Mamba-based Network Design for Keyword Spotting

Final Project No. 203

Guided and Advised by: Zuher Jahshan
Academic Supervisor: Dr. Leonid Yavits

Authors:
Alex Makarov • Ran Levi

September 2025

Contents

Acknowledgement

We would like to express our sincere gratitude to our Academic mentor, Zuher Jahshan. His guidance was invaluable in helping us define clear and achievable goals for this project. He equipped us with the necessary foundational knowledge and skillfully directed us to further resources, which proved essential for our research. Throughout the year, his constructive feedback on our deliverables was instrumental in shaping the final outcome of our work.

We are also deeply grateful to Dr. Leonid Yavits for his academic supervision and unwavering support throughout this project. His oversight provided a strong academic foundation for our research.

Chapter 1

Introduction

1.1 Motivation & Problem Statement

Modern AI leans heavily on sequence modelling - from language to genomics, where the core challenge is handling long-range dependencies without blowing up compute or memory. Transformers set the standard with self-attention, but their quadratic time and memory costs limit scalability, especially in always-on edge settings like keyword spotting (KWS). Our project, "Mamba-based network design for keyword spotting," tackles this head-on by evaluating Structured State-Space Models (SSMs), specifically Mamba - as a post-Transformer alternative that aims to pair the linear scalability of recurrent models with the contextual strength we expect from attention.

We focus on two goals:

- **Model quality:** build a Mamba-based KWS system that matches or surpasses SOTA accuracy.
- **Efficiency & deployability:** analyze compute/memory needs and validate feasibility on low-power, low-latency edge devices.

Why KWS on the edge matters: it powers wake-word detection for assistants, low-power edge devices like cellphones and smart-home IOT, and privacy-preserving triggers on device. The constraints are very important:

- Near real-time latency
- Very low energy consumption
- Small memory budgets (MCUs, mobile SoCs)

Classic CNN/RNN pipelines can work but struggle as windows grow - either due to attention's memory footprint or recurrent optimization frictions. Mamba, a selective SSM, offers linear-time operation and total linear inference, making it a strong fit for edge KWS.

1.2 Project Goals and Methodology

The primary goals of this project were to establish a comprehensive understanding of the foundational technologies, develop a high-performing model, and provide a clear pathway for its practical deployment. These goals were articulated through a series of sequential objectives that form the backbone of our methodological approach:

1. **Foundational Research:** Deepen our understanding of core concepts including neural networks, keyword spotting, and the underlying principles of Structured State-Space Models (SSMs) and the Mamba architecture. This involved a thorough review of key literature and architectural papers, beginning with an exhaustive examination of the history of neural networks, from the foundational Multilayer Perceptrons (MLPs) to the dominant architectures of the pre-Transformer era (CNNs, RNNs/LSTMs) and the breakthroughs of the Transformer. This provided the necessary context to appreciate the inefficiencies addressed by post-Transformer models, eventually taking a deep dive into the Mamba architecture and its underlying SSM principles.

2. **Data Analysis and Preparation:** Using the Google Speech Commands dataset, we built a robust data pipeline. This included converting raw audio waveforms to frequency-domain representations, specifically Mel spectrograms and Mel-Frequency Cepstral Coefficients (MFCCs). We also implemented a series of data augmentations, such as time-shifting and noise injection, to create a more generalized and robust model. This phase was critical to ensuring our models had access to high-quality, well-processed training data.

3. **Model Implementation and Neural Network Development:** Implement the Mamba-based network architecture from scratch, ensuring it is tailored for the keyword spotting task. We designed and implemented a Mamba-based KWS model, creating three distinct variants (small, medium, and large) to explore the relationship between model complexity, parameter count, and performance. We also implemented a Ret-Net model as a parallel post-Transformer baseline to provide comprehensive comparisons.

4. **Model Training and Evaluation for SOTA Performance:** Train the implemented models on the prepared dataset, carefully tuning hyperparameters and monitoring the training process to achieve optimal performance. We targeted state-of-the-art validation accuracy in the range of 97.5%. With the models trained, we conducted a multi-faceted evaluation, comparing our models' accuracy and computational efficiency against a MobileNetV2-based CNN baseline, and qualitatively analyzed how Mamba's performance scaled with longer sequence lengths. We evaluated the trained models using standard metrics such as accuracy and latency, and compare their performance against established baseline architectures from the pre and post-Transformer eras.

5. **Deployment Analysis and Hardware Architectural Suggestions:** Analyze the computational capabilities required for edge device integration. In this final phase, we focused on the physical implementation, characterizing the models' hardware and architectural requirements and developing a series of hardware-level suggestions. This involved assessing the memory and compute footprint of the trained model, evaluating its feasibility on resource-constrained devices, and exploring hardware-software co-design techniques to enable efficient deployment.

The success of this project is measured by achieving state-of-the-art performance with our Mamba models, quantitatively demonstrating their efficiency, and, most importantly, providing a set of concrete, actionable hardware architectural suggestions that bridge the gap between the model's algorithmic elegance and the physical constraints of edge computing.

Looking forward, we envision this project as a foundational step toward broader adoption of post-Transformer architectures in edge computing applications. By demonstrating the viability of this architecture on edge devices, we anticipate this work will contribute to a new generation of efficient AI systems that can operate effectively in environments where traditional Transformer models are computationally prohibitive.

Chapter 2

Background & Literature Review

2.1 Neural Networks

At its most fundamental level, an Artificial Neural Network (ANN) is a computational model inspired by the structure and function of the human brain. It consists of an interconnected network of computational units, or neurons, organized into layers. The most basic of these neurons is the Linear Threshold Unit (LTU), also known as a perceptron. An LTU computes a weighted sum of its inputs and then applies a step function to the result to produce an output.

The core operation of a neuron can be expressed mathematically. Given a set of inputs $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and a corresponding set of weights $\mathbf{w} = [w_1, w_2, \ldots, w_n]$, the weighted sum, Z, is calculated as:

$$Z = \sum_{i=1}^{n} w_i \cdot x_i + b = \mathbf{w}^T \cdot \mathbf{x} + b \tag{2.1}$$

where b is the bias term. The bias is a crucial component that allows the network to capture more complex patterns by shifting the decision boundary, effectively enabling the model to make predictions without needing all inputs to be non-zero.

The final output, $h_{\mathbf{w}}(\mathbf{x})$, is then determined by an activation function applied to this weighted sum. Early perceptrons used a simple step function, but more modern networks employ non-linear functions like the Rectified Linear Unit (ReLU) or Softmax for classification tasks, which provide a more nuanced and differentiable output.

A Multi-Layer Perceptron (MLP) extends this concept by introducing one or more "hidden layers" of neurons between the input and output layers. Networks with multiple hidden layers are often referred to as Deep Neural Networks (DNNs).

The process of training an ANN is a sophisticated form of optimization, driven by two key steps:

- **Forward Pass:** An input is fed through the network, and the output of each neuron in each layer is computed sequentially until a final output is produced.
- **Backpropagation:** The network's output is compared to the true label using a loss function to calculate the error. This error is then propagated backward through the network, and the "gradient of the error with respect to each weight" is computed using a technique called reverse-mode automatic differentiation. This allows the network to determine precisely how each weight contributed to the overall error.

Finally, the weights are updated using an optimization algorithm (like Gradient Descent), with a learning rate (η), to incrementally reduce the error:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \times \frac{\partial \text{Error}}{\partial \mathbf{w}_{\text{old}}} \tag{2.2}$$

2.2 Pre-Transformer Era - CNNs, RNNs/LSTMs

The "pre-Transformer" era of deep learning was defined by specialized network architectures designed to tackle specific data types, with Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) being the most prominent.

2.2.1 Convolutional Neural Networks (CNNs)

CNNs are a class of neural networks specifically designed to process data with a known spatial or temporal structure, such as images, video, or in our case, audio spectrograms. They are distinguished by their use of learnable filters, or kernels, which slide over the input data in a process called convolution. This operation extracts local features, such as edges or patterns, which are then passed to the next layer.

Key components of a CNN include:

- **Kernels (Filters):** Small, learnable matrices (e.g., a 3×3 matrix) that convolve with the input to detect specific patterns.
- **Feature Maps:** The output of a convolution layer, representing where and how strongly a detected pattern appears in the input.
- **Pooling Layers:** Layers that downsample the feature maps, reducing their spatial dimensions and making the model more robust to minor shifts in the input. This also reduces the overall computational complexity.

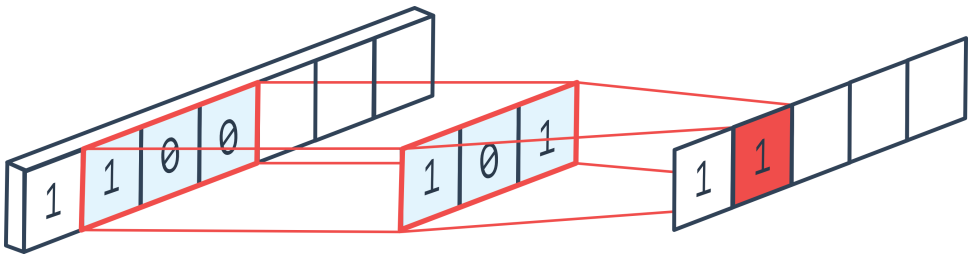

Figure 2.1: Illustration of 1D convolution operation showing how a learnable filter (kernel) slides across the input sequence to produce feature maps. Each position of the kernel computes a weighted sum of local elements, extracting local patterns and features that are useful for sequence modeling tasks like audio processing.

Despite their effectiveness for spatial data, CNNs have significant limitations for tasks requiring sequential understanding. They typically require fixed-size inputs and lack a built-in mechanism to track sequential dependencies, making them unsuitable for tasks like natural language processing or time-series prediction.

2.2.2 Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs)

RNNs were specifically developed to address the limitations of CNNs for sequential data. Their defining characteristic is a "memory" in the form of a hidden state that is updated at each time step based on the current input and the hidden state from the previous time step. This feedback loop allows them to capture temporal dependencies and patterns, making them ideal for tasks like speech recognition.

The state update of a basic RNN can be expressed as:

$$h_t = f(W_h h_{t-1} + W_x x_t + b_h) \tag{2.3}$$
$$y_t = g(W_y h_t + b_y) \tag{2.4}$$

where h_t is the hidden state at time t, x_t is the input, y_t is the output, and f and g are activation functions. The matrices W_h, W_x, and W_y are the learnable weight parameters.

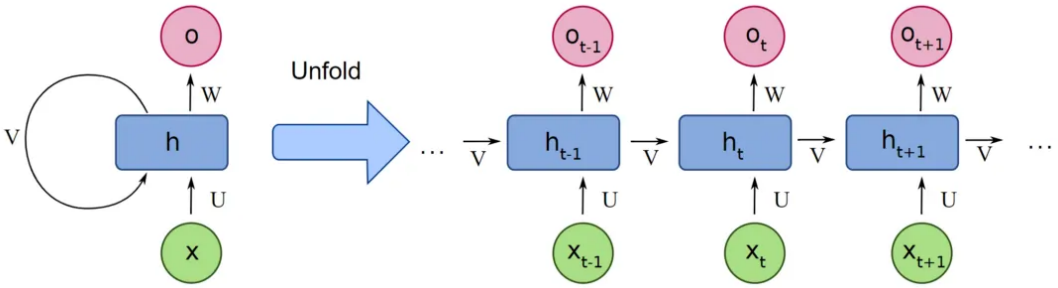

Figure 2.2: RNN architecture and its unfolded representation through time. The left side shows the compact RNN representation with recurrent connections, while the right side illustrates how the network unfolds across multiple time steps, showing how the hidden state h_t depends on both the current input x_t and the previous hidden state h_{t-1}.

