, which employs a greedy heuristic to schedule lifetime tensor offsets. By partitioning a single sequence graph into three smaller, isolated flatbuffer files, we effectively divide the tensor scheduling space into distinct subsets. This segmentation enables the greedy heuristic to achieve a more globally optimal memory layout than is possible when optimizing a massive, unrolled sequence graph. If this hypothesis holds, memory-saving benefit will be highly sensitive to sequence topology and may not scale uniformly across different neural layer combinations.

From a structural standpoint, model segmentation serves as an essential enablement strategy rather than a pure performance optimization. Without this manual loop-partitioning scheme, deploying Mamba models for moderately long sequences is completely unfeasible under current TinyML compilers. This operational boundary was explicitly encountered during KWS compilation: the standard unrolled framework failed execution entirely due to internal TFLM operator constraints, which restrict the native `CONCATENATE` node to a maximum of 10 input dimensions, whereas the unrolled KWS architecture required merging 51 distinct temporal layers.

Several hardware and toolchain limitations qualify the findings of this study:

- **Tooling and Compiler Immaturity:** Mainstream edge inference engines remain deeply optimized for standard feed-forward CNN layers. They lack native support for the recurrence mechanisms and dynamic matrix transformations unique to modern State-Space Models.
- **Dataset and Domain Boundaries:** This evaluation was limited to two representative temporal TinyML benchmarks (HAR and KWS). Consequently, these workloads do not capture performance trends in multidimensional domains, such as computer vision or high-frequency telemetry, where Mamba's linear scaling properties could yield different results.
- **Operator Bit-Width Constraints:** Our analysis was restricted to standard `FLOAT32` and uniform `INT8` quantization. Exploring sub-byte or mixed-precision bit-widths (e.g., `INT4`) represents a critical path to memory reduction, but such implementations are currently blocked by missing operator definitions in the core TFLM distribution.
- **Hardware and Ecosystem Vendor Lock:** Benchmarking was conducted exclusively on the Espressif ESP32 hardware family. Because microcontrollers vary dramatically in their custom vector extensions and non-volatile

memory controller designs, evaluating these identical graphs on competing vendor architectures (e.g., ARM Cortex-M or STM32) would yield different latency and execution efficiency profiles.

Ultimately, manual sub-graph partitioning introduces significant system complexity, requiring host application code to be highly model-aware and to manually orchestrate internal state pointers. While this design layout successfully expands the upper bound of deployable model sizes, it introduces a severe latency penalty that complicates real-time applications. Resolving this trade-off requires the development of a unified, custom-quantized TFLM interpreter kernel that encapsulates the entire recurrent SSM loop natively. Such a specialized kernel would combine the memory efficiency of loop partitioning with the low overhead of an unrolled graph, and we leave its development to future work.

# 7 Responsible Research

This section outlines the ethical considerations, reproducibility frameworks, and the structured use of generative AI tools throughout this study. Specifically, we document dataset and code compliance, experimental replication protocols, and potential dual-use risks alongside corresponding mitigation strategies.

## 7.1 Ethical Considerations

While this study does not involve human subjects or direct third-party participants, the broader implications of deploying machine learning on edge devices warrant careful ethical consideration. By optimizing machine learning frameworks for resource-constrained microcontrollers, this work shifts computational workloads from centralized datacenters to low-power edge devices, thereby contributing to environmental sustainability via reduced operational carbon footprints.

Furthermore, on-device processing inherently enhances user privacy by keeping raw data local and minimizing cloud-based data exposure. However, decentralized processing also introduces potential risks, such as facilitating covert local surveillance. Consequently, downstream deployments must integrate robust hardware safeguards, operational transparency, and appropriate governance frameworks. Because TinyML applications frequently process highly sensitive telemetry (e.g., biomedical or acoustic data), developers utilizing this framework must implement stringent security measures and maintain transparency regarding data lifecycle management.