However, standard RNNs famously suffer from two major problems: the vanishing gradient problem and the exploding gradient problem, which make it difficult for them to learn and retain long-term dependencies in a sequence.

To solve this, specialized memory cells were developed, most notably the Long Short-Term Memory (LSTM) cell.

2.3 Transformers

The Transformer architecture, introduced in the seminal 2017 paper "Attention Is All You Need," was designed to overcome the sequential bottlenecks of RNNs by completely abandoning recurrence in favor of a mechanism called self-attention. This allowed the model to process all tokens in a sequence simultaneously, achieving unprecedented parallelizability and scalability.

The core intuition behind self-attention is that it allows the model to "weigh the importance of different parts of the input when processing each element." It does this by using three learned vectors for each token in the sequence:

- **Query (Q):** Represents the current token's request for information from other tokens.

- **Key (K):** Represents the information each token offers to others.
- **Value (V):** Contains the actual content that is passed from the token.

The attention mechanism calculates a score by comparing the Query of the current token against the Key of every other token in the sequence. These scores are then used to create a weighted sum of the Value vectors, effectively producing a new, context-aware representation for each token.

The key challenge for Transformers is that by processing tokens in parallel, they lose the inherent sequential order that RNNs rely on. This is solved by adding positional embeddings-vectors that encode the position of each token in the sequence - to the initial word embeddings. The positional embeddings are generated using sine and cosine functions of different frequencies, allowing the model to learn and distinguish between token positions without relying on an iterative process.

The overall architecture of a Transformer consists of an encoder and a decoder, each made up of multiple layers containing self-attention mechanisms and feed-forward networks.

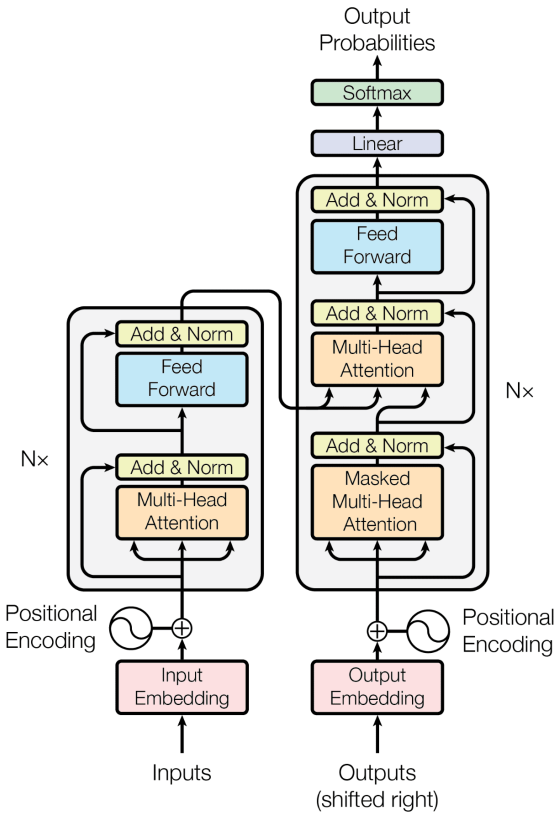

Figure 2.3: The Transformer architecture showing the encoder-decoder structure with multi-head self-attention mechanisms and feed-forward networks. The encoder processes the input sequence in parallel, while the decoder generates the output sequence. Both components utilize positional embeddings to maintain sequence order information.

Despite their success, Transformers have a critical drawback: the computational cost and memory usage of the self-attention mechanism scale quadratically with the sequence length, i.e., $O(L^2)$ where L is the sequence length. This makes them extremely inefficient for very long sequences, a problem that became known as the "memory wall" of Transformers.

2.4 Post-Transformer Era

The "Post-Transformer Era" is a term that encapsulates the ongoing effort to develop new architectures that retain the strengths of Transformers - their parallelizability and long-range contextual understanding, while circumventing their quadratic complexity and high memory footprint. The goal is to design models that scale gracefully with sequence length, making them more suitable for long-context tasks and, crucially, for resource-constrained environments.

This movement has given rise to a "zoo" of innovative models, including Retentive Networks (Ret-Net) and Structured State-Space Models (SSMs). Our project focuses on Mamba, a particularly promising development within the SSM family.

2.5 SSM Intuition

Structured State-Space Models (SSMs) provide an elegant framework for modeling sequential data by treating it as a continuous-time dynamical system that is then discretized for computation. At its core, an SSM is a linear system that maps an input sequence $x(t)$ to an output sequence $y(t)$ via a latent hidden state $h(t)$.

In a continuous setting, this system is defined by a pair of linear differential equations:

$$h'(t) = Ah(t) + Bx(t) \quad \text{(state equation)} \tag{2.5}$$
$$y(t) = Ch(t) + Dx(t) \quad \text{(output equation)} \tag{2.6}$$

where A, B, C and D are matrices that define the dynamics of the system.

For practical computation, these continuous equations must be converted into a discrete-time representation. This discretization process, using a fixed time step (Δ), yields a set of recurrent equations that are fundamentally similar to those of a simple RNN:

$$h_k = \bar{A}(\Delta)h_{k-1} + \bar{B}(\Delta)x_k \tag{2.7}$$
$$y_k = Ch_k \tag{2.8}$$

where $\bar{A}$ and $\bar{B}$ are the discretized parameters (D discarded for simplicity).

A powerful feature of SSMs is their recurrent-convolutional duality. The recurrent formula can be "unrolled" into a convolutional operation, allowing the model to process the entire input sequence in parallel during training, a property similar to Transformers. This duality allows for efficient training with convolutions and fast, linear-time inference with recurrence. This is a critical advantage over RNNs, which are inherently sequential and slow to train.

To demonstrate this duality mathematically, consider unrolling the recurrent equations. Starting from the discrete recurrent form:

$$h_0 = \bar{B}x_0 \tag{2.9}$$
$$h_1 = \bar{A}h_0 + \bar{B}x_1 = \bar{A}\bar{B}x_0 + \bar{B}x_1 \tag{2.10}$$
$$h_2 = \bar{A}h_1 + \bar{B}x_2 = \bar{A}^2\bar{B}x_0 + \bar{A}\bar{B}x_1 + \bar{B}x_2 \tag{2.11}$$
$$\vdots \tag{2.12}$$

In general, we can express the hidden state at time k as:

$$h_k = \sum_{j=0}^{k} \bar{A}^{k-j} \bar{B} x_j \tag{2.13}$$

Substituting this into the output equation $y_k = Ch_k$, we get:

$$y_k = C \sum_{j=0}^{k} \bar{A}^{k-j} \bar{B} x_j = \sum_{j=0}^{k} C \bar{A}^{k-j} \bar{B} x_j \tag{2.14}$$

This can be rewritten as a convolution with a structured kernel. Define the convolution kernel:

$$\mathcal{K} = (C\bar{B}, C\bar{A}\bar{B}, C\bar{A}^2\bar{B}, \ldots, C\bar{A}^L\bar{B}) \tag{2.15}$$

Then the output sequence can be computed as:

$$\mathbf{y} = \mathcal{K} * \mathbf{x} \tag{2.16}$$

where $*$ denotes the convolution operation and $\mathbf{x} = (x_0, x_1, \ldots, x_L)$ is the input sequence.

This formulation reveals the key insight: during training, we can compute the entire output sequence in parallel using Fast Fourier Transform (FFT)-based convolution algorithms with $O(L \log L)$ complexity, while during inference, we can use the recurrent form for $O(1)$ per-step computation.

An improvement to this base model involves using a specially structured A matrix, often parameterized as diagonal, which makes it computationally efficient to calculate powers of A and effectively capture long-term memory dependencies.

2.6 Deep Dive Into Mamba

Mamba, introduced in the 2023 paper "Mamba: Linear-Time Sequence Modeling with Selective State Spaces," is a significant evolution of the Structured State-Space Model (SSM) paradigm, designed to address the weaknesses of earlier models while retaining their efficiency. While prior SSMs were highly effective for continuous data like audio, they struggled with "information-dense" and discrete data (e.g., text and genomics) because they lacked a mechanism for content-based reasoning.

2.6.1 The Limitation of Linear Time-Invariance (LTI)

The core limitation of previous SSMs stemmed from a property known as Linear Time-Invariance (LTI). This meant that their state-space parameters - the matrices A, B, C, and the discretization time step Δ-remained fixed and unchanging for all time steps and inputs. While this property enabled the efficient "recurrent-convolutional duality" that was key to their fast training, it also rendered them inflexible. They could not dynamically adapt their behavior based on the content of the input, making them unable to perform crucial tasks that require content-aware reasoning, such as:

- **Selective Copying:** A synthetic task that requires a model to filter out irrelevant "noise" tokens and only remember specific, relevant ones, even when their spacing varies. LTI models fail here because their static convolutional kernels cannot account for variable spacing between tokens.

Figure 2.4: Selective copying task demonstration. The model must selectively copy and remember only the relevant tokens while ignoring noise tokens, even when their spacing varies throughout the sequence.

- **Induction Heads:** A task that measures a model's ability to perform in-context learning by recalling patterns from earlier in a sequence to make predictions later on. This is a critical ability for modern Large Language Models (LLMs). LTI models, with their fixed dynamics, cannot "select" which previous token to recall from their history.

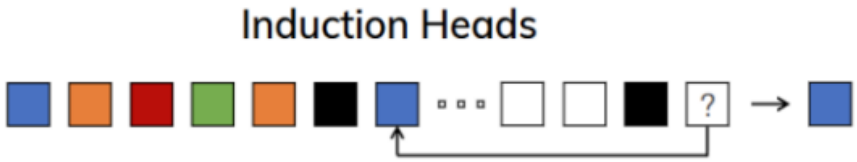

Figure 2.5: Induction heads task illustration. The model must identify patterns from earlier in the sequence and use them to make predictions later on, demonstrating in-context learning capabilities essential for modern language models.

In essence, the efficiency of SSM models was tied to a static, one-size-fits-all input approach to information processing, limiting their effectiveness on complex, discrete data modalities and being unable to parallelize using a fixed convolution kernel.

2.6.2 The Selective SSM Mechanism: Breaking LTI

Mamba's central innovation is its selection mechanism, which breaks the LTI constraint by making the model's key SSM parameters a function of the input data itself. Specifically, the B, C, and Δ parameters are no longer static but are dynamically computed for each token. This transforms the system from being time-invariant to time-varying.

The standard, time-invariant SSM recurrence is:

$$h_k = \bar{A}h_{k-1} + \bar{B}x_k \tag{2.17}$$
$$y_k = Ch_k \tag{2.18}$$

where $\bar{A}$ and $\bar{B}$ are the discretized parameters derived from the continuous-time parameters A, B, and Δ.

In Mamba, the selection mechanism takes the input sequence $\mathbf{x} \in \mathbb{R}^{B\times L\times D}$ and dynamically generates the SSM parameters for each time step. The continuous-time parameters B, C, and Δ are functions of the input x_k at time step k:

$$\Delta_k = \text{softplus}(\text{Linear}_1(x_k) + \text{Parameter}) \tag{2.19}$$
$$B_k = \text{Linear}_N(x_k) \tag{2.20}$$
$$C_k = \text{Linear}_N(x_k) \tag{2.21}$$

where Linear_N and Linear_1 are linear projections. The softplus activation function ensures that Δ_k is always positive, similar to gating mechanisms in recurrent networks.

The discretization step then uses these dynamic parameters to create a new, input-dependent recurrence at each time step:

$$\bar{A}_k = \exp(\Delta_k A) \tag{2.22}$$
$$\bar{B}_k = (\Delta_k A)^{-1}(\exp(\Delta_k A) - I) \cdot \Delta_k B_k \tag{2.23}$$
$$h_k = \bar{A}_k h_{k-1} + \bar{B}_k x_k \tag{2.24}$$

This is the selective scan operation, which allows Mamba to perform content-based reasoning. The magnitude of the time step parameter Δ_k is particularly important, as it determines how much the current input affects the state: a large Δ_k makes the model focus on the current input and "forget" previous history, while a small Δ_k allows the model to "persist" its state and ignore the current input. This dynamic behavior allows the model to filter out irrelevant information and remember relevant context indefinitely, much like a Transformer's attention mechanism but with a linear-scaling recurrent state rather than a quadratically-scaling key-value cache.

The A matrix, which handles the "forgetting" or decay dynamics, remains fixed (typically diagonal for computational efficiency). The B_k parameter controls what information from the current input x_k is stored in the state, while C_k controls which parts of the state are used to produce the output. This input-dependent dynamic gives the model the power of selective information processing.

This input-dependency, however, has a critical consequence: it invalidates the recurrent-convolutional duality. Since the parameters are no longer static, the efficient convolutional representation that made prior SSMs so fast to train is no longer applicable. A convolution would require a different kernel for each position in the sequence, which is computationally and memory-wise infeasible. To address this, Mamba introduces a novel hardware-aware parallel algorithm designed to run efficiently in recurrent mode on modern GPUs.

Algorithmic Comparison: Vanilla SSM vs. Selective SSM

To clearly illustrate the fundamental difference between traditional SSMs and Mamba's selective approach, we present a side-by-side algorithmic comparison. The key dimensions are:

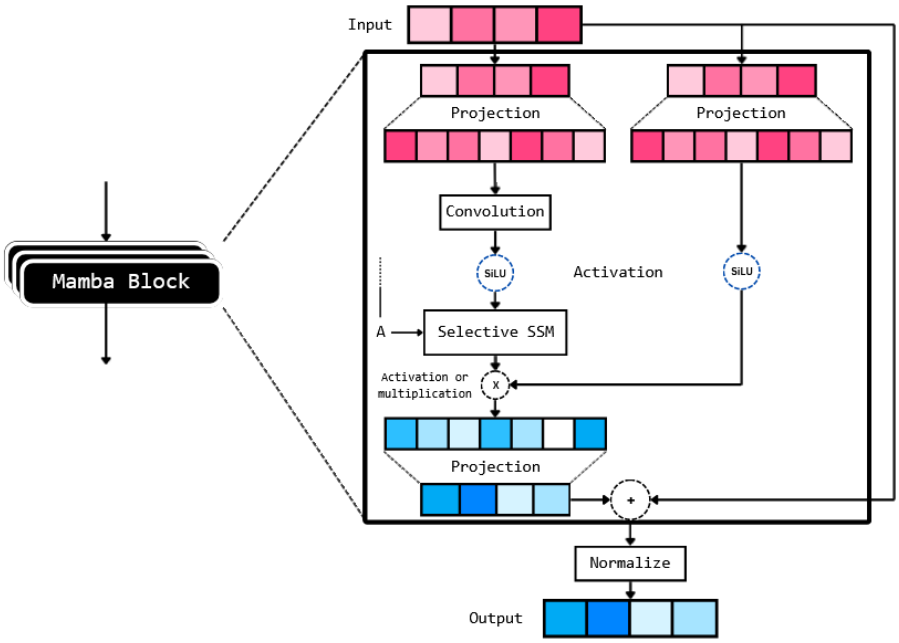

Figure 2.6: The Mamba block architecture showing the selective SSM mechanism. The input x is processed through multiple linear projections to generate dynamic parameters Δ, B, and C. The selective scan operation uses these input-dependent parameters to update the hidden state, enabling content-based reasoning while maintaining linear complexity.

- B: Batch size
- L: Sequence length
- D: Model dimension (hidden size)
- N: State dimension (SSM state size)

The key differences are immediately apparent:

1. **Parameter Dependencies:** In Algorithm 1 (Vanilla SSM), the parameters B, C, and Δ are static across all time steps and batch elements. In Algorithm 2 (Selective SSM), these parameters are functions of the input $\mathbf{x}$, making them dynamic and input-dependent.

2. **Memory Complexity:** The vanilla SSM has constant-sized parameters regardless of sequence length, while the selective SSM's parameters scale with both batch size B and sequence length L. This represents a trade-off between expressiveness and memory efficiency.

3. **Computational Mode:** The vanilla SSM can leverage both recurrent and convolutional computation modes due to its time-invariance, while the selective SSM is restricted to recurrent computation only, necessitating the hardware-aware parallel algorithm.

4. **Selective Capacity:** Only the selective SSM can perform content-based reasoning, filtering relevant information and forgetting irrelevant details based on the input content itself.

Algorithm 1: Vanilla SSM (S4)
Input: $\mathbf{x} \in \mathbb{R}^{B \times L \times D}$
Output: $\mathbf{y} \in \mathbb{R}^{B \times L \times D}$

1. $A \in \mathbb{R}^{D \times N} \leftarrow$ Parameter
2. $B \in \mathbb{R}^{D \times N} \leftarrow$ Parameter
3. $C \in \mathbb{R}^{D \times N} \leftarrow$ Parameter
4. $\Delta \in \mathbb{R}^{D} \leftarrow \text{softplus}(\text{Parameter})$
5. $\bar{A}, \bar{B} \in \mathbb{R}^{D \times N} \leftarrow \text{discretize}(\Delta, A, B)$
6. $\mathbf{y} \leftarrow \text{SSM}(\bar{A}, \bar{B}, C)(\mathbf{x})$
   // Time-invariant: recurrence or convolution
7. **return y**

Algorithm 2: Selective SSM (S6, Mamba)
Input: $\mathbf{x} \in \mathbb{R}^{B \times L \times D}$
Output: $\mathbf{y} \in \mathbb{R}^{B \times L \times D}$

1. $A \in \mathbb{R}^{D \times N} \leftarrow$ Parameter
2. $B \in \mathbb{R}^{B \times L \times N} \leftarrow \text{Linear}_N(\mathbf{x})$
3. $C \in \mathbb{R}^{B \times L \times N} \leftarrow \text{Linear}_N(\mathbf{x})$
4. $\Delta \in \mathbb{R}^{B \times L \times D} \leftarrow$ $\text{softplus}(\text{Linear}_1(\mathbf{x}) + \text{Parameter})$
5. $\bar{A}, \bar{B} \in \mathbb{R}^{B \times L \times D \times N} \leftarrow$ $\text{discretize}(\Delta, A, B)$
6. $\mathbf{y} \leftarrow \text{SSM}(\bar{A}, \bar{B}, C)(\mathbf{x})$
   // Time-varying: recurrence (scan) only
7. **return y**

2.6.3 Mamba's Hardware-Aware Design

The move to a time-varying system creates a significant computational problem. To overcome this, Mamba introduces a novel hardware-aware parallel algorithm designed to run efficiently in recurrent mode on modern GPUs, directly leveraging their specific memory hierarchy. This algorithm's goal is to circumvent the performance and memory bottlenecks associated with traditional hardware by intelligently managing data movement and computation.

The authors make two key observations that justify this approach:

1. The naive recurrent computation uses $O(BLDN)$ floating-point operations (FLOPs), while the convolutional computation uses $O(BLD\log(L))$ FLOPs. For long sequences and a medium size state dimension N, the recurrent mode can actually use fewer FLOPs with a lower constant factor, despite its sequential nature.

2. The main challenges of the recurrent mode are its sequential nature and large memory usage for the expanded state (h). The goal is to avoid materializing the full state h by keeping it only in the faster levels of the memory hierarchy.

This algorithm is built on three key techniques that are integral to the Mamba implementation:

Kernel Fusion

On modern GPUs, most operations (except matrix multiplication) are limited by memory bandwidth, with frequent data transfers between the slow HBM (High-Bandwidth Memory) and the fast on-chip SRAM being a major bottleneck. Kernel fusion is a technique that combines a sequence of operations into a single, unified computation unit, or "kernel."

In the Mamba block, the discretization step, the recurrent state update, and the final output multiplication are fused into one custom CUDA kernel. A standard, unfused implementation would involve multiple memory transfers: load parameters, compute a partial result, write that result to HBM, read it back, and then perform the next step. Mamba's fused kernel avoids these costly I/O operations by loading all necessary SSM parameters (A, B, C, Δ) from the slow HBM into the fast on-chip SRAM just once, performing the entire computation there, and only writing the final output back to HBM. This dramatically reduces memory bandwidth usage by a factor of the state dimension (N), and can lead to a 20–40× speedup compared to a standard, unfused implementation.

Parallel Scan

Traditional recurrent computations are inherently sequential, making them difficult to parallelize. Mamba's algorithm overcomes this with a work-efficient parallel associative scan algorithm, which is based on the prefix-sum operation. This algorithm allows the input sequence to be broken into chunks, which are then processed in parallel, with intermediate states from these chunks efficiently combined in a separate step.

The associative property of the recurrent equation is what makes this parallelization possible. The hidden state unfolds as a series of multiplications and additions:

$$h_t = \sum_{j=1}^{t} \left(\prod_{k=j+1}^{t} \bar{A}_k \right) \bar{B}_j x_j \tag{2.25}$$

This structure can be rephrased as a parallel scan, which is typically implemented with a two-phase approach: an up-sweep (reduce) and a down-sweep (scan). This approach leverages a binary tree structure to efficiently combine values, reducing the time complexity from $O(L)$ for a sequential loop to $O(L/T)$ with T threads and a depth of $O(\log L)$ for the parallel operations. This allows the model to scale linearly with sequence length ($O(L)$), a critical advantage over the quadratic complexity of Transformers, which is limited by the attention mechanism's need to store a large key-value (KV) cache.

Recomputation

To further reduce the massive memory footprint, especially during training, the algorithm uses recomputation. Intermediate states of the selective scan, which are necessary for the backward pass (backpropagation), are not stored. Instead, they are recomputed on-the-fly when gradients are needed. The logic is that on modern GPUs, the cost of recomputing these states is less than the cost of storing and retrieving them from slow memory (HBM). This memory-saving technique is essential for training Mamba models on extremely long sequences without running out of memory, a problem that plagues Transformer models with their large key-value (KV) caches.

2.6.4 Mamba's Performance and Efficiency

The combination of the selective SSM and the hardware-aware algorithm has resulted in a powerful and highly efficient architecture that achieves state-of-the-art performance across several modalities, including language, audio, and genomics.

- **Language Modeling:** A Mamba-3B model was shown to outperform Transformers of the same size and match the quality of Transformers twice its size. The model

achieves up to 5× higher generation throughput than Transformers of similar size due to its linear scaling and lack of a large KV cache.

- **Long-Context Tasks:** Mamba is capable of extrapolating solutions to synthetic tasks like "Induction Heads" for sequences up to a million tokens long, while other models fail beyond a few thousand tokens. On real-world data like DNA sequences, Mamba's performance continues to improve with context length up to 1M tokens, while baselines degrade.
- **Memory Efficiency:** The selective scan layer has a memory footprint comparable to a highly optimized Transformer implementation with FlashAttention. This is critical for both training and inference, especially for the memory-constrained nature of our KWS task.