Additionally, the proliferation of specialized edge devices introduces e-waste risks due to rapid hardware obsolescence. To mitigate this impact, practitioners are encouraged to prioritize hardware longevity, provide long-term software maintenance, and adhere to responsible electronic waste disposal practices.

Finally, we acknowledge the systematic risk of algorithmic and dataset bias. The datasets benchmarked in this work (HAR and Google Speech Commands) are established standards within the TinyML community; however, they may exhibit representation gaps across diverse populations or operational environments. These limitations must be factored into the generalization and interpretation of the empirical results.

## 7.2 Reproducibility

All datasets utilized in this research are publicly available, and all associated software dependencies are governed by open-source licenses permitting research usage. To ensure reproducibility, we have comprehensively documented the experimental pipeline, including model training, optimization, quantization, and microcontroller benchmarking. While full replication requires specific hardware targets, the ESP32 family utilized in this study is widely accessible within the research community.

To ensure empirical fidelity, performance profiling was conducted using a unified, consistent profiler across all experimental iterations, with results systematically annotated by the specific microcontroller variant used. The complete codebase, including replication scripts, library dependencies, and exact version constraints, is hosted in a public repository to facilitate community verification.

## 7.3 Use of AI Tools

Generative AI tools were utilized strictly as supportive instruments during the development and drafting phases of this research. AI assistance was restricted to writing utility scripts, debugging software implementations, and refining the grammatical clarity of the manuscript.

Specifically, AI tools were leveraged for the following tasks:

- Automating boilerplate scripts for the experimental pipeline execution. Debugging syntax and integration issues within the experimental codebase.
- Enhancing the syntactic clarity, flow, and structure of the written narrative.
- Assisting with automated LaTeX formatting and structural consistency.

All core scientific decisions, experimental designs, and data interpretations were executed independently by the authors. Generative AI tools were not employed to synthesize, alter, or influence experimental outcomes, empirical data, or scientific conclusions.

# 8 Conclusions and Future Work

This paper examined how Mamba can be adapted and optimized for inference on microcontrollers. Our results support three concise findings:

- Deploying via TFLM (with model-graph stripping) substantially reduces peak RAM compared to the C-based MambaLite-Micro workflow (roughly a 75% reduction in our runs).
- Memory vs. latency trade-off: INT8 quantization compared to FLOAT32 cuts peak RAM by about $\approx 33\%$ and model file size by $\approx 60\%$, but on the tested ESP32-class hardware it increased latency because efficient INT8 vector instructions are lacking.

- Split vs. full deployment: splitting into pre-SSM / SSM-step / post-SSM lowers peak RAM and permits longer sequences, but increases sensitivity to static quantization and complicates the runtime pipeline.

Given our findings, the TFLM-based deployment pipeline seems to be a better path for Mamba on microcontrollers than the C-based approach of MambaLite-Micro, as it allows for more flexible experimentation and even better memory efficiency. While the C-based approach could allow for even more aggressive optimizations, they would be hard to deploy for general use models. To make the deployment more practical, the model split that we implemented would need to be integrated into the TFLM as a custom operator.

Taken together, Mamba is a strong candidate for TinyML on MCUs when memory optimizations are prioritized, but real-world latency depends heavily on hardware and kernel support. A next step that could be pursued is implementing a custom, quantized Selective State-Space Model kernel for TFLM and running targeted microbenchmarks across MCU families to guide low-level optimizations. The model split that we applied was applied in order to mitigate the lack of a looping mechanism in TFLM, but it had a side effect of reducing the peak RAM usage, the reason for this could be further investigated. We also recommend systematically evaluating pruning methods (structured and unstructured) to reduce parameter count and memory footprint, and combining those experiments with Quantization-Aware Training and lower-bit formats.

## References


[1] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. "O'Reilly Media, Inc.".