Chapter 3

Data & Audio Pre-Processing

3.1 Keyword Spotting

Keyword Spotting (KWS) is a specialized application of speech recognition. Unlike general-purpose speech recognition systems, which transcribe entire sentences, KWS focuses on identifying a small, predefined set of keywords (or "wake words") from a continuous stream of audio. Think of it as an efficient gatekeeper. A KWS model is designed to be "always-on," continuously listening in a low-power state. When it detects a keyword, it triggers a larger, more complex, and more power-intensive system, like a full-scale natural language processing (NLP) model on a server.

This dual-system approach is a core principle for designing real-world, on-device AI applications. For a device like a smart speaker or a mobile phone, running a large NLP model constantly would drain the battery, incur significant data transfer costs, and raise serious privacy concerns by continuously sending audio data to the cloud. KWS elegantly solves these problems by providing a low-latency, low-power solution that can run entirely on the device itself.

For our project, we chose the Google Speech Commands dataset as our primary training and evaluation source. This dataset is a well-established benchmark for limited-vocabulary speech recognition, consisting of over 100,000 one-second audio recordings of 35 different keywords. In our model's main script, we load this dataset using the Hugging Face `datasets` library:

```
# ---- dataset
ds = load_dataset("google/speech_commands", "v0.02")
n_classes = len(ds["train"].features["label"].names)
```

This dataset's structure, with its short, one-second clips, is perfectly suited for our KWS task, allowing us to train and test our model on a controlled and standardized set of audio examples.

3.2 Audio Foundations

Before a neural network can understand an audio signal, that signal must be transformed from its raw, physical form into a meaningful representation that the network can process. At its most basic, sound is a mechanical wave that travels through a medium like air. A microphone captures these waves, converting the fluctuating air pressure into a one-dimensional, time-domain electrical signal, or waveform. This waveform is a sequence of

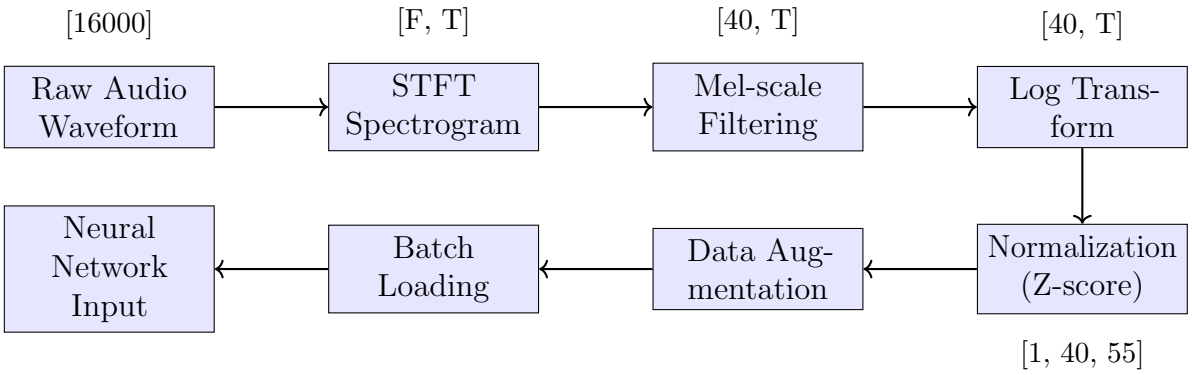

Figure 3.1: Complete data processing pipeline from raw audio input to model-ready features. The pipeline shows the transformation from waveform to spectrograms, followed by feature extraction and normalization steps that prepare the data for neural network processing.

discrete amplitude values, sampled at a specific rate (e.g., 16,000 times per second, or 16 kHz).

The human ear, however, does not perceive this waveform linearly. Our inner ear acts as a natural frequency analyzer. Tiny hair cells, organized along a structure called the basilar membrane, resonate at different frequencies. Longer hairs resonate with lower frequencies, while shorter hairs resonate with higher frequencies, effectively transforming the time-domain pressure wave into a frequency spectrum that is then interpreted by the brain. This biological process is analogous to a Fourier transform, which converts a signal from the time domain to the frequency domain. This key insight from human biology informs how we preprocess audio data for neural networks.

3.3 Deep Dive: Sound Features (STFT → Mel → Log, MFCC)

Raw audio waveforms are often too noisy and computationally expensive for a neural network to process directly. Instead, we extract "features" that represent the signal in a more abstract, human-like way. In our project, we focused on two such features: Mel spectrograms and Mel-Frequency Cepstral Coefficients (MFCCs).

3.3.1 The Mel Spectrogram Pipeline

The process of generating a Mel spectrogram is a multi-step pipeline that mimics key aspects of human hearing:

Short-Time Fourier Transform (STFT)

A standard Fourier transform assumes a signal is stationary over a long period, which is not true for a dynamic speech signal. To address this, we use the STFT, which breaks the raw audio waveform into short, overlapping windows (frames). The Fourier transform is applied to each of these frames, which are small enough to be considered approximately

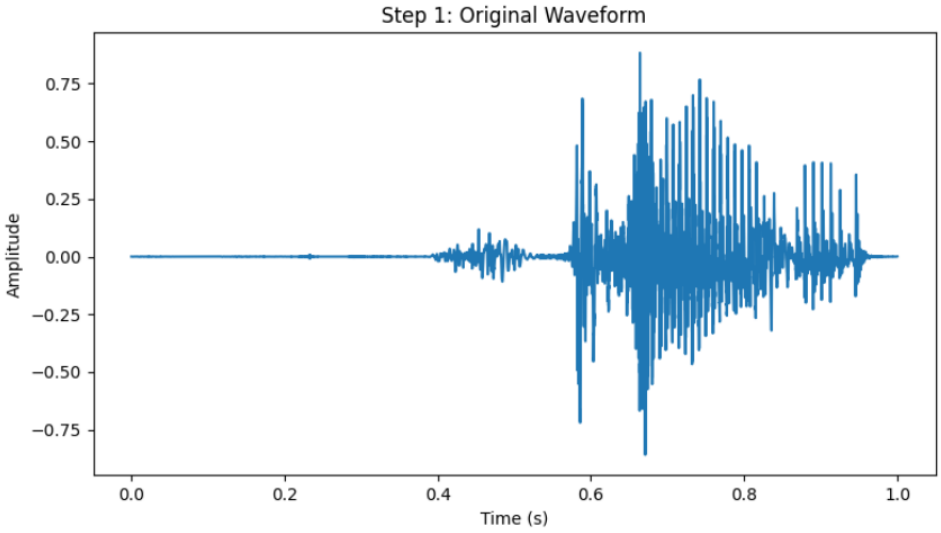

Figure 3.2: Raw audio waveform of a one-second speech sample from the Google Speech Commands dataset. The waveform shows the amplitude variations over time that represent the acoustic signal before any feature extraction.

stationary. The output is a spectrogram, a 2D plot of frequency vs. time, with the magnitude of each frequency component at each time step represented by a value.

For a discrete signal $x[n]$, the STFT is defined as:

$$X[m,k] = \sum_{n=0}^{N-1} x[n+mH] \cdot w[n] \cdot e^{-j2\pi kn/N} \tag{3.1}$$

where m is the frame index, k is the frequency bin index, H is the hop length, $w[n]$ is the window function (typically a Hann window), and N is the FFT size. The parameters `n_fft` (Fast Fourier Transform size) and `hop_length` (the step size between windows) are crucial to this step.

Mel Scale Conversion

Our ears are not linearly sensitive to frequency. The Mel scale is a non-linear scale of pitches judged by listeners to be equal in distance from one another. We apply a series of overlapping triangular filters, known as a Mel filter bank, to the linear frequency-domain spectrogram. This process weights the frequencies according to the Mel scale, effectively "remapping" the spectrogram to a representation that is more perceptually uniform for human ears.

The conversion from frequency (in Hz) to the Mel scale is given by:

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{3.2}$$

where f is the frequency in Hz and m is the corresponding value in mels. The Mel filter bank applies M triangular filters to the power spectrum, producing a Mel spectrogram:

$$S_{\text{mel}}[m,k] = \sum_{n=0}^{N/2} |X[m,n]|^2 \cdot H_k[n] \tag{3.3}$$

where $H_k[n]$ represents the k-th triangular filter in the Mel filter bank. The number of these filters, `n_mels`, determines the resolution of the frequency axis in the final output.

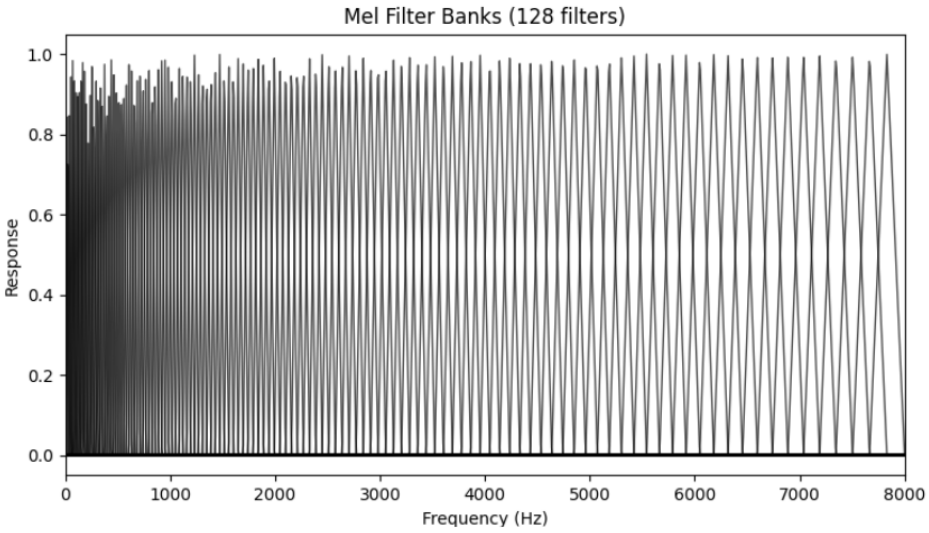

Figure 3.3: Mel filter bank showing the triangular filters applied to the linear frequency spectrum. Each filter has a triangular shape with overlapping regions, and the filters are spaced according to the Mel scale, providing higher resolution at lower frequencies where human hearing is more sensitive.

Logarithmic Scaling

The final step is to convert the power of these Mel-scaled frequencies to a logarithmic scale, typically in decibels (dB). This is done because human perception of volume is also logarithmic. A small change in sound pressure at low volumes is perceived much more acutely than the same change at high volumes. Converting the power values to a log scale makes the representation more robust to changes in volume and more closely matches human perception.

The logarithmic transformation is applied as:

$$S_{\log}[m, k] = 10 \log_{10}(S_{\text{mel}}[m, k] + \epsilon) \tag{3.4}$$

where ϵ is a small constant (e.g., 10^{-10}) added for numerical stability to avoid taking the logarithm of zero. This produces the final Mel spectrogram in decibels.

The result of this pipeline is a Mel spectrogram: a 2D "image" of the audio signal where the axes are time and Mel-scaled frequency. This image-like representation is an excellent input for neural networks, especially those with convolutional front-ends.

This entire process is encapsulated in our model's `WaveToSpec` class, as shown in the code below. The `feature_type` parameter allows us to switch between a Mel spectrogram and MFCCs.

```
class WaveToSpec:
    def __init__(self,
                 feature_type: str = "mel",
                 sample_rate: int = 16000,
                 n_fft: int = 2048,
                 hop_length: int = 256,
```

```
                 n_mels: int = 128,
                 n_mfcc: int = 40,
                 top_db: int | None = 80,
                 apply_mask: bool = True,
                 freq_mask_param: int = 15,
                 time_mask_param: int = 10):

        self.feature_type = feature_type.lower()
        assert self.feature_type in {"mel","mfcc"}
        self.apply_mask = apply_mask and self.feature_type == "mel"

        if self.feature_type == "mel":
            self.spec = T.MelSpectrogram(sample_rate, n_fft, hop_length,
                                         n_mels, power=2)
            self.to_db = T.AmplitudeToDB(stype="power", top_db=top_db)
            if self.apply_mask:
                self.freq_mask = T.FrequencyMasking(freq_mask_param)
                self.time_mask = T.TimeMasking(time_mask_param)
        else:
            self.spec = T.MFCC(sample_rate, n_mfcc,
                               melkwargs=dict(n_fft=n_fft,
                                              hop_length=hop_length,
                                              n_mels=n_mels))
            self.to_db = None
            self.freq_mask = self.time_mask = None
```

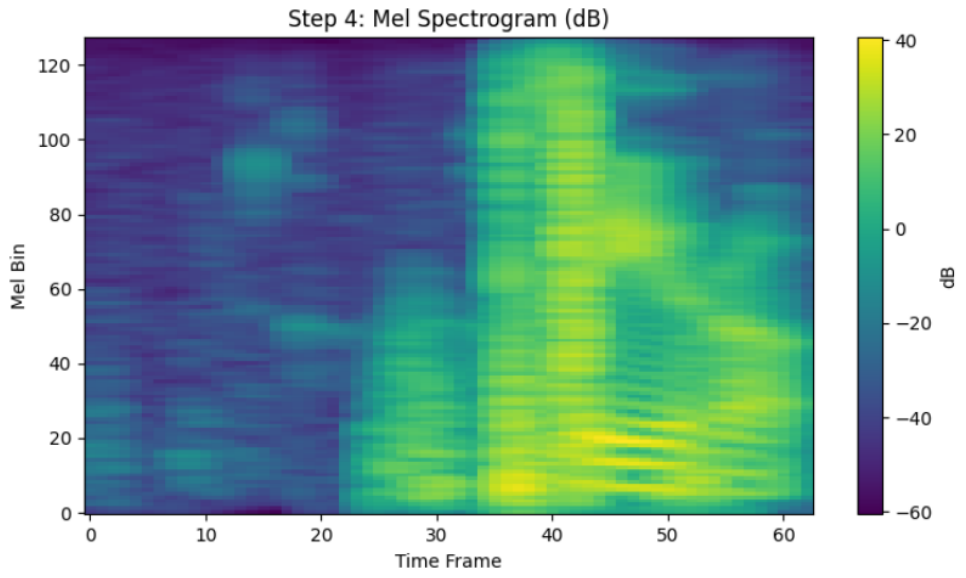

Figure 3.4: Mel spectrogram in decibels showing the final result of the STFT → Mel → Log pipeline. The horizontal axis represents time, the vertical axis represents Mel-scaled frequency bins, and the color intensity represents the logarithmic power in dB. This 2D representation captures both spectral and temporal characteristics of the speech signal.

3.3.2 Mel-Frequency Cepstral Coefficients (MFCCs)

As an alternative to Mel spectrograms, MFCCs are a feature representation widely used in speech recognition. MFCCs are derived from a Mel spectrogram by applying a Discrete Cosine Transform (DCT) to the logarithmically-scaled Mel filter bank outputs. The purpose of the DCT is to de-correlate the filter bank energies, producing a set of coefficients that are a more compact and abstract representation of the audio's spectral envelope.

The MFCC computation involves applying the DCT to the log Mel spectrogram:

$$\text{MFCC}[m, c] = \sum_{k=0}^{M-1} S_{\log}[m, k] \cos\left(\frac{c(k + 0.5)\pi}{M}\right) \tag{3.5}$$

where $c = 0, 1, \ldots, C - 1$ are the cepstral coefficient indices, M is the number of Mel filter banks, and C is the desired number of MFCC coefficients. Typically, only the first 12–13 coefficients are retained, as higher-order coefficients often contain noise.

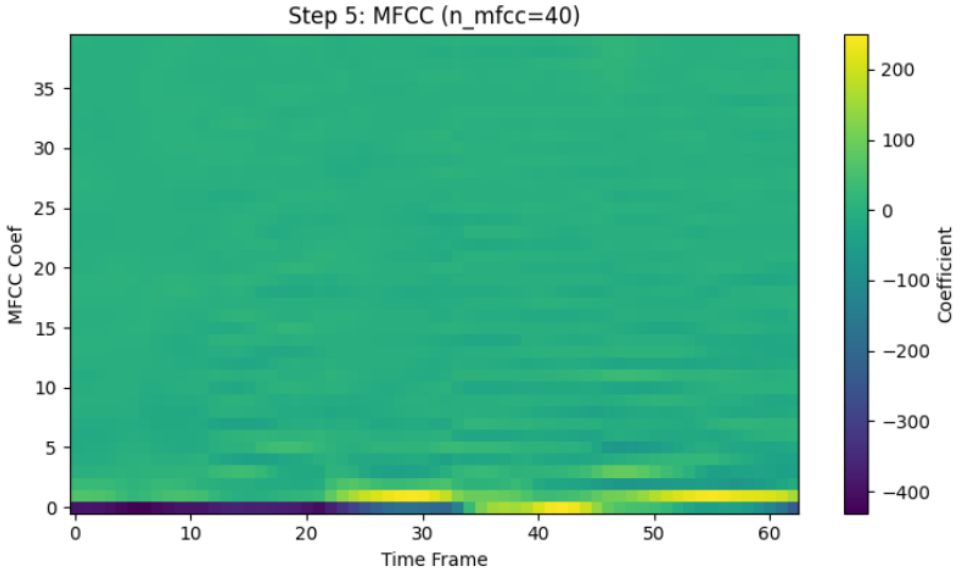

Figure 3.5: MFCC coefficients showing the result of applying the Discrete Cosine Transform (DCT) to the log Mel spectrogram. The horizontal axis represents time, and the vertical axis represents the cepstral coefficient indices. Lower-order coefficients (bottom) capture the spectral envelope, while higher-order coefficients represent finer spectral details.

While MFCCs can reduce the computational load on the neural network, they come at the cost of increased preprocessing complexity. Our project code allows for MFCC feature extraction, and our experiments showed that a model trained on MFCCs achieved a slightly lower accuracy compared to the Mel spectrogram variant, which suggests the latter provided richer information for our specific architecture.

3.4 Augmentations

In deep learning, data augmentation is a technique used to artificially increase the size and diversity of a training dataset by applying random transformations to the existing data. This is crucial for improving a model's ability to generalize to new, unseen data and to make it robust to real-world variations. We implemented both waveform-level and spectrogram-level augmentations. It is a critical design choice, explicitly mentioned in the documentation of our code: "Maintains separate front-ends for train vs. eval (no masks in eval)."

3.4.1 Waveform-Level Augmentation

These techniques modify the raw audio signal before it is converted into a feature representation. We applied two simple but effective methods, both handled by our `Augment` class:

Time Shifting

A random shift is applied to the audio waveform, moving it forward or backward by a small, random number of samples. This teaches the model that the keyword can appear at any point in the one-second clip, rather than being fixed to a specific position. This is implemented by padding the audio tensor at the beginning or end, and then cropping it to the original length, as seen in the `_shift` method.

Noise Injection

A small amount of random Gaussian noise is added to the signal. This helps the model become more robust to background noise and different acoustic environments. The noise injection process can be expressed as:

$$x_{\text{aug}}[n] = x[n] + \mathcal{N}(0, \sigma^2) \tag{3.6}$$

where $x[n]$ is the original signal, $\mathcal{N}(0, \sigma^2)$ is zero-mean Gaussian noise with variance σ^2, and σ is randomly sampled from a predefined range during training. The magnitude of this noise is controlled by a randomly sampled sigma value.

3.4.2 Spectrogram-Level Augmentation (SpecAugment)

These augmentations are applied directly to the computed Mel spectrogram, treating it as a 2D image. Our `WaveToSpec` class includes an `apply_mask` parameter that controls this process. This implementation, known as SpecAugment, uses two types of masking:

Frequency Masking

Random blocks of Mel frequency channels are masked (set to zero) on the spectrogram. This forces the model to learn features from a wider range of frequencies rather than relying on a single, narrow band. In our code, this is handled by `T.FrequencyMasking`. The `freq_mask_param` controls the size of the masked region.

Time Masking

Random blocks of time steps are masked on the spectrogram. This encourages the model to learn context from the entire sequence rather than from a single, specific moment in time. This is handled by `T.TimeMasking`. The `time_mask_param` controls the size of the masked region along the time axis.

As noted in the code, our implementation of SpecAugment applies both frequency and time masking twice, for a total of four masks, to provide a stronger regularization effect and further improve the model's ability to generalize. These augmentations are only enabled for the training dataset and are explicitly disabled for the validation and test datasets to ensure a fair and unbiased evaluation of the model's performance.

Chapter 4

Model Architecture & Methodology

4.1 Overall Pipeline

The end-to-end process of taking a raw audio file and producing a classification prediction requires a cohesive pipeline that integrates data handling, feature extraction, and the neural network architecture itself. Our system is designed as a series of interconnected stages, from raw data ingestion to a final output logit vector.

The pipeline begins with the raw audio input, which is a one-dimensional waveform of a single one-second clip. This raw data is loaded from the Google Speech Commands dataset using the `datasets` library, as shown in the main script. The raw audio is then fed into our data preprocessing classes:

```
# Datasets
train_ds = SpeechCommands(ds["train"], aug, frontend_train,
                          mean=train_mean, std=train_std)
val_ds   = SpeechCommands(ds["validation"], None, frontend_eval,
                          mean=train_mean, std=train_std)
test_ds  = SpeechCommands(ds["test"], None, frontend_eval,
                          mean=train_mean, std=train_std)
```

The `SpeechCommands` class, which acts as a wrapper around the dataset, first handles optional waveform-level augmentations via the `Augment` class. This includes applying random time shifts and adding noise to the waveform. The raw waveform is then passed to the `WaveToSpec` class, which performs the core feature extraction, converting the one-dimensional signal into a two-dimensional log-Mel spectrogram or MFCC feature map.

After feature extraction, a crucial normalization step is performed, as shown in the `__getitem__` method of the `SpeechCommands` class:

```
feats = self.front(wav)           # [C=1, 40, ~55] for MFCC
feats = (feats - self.mean) / (self.std + 1e-6)  # Normalize with
    precomputed stats
feats = feats.squeeze(0).transpose(0, 1)
```

This normalization ensures that the feature maps have zero mean and unit variance:

$$\mathbf{x}_{\text{norm}} = \frac{\mathbf{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma} + \epsilon} \tag{4.1}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the precomputed mean and standard deviation, and $\epsilon = 10^{-6}$ prevents division by zero. This is a standard practice to improve network training stability and convergence speed.

The final feature tensor is then fed to a `DataLoader` to be batched and prepared for model training or inference. Because audio samples in the dataset can have slightly different lengths, we use a custom `collate_fn` to pad the feature maps to a uniform length within each batch, while also retaining the true lengths to be used later in the model's forward pass.

The data then enters our `MambaKWS` neural network model, which can be thought of as a four-stage process:

1. **Convolutional Embedding (`conv_embed`):** The 2D feature map is first processed by a series of convolutional layers. This front-end acts as a feature extractor, learning to identify local patterns in the time-frequency domain.

2. **Linear Projection (`proj`):** The output of the convolutional layers is flattened and projected to a higher-dimensional vector space, matching the `d_model` dimension of the Mamba blocks.

3. **Mamba Blocks:** The core of the model consists of a stack of Mamba blocks. These layers process the sequential data, modeling long-range dependencies and contextual understanding of the entire input.