[2] M. Tri Lê, P. Wolinski, and J. Arbel, "Efficient Neural Networks for Tiny Machine Learning: A Comprehensive Review," vol. 17, no. 4, pp. 1–41.

[3] A. Gu and T. Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces."

[4] Z. Xu, Y. Yue, X. Hu, Z. Yuan, Z. Jiang, Z. Chen, J. Yu, C. Xu, S. Zhou, and D. Yang, "MambaQuant: Quantizing the Mamba Family with Variance Aligned Rotation Methods."

[5] A. Tegon, N. Lehmann, Y. Li, A. Cossettini, L. Benini, and T. M. Ingolfsson, "FEMBA on the Edge: Physiologically-Aware Pre-Training, Quantization, and Deployment of a Bidirectional Mamba EEG Foundation Model on an Ultra-low Power Microcontroller."

[6] I. F. Shihab, S. Akter, and A. Sharma, "Efficient Unstructured Pruning of Mamba State-Space Models for Resource-Constrained Environments," *arXiv preprint*, 2025.

[7] H. Xu, J. Xia, W. Yang, Y. Sui, and S. Xia, "MambaLite-Micro: Memory-Optimized Mamba Inference on MCUs." arXiv preprint https://arxiv.org/abs/2509.05488.

[8] T. Cassimon, S. Vanneste, S. Bosmans, S. Mercelis, and P. Hellinckx, "Designing resource-constrained neural networks using neural architecture search targeting embedded devices," vol. 12, p. 100234.

[9] K. Dokic, M. Martinovic, and D. Mandusic, "Inference speed and quantisation of neural networks with TensorFlow Lite for Microcontrollers framework," in *2020 5th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, pp. 1–6.

[10] T. Suwannaphong, F. Jovan, I. Craddock, and R. McConville, "Optimising TinyML with quantization and distillation of transformer and mamba models for indoor localisation on edge devices," vol. 15, no. 1, p. 10081.

[11] C. Li, D. Huang, K. Yao, X. Ni, L. Shen, and F. Luo, "Physics-Guided Tiny-Mamba Transformer for Reliability-Aware Early Fault Warning."

[12] X. Ma, Z. Ni, and X. Chen, "TinyViM: Frequency Decoupling for Tiny Hybrid Vision Mamba."

[13] T. Wang, Y. Shu, L. Qiao, D. Ding, and G. Li, "LightMamba: A Resource-Efficient Deep-Learning Method for UWB NLOS Identification Based on Selective State-Space Modeling," vol. 13, no. 5, pp. 8735–8748.

[14] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems."

[15] M. Nachin, D. Desai, S. S. Jia, C. Lai, M. Liu, J. Szwejbka, R. Alvarez, R. Ascani, D. Bort, M. Candales, *et al.*, "ExecuTorch - a unified PyTorch solution to run AI models on-device," *arXiv preprint arXiv:2605.08195*, 2026.

[16] G. Franco, A. Pappalardo, and N. J. Fraser, "Xilinx/brevitas," 2025.

[17] A. Or, A. Jain, D. Vega-Myhre, J. Cai, C. D. Hernandez, Z. Zheng, D. Guessous, V. Kuznetsov, C. Puhrsch, M. Saroufim, S. Rao, T. Tran, and A. Samardžić, "TorchAO: PyTorch-Native Training-to-Serving Model Optimization."

[18] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A Public Domain Dataset for Human Activity Recognition Using Smartphones,"

[19] P. Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition," *ArXiv e-prints*, Apr. 2018.

[20] Y. Enokibori, "rTsfNet: A DNN model with Multi-head 3D Rotation and Time Series Feature Extraction for IMU-based Human Activity Recognition."

[21] J. Wang, L. Yu, L. Huang, C. Zhou, H. Zhang, Z. Song, Z. Ma, and Z. Zhang, "Efficient Speech Command Recognition Leveraging Spiking Neural Network and Curriculum Learning-based Knowledge Distillation,"