4. **Classifier Head:** The outputs of the Mamba layers are pooled across the time dimension and fed into a final classifier, which outputs a probability distribution over the possible keywords.

The entire process is orchestrated by our main script, which loads the data, initializes the model, and then proceeds with the training loop, managing the forward and backward passes.

4.2 MambaKWS Model Implementation

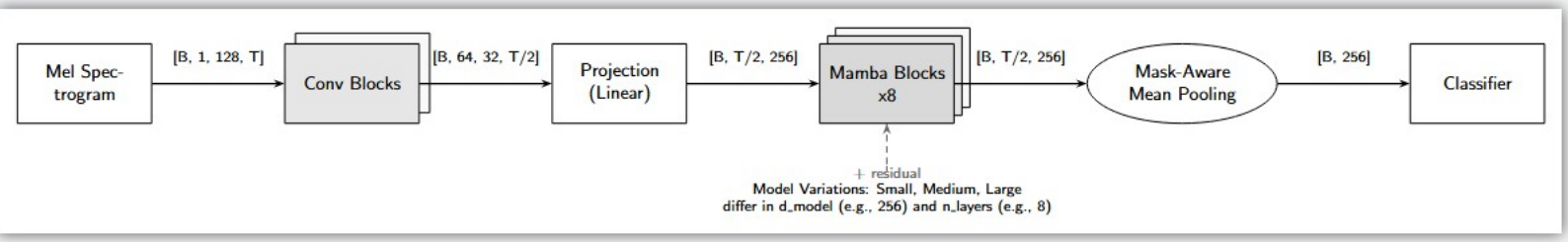

Figure 4.1: Our Mamba architecture for keyword spotting. The model combines convolutional layers for local feature extraction with Mamba blocks for sequence modeling.

Our model architecture, encapsulated within the `MambaKWS` class, is a hybrid design that combines the strengths of both convolutional and state-space models. This is a deliberate design choice that was proven to be more effective than simply feeding the raw feature map directly into Mamba blocks. As noted in our project analysis, a model that directly projects the Mel spectrograms with a single linear layer "underuses local

structure," causing the learning to "stall." Our hybrid approach addresses this by first extracting robust local features before processing the global sequence.

The `MambaKWS` class is defined with the following key components:

4.2.1 Convolutional Front-end

This section of the model acts as an initial feature extractor. It takes the input Mel spectrogram, which has been reshaped to (B, C, F, T) format (Batch, Channels, Frequency, Time), and processes it with a series of 2D convolutional layers.

```
self.conv_embed = nn.Sequential(
    nn.Conv2d(in_ch, 32, 3, padding=1),
    nn.BatchNorm2d(32), nn.SiLU(),
    nn.Conv2d(32, 32, 3, padding=1),
    nn.BatchNorm2d(32), nn.SiLU(),
    nn.MaxPool2d(2),

    nn.Conv2d(32, 64, 3, padding=1),
    nn.BatchNorm2d(64), nn.SiLU(),
    nn.Conv2d(64, 64, 3, padding=1),
    nn.BatchNorm2d(64), nn.SiLU(),
    nn.MaxPool2d((2, 1)),
)
```

The stack consists of two pairs of `Conv2d` layers, each followed by `BatchNorm2d` and a `SiLU` activation function. The `MaxPool2d` layers downsample the feature map, reducing its dimensionality and making the model more robust to minor translations. The use of 2D kernels allows the model to mix adjacent time-frequency bins, capturing the local neighborhood structure that is crucial for audio signal analysis. This process learns a richer, compressed representation of the audio signal before it is passed to the sequence-modeling layers.

4.2.2 Projection Layer

After the convolutional layers, the feature map is reshaped into a flattened sequence, where each time step is a vector. This vector is then passed to a linear projection layer that maps it to the dimensionality required by the Mamba blocks.

```
self.proj = nn.Sequential(
    nn.Linear(flattened_dim, d_model),
    nn.LayerNorm(d_model),
    nn.SiLU(),
    nn.Dropout(0.1)
)
```

This layer is critical for bridging the two parts of our hybrid architecture. It ensures that the output from the convolutional front-end, which has a potentially large and varying dimension, is correctly transformed into a uniform tensor of shape (B, T', D), which is the standard input format for Mamba.

4.2.3 Mamba Blocks

The core of our model is a stack of Mamba blocks, as specified in `n_layers`. Each block is implemented with a residual connection, LayerNorm, a Mamba layer, and Dropout:

```
self.blocks = nn.ModuleList()
for _ in range(n_layers):
    self.blocks.append(nn.Sequential(
        ResidualBlock(
            MambaBlock(d_model, d_state, d_conv, expand),
            nn.LayerNorm(d_model),
            nn.Dropout(0.1)
        )
    ))
```

The Mamba layer itself contains the selective state-space model that processes the sequence. It operates with linear time complexity $O(L)$ and is able to model long-range dependencies, making it an excellent fit for the temporal nature of audio data. The residual connection helps prevent the vanishing gradient problem and facilitates the training of deep networks.

4.2.4 Classifier Head and Pooling

The output of the final Mamba block is a tensor of shape (B, T', D). To convert this to a single output for classification, we need to collapse the time dimension T'. We achieve this through mask-aware mean pooling:

```
if lengths is not None:
    t_lens = torch.div(lengths, 2,
    rounding_mode='floor').clamp(min=1).to(x.device)
    Tprime = x.size(1)
    mask = (torch.arange(Tprime, device=x.device)[None, :] < t_lens[:,
    None]).float()
    mask = mask.unsqueeze(-1)
    x_sum = (x * mask).sum(dim=1)
    denom = mask.sum(dim=1).clamp(min=1.0)
    pooled = x_sum / denom
else:
    pooled = x.mean(dim=1)
```

This pooling mechanism uses the `lengths` tensor, which we returned from our custom `collate_fn`, to compute an accurate mean of the sequence by ignoring the padded values. This is a more robust approach than naive mean pooling, as it handles variable-length inputs without introducing bias from padded zeros.

4.3 Model Variants

To assess the trade-offs between model complexity, computational efficiency, and accuracy, we developed three distinct variants of our `MambaKWS` model. These variants, named Small, Medium, and Large, share the same core architecture but differ in their key hyperparameters, as summarized in Table 4.1.

- **d_model:** This parameter, also known as the model's width, determines the dimensionality of the vector representation for each time step within the Mamba blocks.

Model Variant	d_model	n_layers	d_state	expend
Mamba-Small	64	8	16	Default (2)
Mamba-Medium	128	10	16	Default (2)
Mamba-Large	192	12	16	Default (2)

Table 4.1: Model variants with their architectural hyperparameters.

A larger `d_model` allows the model to capture more information, leading to a greater number of parameters and, in our case, a corresponding increase in accuracy.

- **n_layers:** This parameter defines the depth of the model, or the number of stacked Mamba blocks. A deeper model can learn more complex hierarchical features. Our variants scale the number of layers with the model's width, ranging from 8 in the Small variant to 12 in the Large variant.
- **d_state:** This parameter, which controls the size of the latent state **h** within the Mamba's selective scan operation, was kept constant at 16 across all three variants. This was done to isolate the effects of the `d_model` and `n_layers` on performance while maintaining the core memory mechanism of the SSM at a consistent level.
- **expend:** This parameter controls the expansion factor within the Mamba block's internal feed-forward network. It determines how much the intermediate hidden dimension is expanded relative to `d_model` during the block's internal computations. A value of 2 (the default) means the internal dimension is twice the size of `d_model`, providing a balance between model expressiveness and computational efficiency. All variants use the default expansion factor to maintain consistency across different model sizes.

The implementation details for each model variant are found in our main script:

```
# Small model
model = MambaKWS(n_classes, d_model=64, d_state=16,
    n_layers=8).to(device)

# Medium model (our main subject of study)
model = MambaKWS(n_classes, d_model=128, d_state=16,
    n_layers=10).to(device)

# Large model
model = MambaKWS(n_classes, d_model=192, d_state=16,
    n_layers=12).to(device)
```

These model variants provide a systematic exploration of the trade-offs between model complexity and computational efficiency. The scaling of both width (`d_model`) and depth (`n_layers`) allows for comprehensive analysis of how architectural choices impact both accuracy and resource requirements on edge devices. The detailed performance analysis and comparison of these variants will be presented in Chapter 5.

4.4 Training Setup

The high-performance and stable training of deep sequence models, such as Mamba, relies heavily on strategic choices for optimization, loss function, and learning rate scheduling.

Our approach utilized advanced techniques to push model performance while maintaining convergence stability.

4.4.1 Loss Function and Regularization

For our multi-class keyword spotting task (35 classes), we employed the standard Cross-Entropy Loss. We added Label Smoothing with a factor of 0.07:

```
criterion = nn.CrossEntropyLoss(label_smoothing=0.07)
```

Label smoothing is a regularization technique that replaces the hard one-hot target probabilities (where the correct class y has probability 1 and all others have 0) with a smoother distribution. This prevents the model from becoming overly confident in its predictions and improves generalization, particularly preventing overfitting in high-capacity models like our Large Mamba variant.

4.4.2 Optimizer and Weight Decay

We selected the AdamW optimizer over the standard Adam. AdamW is preferred in modern deep learning as it correctly separates weight decay from the adaptive gradient updates, applying it directly to the weights for better regularization.

Our training uses a base learning rate of 5×10^{-4} across all model variants, with 100 training epochs and a warmup fraction of 0.12 (12% of total training steps). The batch size is set to 64 for all experiments.

The core training parameters show a strategic difference across our model variants, particularly concerning regularization:

Model Variant	d_model	weight_decay	Rationale
Small/Medium	64/128	1.8×10^{-4}	Standard regularization for initial stability
Large	192	5.0×10^{-4}	Increased weight decay for stronger regularization

Table 4.2: Weight decay scaling across model variants.

The Large model has over 3 million parameters, necessitating stronger L2 regularization to suppress overfitting. This deliberate scaling of the `weight_decay` hyperparameter ensures that the increased capacity of the Large model translates into better generalization rather than just overfitting the training set.

4.4.3 One-Cycle Learning Rate Scheduling

To achieve fast convergence and high final accuracy, we utilized a custom One-Cycle Learning Rate (LR) Schedule, based on a linear warm-up followed by a cosine decay:

```
# Defines your learning rate schedule
def lr_lambda(step):
    if step < warmup_steps:
        return float(step) / float(max(1, warmup_steps))
    progress = float(step - warmup_steps) / float(max(1, total_steps - warmup_steps))
    return max(0.005, 0.5 * (1.0 + math.cos(math.pi * progress)))
```

The schedule operates in two phases, where the learning rate η is updated per batch (`sched.step()` is called inside the inner loop):

Linear Warmup

The LR starts near zero and linearly increases to the peak base rate (`base_lr`) over a fraction of the total steps (`warmup_frac = 0.12`). This gradual increase prevents early large gradients from destabilizing the randomly initialized weights:

$$\eta_{\text{warmup}}(t) = \text{base\_lr} \cdot \frac{t}{\text{warmup\_steps}} \tag{4.2}$$

Cosine Decay

After the warmup, the LR decays smoothly towards a minimum floor ($0.005 \cdot \text{base\_lr}$) following a cosine function. This soft decay allows the model to explore the error surface and settle into a robust minimum:

$$\eta_{\text{cosine}}(t) = \max(\eta_{\min}, 0.5 \cdot \text{base\_lr} \cdot (1 + \cos(\pi \cdot \text{progress}))) \tag{4.3}$$

4.4.4 Training Stability Enhancements (Hardware-Aware)

The training loop incorporates several techniques essential for managing deep networks on GPU hardware:

Mixed Precision Training (AMP)

We employed Automated Mixed Precision (`torch.amp.autocast` with `bfloat16`) and a `GradScaler`. This leverages the GPU's Tensor Cores for faster Multiply-Accumulate (MAC) operations and cuts the memory footprint of activation tensors in half. This is vital for fitting large batch sizes and deep models onto modern GPU memory.

Gradient Clipping

To prevent the exploding gradient problem, common in deep architectures with recurring layers (like Mamba's SSM blocks), gradients were clipped to a maximum norm of 0.3:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=0.3)
```

This ensures that excessive gradient magnitudes do not corrupt the model's weights during backpropagation.

Collapse Guard

A critical mechanism was implemented to monitor for catastrophic divergence during training. If the validation accuracy suddenly dropped by more than 50% of the previous best accuracy, the model would halt, reload the best-performing weights from the checkpoint, and reduce the learning rate by a factor of 5.0. This robust guard ensures that training can recover from transient instabilities without restarting the entire run.

4.5 Inference Benchmarking Methodology

A central tenet of this project is validating that the algorithmic efficiency of the Mamba architecture—its $O(L)$ complexity—translates into measurable, real-world performance

benefits for the KWS task. Since conventional metrics like FLOPs do not account for modern memory bottlenecks, we implemented a custom benchmarking suite to directly measure key hardware efficiency indicators: latency, throughput, and memory consumption.

4.5.1 Latency Benchmark (Time per Sample)

Latency is the single most critical metric for an "always-on" KWS system, as it determines the delay between a keyword being spoken and the system reacting. High latency leads to missed activation windows and poor user experience. We measured the average latency for a single sample inference ($B = 1$) to determine the real-time capability of each model variant.

Process

The benchmark runs the forward pass of the model multiple times (`num_runs = 1000`) after a set of initial warm-up runs (10 runs) to stabilize the GPU cache and clock rates. We explicitly synchronize the CUDA threads (`torch.cuda.synchronize()`) before and after timing to ensure all GPU operations are complete before recording the time:

```
start_time = time.perf_counter()
_ = model(dummy_input, dummy_lengths)
if device.type == 'cuda':
    torch.cuda.synchronize()
end_time = time.perf_counter()
times.append((end_time - start_time) * 1000)  # Convert to ms
```

Metric

Mean latency in milliseconds (ms), which provides a stable average of single-sample processing time. We also track the P95 and P99 latency to capture the worst-case time delays, ensuring robust performance guarantees for real-time applications.

4.5.2 Throughput Benchmark (Samples per Second)

Throughput measures the system's overall capacity to process data, which is crucial for maximizing device utilization or handling multiple concurrent streams. This metric is defined as the number of samples the model can process per second (sps).

Process

The script systematically tests the model across a range of increasing batch sizes (B : $1, 2, 4, 8, 16, 32$). For each batch size, we measure the total inference time over multiple runs, and then calculate the throughput:

$$\text{Throughput} = \frac{\text{Batch Size}}{\text{Average Inference Time (s)}} \tag{4.4}$$

Significance

Measuring throughput versus batch size reveals how well the model parallelizes on GPU hardware. As the model size increases (Small to Large), we observe the efficiency penalty

of scaling the model complexity. This analysis is crucial for understanding deployment trade-offs between accuracy and computational efficiency.

4.5.3 Memory Consumption Benchmark

For resource-constrained environments, the memory footprint during inference is a hard constraint. While the model parameter count determines the static memory size, the peak memory usage (activations and transient data) determines deployment feasibility.

Process

This benchmark is only run on the CUDA device. It uses PyTorch's native memory tracking (`torch.cuda.max_memory_allocated()`) to determine the total memory allocated during the forward pass:

```
torch.cuda.reset_peak_memory_stats()
baseline_memory = torch.cuda.memory_allocated()
# Run inference
_ = model(dummy_input, dummy_lengths)
peak_memory = torch.cuda.max_memory_allocated()
activation_memory = peak_memory - baseline_memory
```

Metrics

We track the memory consumed by the model weights themselves (baseline) and the activation memory (the memory dynamically used for intermediate calculations during the forward pass). This distinction is critical because Mamba's hardware-aware design is explicitly meant to avoid materializing large intermediate states, thus minimizing the activation memory overhead compared to quadratic models like Transformers.

The memory efficiency can be quantified as:

$$\text{Memory per Sample} = \frac{\text{Activation Memory}}{\text{Batch Size}} \tag{4.5}$$

This metric allows direct comparison of memory scaling behavior across different model architectures and provides insights into the practical deployment constraints for edge devices.

4.6 Baselines for Comparison (CNN, RetNet)

To rigorously validate the performance and hardware efficiency of the novel MambaKWS architecture, a comparative study was conducted against two foundational baselines. This comparison was designed to control for variables, ensuring that any observed gains could be directly attributed to the Selective State-Space Model (SSM) mechanism rather than superior feature engineering or architectural depth.

The two main comparative benchmarks represent the state-of-the-art in their respective domains: the efficient CNN-based model (MobileNetV2) from the pre-Transformer era, and the Retentive Network (RetNet), which is a direct competitor in the post-Transformer $O(L)$ complexity space.

4.6.1 CNN-Based Architecture (MobileNetV2)

This model serves as the primary benchmark against the established efficiency paradigm for spatial data processing on edge devices. CNNs excel at extracting local patterns (like edges or textures in images) but inherently struggle with long-range temporal dependencies.

Architecture and Rationale

We selected a customized version of MobileNetV2 with a width multiplier (α) of 0.75 as a highly optimized baseline. This is a state-of-the-art model known for achieving high accuracy with a minimal parameter count by relying on depth-wise separable convolutions (which drastically reduce computational load) and inverted residual blocks. The primary goal of testing against MobileNetV2 was two-fold:

- **Local Feature Control:** To assess the architectural trade-off between Mamba's explicit long-range sequence modeling versus the implicit temporal aggregation performed by purely convolutional layers.
- **Efficiency Baseline:** To establish the minimum viable accuracy and efficiency floor achieved by the best existing CNN architectures designed for TinyML. The model achieved $\approx 96.5\%$ accuracy.

Implementation Adaptation

The canonical MobileNetV2 architecture was adapted for KWS by adjusting its initial layer to accept our 1-channel log-Mel spectrogram input:

```
def build_mobilenet_v2(num_classes: int, alpha: float = 0.75):
    net = tvm.mobilenet_v2(width_mult=alpha, weights=None)
    # Adapt first conv to 1 channel
    first = net.features[0][0]
    net.features[0][0] = nn.Conv2d(1, first.out_channels,
                                   kernel_size=first.kernel_size,
                                   stride=first.stride,
   padding=first.padding,
                                   bias=False)
```

Furthermore, the pooling mechanism was replaced with `nn.AdaptiveAvgPool2d(1)` to produce a fixed-size classification vector, enabling the model to handle the variable input dimensions present in the audio dataset.

4.6.2 Retentive Network (RetNet) Model

The Retentive Network (RetNet) is a direct contemporary of Mamba in the post-Transformer landscape, achieving linear time complexity ($O(L)$) by replacing the quadratic attention mechanism with a Retention mechanism.

Architectural Equivalence (Apples-to-Apples Comparison)

Crucially, the RetNetKWS model was engineered to share the exact same convolutional front-end and projection layer design as our MambaKWS models. This methodological choice ensures that the comparison focuses purely on the difference in the deep sequence blocks (Selective SSM vs. Retention), eliminating variance caused by input feature differences.

- **Shared Feature Extraction:** Both MambaKWS and RetNetKWS utilize the identical CNN stack (`self.conv_embed`) to perform initial 2D feature aggregation and dimensionality reduction.
- **Shared Projection:** Both models use the identical `self.proj` layer to map the convolutional output to the internal embedding dimension (`d_model=128`) before sequence processing begins.

RetNet Block Structure

The RetNetKWS substitutes the Mamba layer with the custom `RetNetBlock`, which is built around the `MultiScaleRetention` component:

```
class RetNetKWS(nn.Module):
    def __init__(self, num_classes, d_model=128, n_layers=6, n_heads=8):
        # ... shared conv_embed and proj layers ...
        self.blocks = nn.ModuleList([
            RetNetBlock(d_model=d_model, n_heads=n_heads, drop=dropout)
            for _ in range(n_layers)
        ])
        # ... classifier head ...
```

The implementation of the `RetNetBlock` follows a pre-normalization residual structure similar to Transformer layers, using linear transformations for Query (Q), Key (K), and Value (V) tensors, which are then processed by the `MultiScaleRetention` component. This system is fundamentally different from Mamba's recurrent state transition:

$$\text{Retention}(Q, K, V) \approx \sum_t M_t (K_t \cdot V_t) \tag{4.6}$$

The RetNet model achieved stable training at $\approx 97.2\%$ accuracy but was observed to be much bigger parameter-wise in terms of model size and memory usage, a crucial point of efficiency that our analysis aims to quantify in Chapter 5.

4.6.3 Mamba Baseline: MFCC-input variant

A third, Mamba-based baseline involved training the MambaKWS model using MFCC features combined with a simple linear projection (bypassing the CNN front-end). This config study was critical for verifying the efficacy of our preprocessing choices.

Architecture

The config follows the structure: MFCC $\rightarrow$ Linear Projection $\rightarrow$ Mamba Stack, where the 2D convolutional front-end is replaced with a direct linear mapping from MFCC coefficients to the Mamba input dimension.

Results and Conclusions

As documented in the config summary, this model performed poorly ($\approx 96.5\%$ accuracy). This demonstrated two important facts:

- **Feature Richness:** The raw Log-Mel Spectrogram, when properly processed by a learned CNN front-end, carries richer, more robust information than the pre-correlated MFCC features.
- **Front-End Necessity:** The CNN stack is essential for Mamba's success in KWS, as it provides the local feature mixing and compression required to transform the 2D input into a high-quality 1D sequence for the SSM layers.

This config validates our hybrid architecture design, confirming that the combination of convolutional feature extraction with Mamba's sequence modeling creates a synergistic effect that neither component achieves alone.

Chapter 5

Experiments & Results

This chapter presents a comprehensive evaluation of our Mamba-based keyword spotting models, detailing their performance across multiple dimensions. We conduct extensive experiments to validate our architectural choices, demonstrate state-of-the-art accuracy, and provide insights into the models' behavior under various conditions. Our experimental framework encompasses accuracy benchmarks, efficiency analysis, scalability studies, and deployment feasibility assessments.

5.1 Main Accuracy Results

Our experimental evaluation demonstrates that Mamba-based architectures achieve competitive and often superior performance compared to traditional CNN approaches for keyword spotting. We trained three model variants—Small (S), Medium (M), and Large (L)—each representing different points in the accuracy-efficiency trade-off space.

5.1.1 Model Configurations and Performance

The three Mamba model variants were designed to span a range of computational complexities while maintaining the core architectural principles established in Chapter 4. Each configuration differs primarily in the number of layers, state dimension (`d_state`), and model dimension (`d_model`), allowing us to study the scaling behavior of the selective SSM mechanism.

Our hyperparameter exploration, conducted through extensive TensorBoard experimentation across 25 epochs for each configuration, revealed that the combination of convolutional front-end with Mel spectrogram input consistently outperformed alternative approaches. The systematic evaluation of `d_state` values (16), `d_model` dimensions (64, 128, 192), layer counts (8, 10, 12), and expansion factors (2, 4) provided comprehensive insights into the selective SSM scaling behavior.

5.1.2 Comparison with Baseline Architectures

To establish the effectiveness of our approach, we implemented and evaluated several baseline architectures using identical training procedures and data preprocessing pipelines. This methodological consistency ensures that performance differences can be attributed to architectural choices rather than implementation artifacts.

The results demonstrate several key findings:

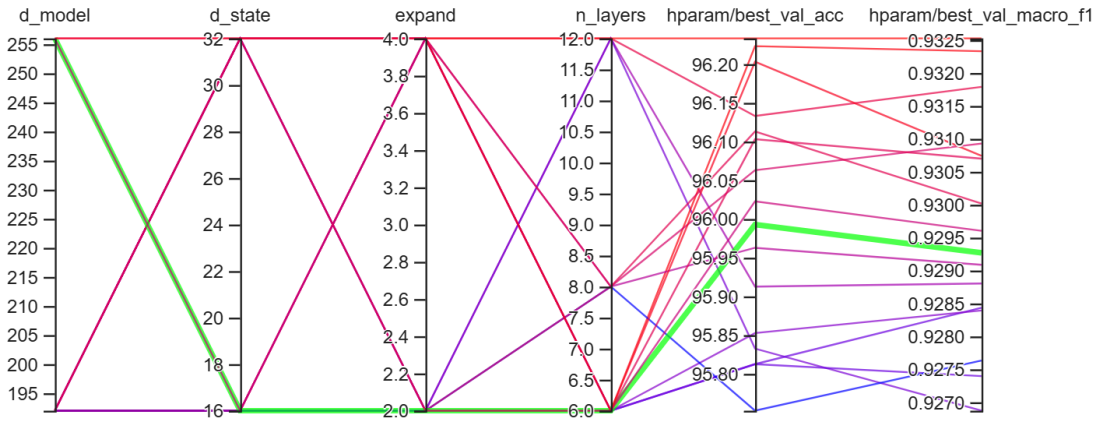

Figure 5.1: TensorBoard parallel coordinates plot showing the exploration of different hyperparameter configurations during model development. Each line represents a different configuration, with coordinates corresponding to d_state, d_model, number of layers, and expansion factor. The color intensity indicates validation accuracy, helping identify optimal parameter combinations.

Table 5.1: Accuracy Comparison Across Different Architectures

Architecture	**Parameters**	**Input Type**	**Accuracy (%)**
MobileNetV2 (alpha=0.75)	450K	Mel Spectrogram	96.5
RetNet-KWS	1.6M	Mel Spectrogram	97.2
Mamba-KWS (MFCC)	404K	MFCC	96.5
Mamba-S	404K	Mel Spectrogram	97.42
Mamba-M	1.34M	Mel Spectrogram	97.59
Mamba-L	3.17M	Mel Spectrogram	97.75

- **Architectural Superiority:** Even our smallest Mamba model (Mamba-S) with 404K parameters achieves 97.42% accuracy, significantly outperforming both the MobileNetV2 baseline (96.5% with 450K parameters) and demonstrating superior parameter efficiency.
- **Feature Representation Impact:** The comparison between Mamba-KWS with MFCC input (96.5%) and Mel spectrogram input (97.42%) validates our choice of Mel spectrograms as the preferred input representation, highlighting the importance of preserving spectral detail.
- **Post-Transformer Competition:** Our Mamba models demonstrate competitive performance with the RetNet baseline. While Mamba-M (97.59% with 1.34M parameters) shows comparable accuracy to RetNet (97.2% with 1.6M parameters), it achieves this with superior parameter efficiency. Notably, our largest model Mamba-L (97.75% with 3.17M parameters) achieves the best overall accuracy while maintaining reasonable model size.

5.1.3 Input Configuration Analysis

Our systematic evaluation of different input processing approaches revealed important insights about the interaction between feature extraction and the Mamba architecture:

Linear-Projection + Mel → Plateau

Directly projecting Mel spectrograms with a single linear layer underuses local structure; there's no neighborhood mixing across time-frequency, so the model's learning stalls. This approach failed to extract meaningful patterns from the 2D spectrogram structure, leading to poor performance.

Conv Embedding Helps

A shallow conv stack mixes adjacent time–freq bins (2D kernels), adds nonlinearity and pooling, and learns richer, compressed features before projection. This convolutional front-end proved essential for our Mamba models, enabling them to achieve the reported accuracies by properly processing the spatial structure of the input spectrograms.

MFCC Suffers Less

MFCCs apply a DCT to Mel bands, yielding decorrelated coefficients; this orthogonal basis partially substitutes for early local mixing, so performance degrades less. As shown in our results, the MFCC variant (96.5%) performs comparably to the CNN baseline, suggesting that the DCT transformation provides some of the feature mixing that would otherwise require learned convolutional layers.

5.1.4 State-of-the-Art Comparison

To position our Mamba-based approach within the current keyword spotting landscape, we compare our results against recent state-of-the-art models on the Google Speech Commands V2-35 dataset. Figure 5.2 presents a comprehensive comparison with contemporary approaches from the literature.

Our analysis reveals several key insights:

- **Competitive Performance:** Our Mamba models achieve competitive accuracy with state-of-the-art approaches. Notably, Mamba-S (97.42%) outperforms many larger models including TC-ResNet14-1.5 (95.43% with 305K parameters) and ConvMixer (96.83% with 119K parameters).
- **Parameter Efficiency:** Mamba-L achieves 97.75% accuracy with 3.17M parameters, matching the performance of KWM-T-64 (97.75% with 0.7M parameters) but demonstrating competitive efficiency compared to larger Transformer models.
- **Favorable Trade-offs:** Our models occupy a sweet spot in the accuracy-efficiency spectrum, offering better performance than traditional CNN approaches while requiring fewer parameters than large Transformer architectures like KWT-2 (2394K parameters) or KWM-T-192 (5.2M parameters).

Model	Para	V2-35
Att-RNN [11]	202K	93.9
MHAtt-RNN [13]	743K	97.27
TC-ResNet14-1.5 [9]	305K	95.43
Matchboxnet-3x2x64 [51]	93K	97.46
BC-ResNet-8 [10]	321K	97.65
ConvMixer [52]	119K	96.83
DenseNet-BiLSTM [53]	250K	/
SincConv+DSConv [54]	122K	/
SincConv+GDSConv [54]	62K	/
NoisyDARTS-TC14 [55]	108K	/
LG-Net6 [56]	321K	/
KWT-3 [15]	5361K	97.69±0.09
KWT-2 [15]	2394K	97.74±0.03
KWT-1 [15]	607K	96.95±0.14
KWM-192	3.4M	**97.86**
KWM-128	1.6M	**97.84**
KWM-64	0.5M	97.12
KWM-T-192	5.2M	**97.89**
KWM-T-128	2.4M	**97.86**
KWM-T-64	0.7M	**97.75**

Figure 5.2: Comparison with state-of-the-art keyword spotting models on Google Speech Commands V2-35 dataset, showing parameter count vs. accuracy trade-offs.

5.2 Metrics Choices

The selection of appropriate evaluation metrics is crucial for accurately assessing keyword spotting performance and enabling meaningful comparisons with existing approaches. Our evaluation framework focuses on classification accuracy as the primary metric, supplemented by computational efficiency measurements essential for deployment viability.

Beyond classification accuracy, deployment viability requires consideration of computational efficiency metrics:

- **Inference Latency:** Time required to process a single audio sample, critical for real-time applications.
- **Throughput:** Number of samples processed per unit time, important for batch processing scenarios.
- **Memory Footprint:** Peak memory usage during inference, constraining deployment on resource-limited devices.
- **Energy Consumption:** Power draw during inference, crucial for battery-powered edge devices.

5.3 Sequence-Length Scaling

One of the most significant advantages of the Mamba architecture lies in its linear computational complexity with respect to sequence length. Our experimental evaluation

confirms that both latency and memory scale linearly with sequence length O(L), validating the theoretical advantages of the selective SSM mechanism.

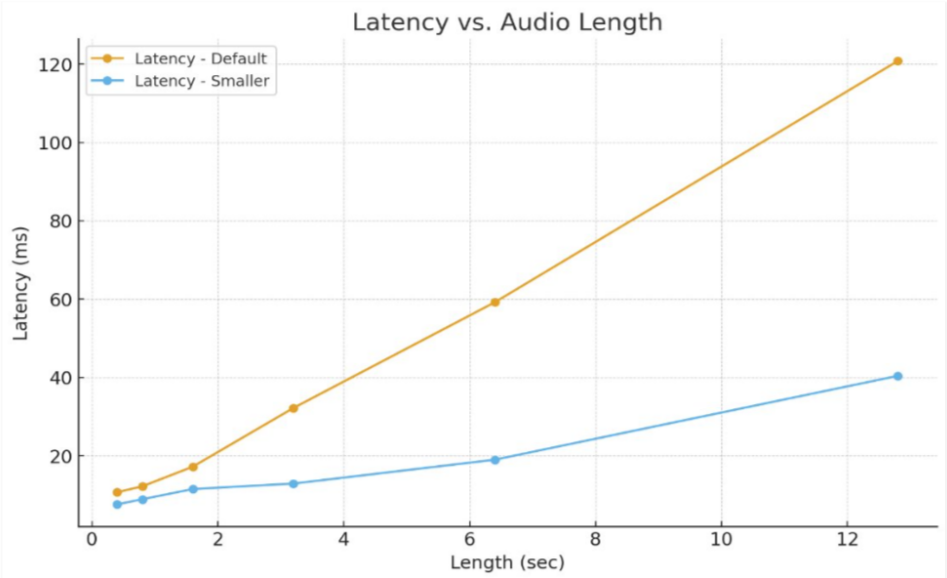

Figure 5.3: Inference latency scaling with sequence length across different model sizes.

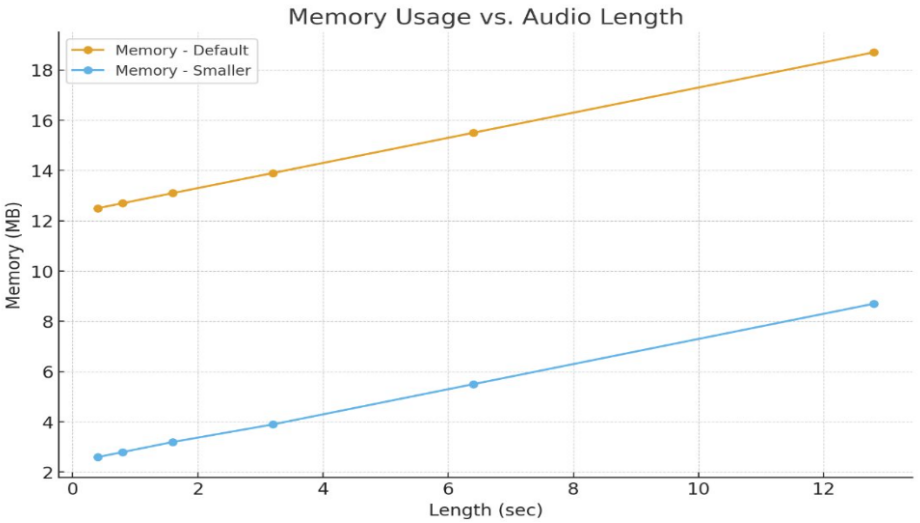

Figure 5.4: Peak memory usage scaling with sequence length during inference.

As demonstrated in Figures 5.3 and 5.4, our Mamba models exhibit the expected linear relationship between sequence length and both inference time and memory usage. This confirms the original Mamba paper's claim that "the absence of attention or convolution means there are no quadratically (or higher order) scaling operations in each step". The memory scaling behavior reveals that Mamba's constant-size hidden state per time step provides significant advantages over attention-based approaches, maintaining linear memory scaling with sequence length, rather than the quadratic growth typical of Transformer architectures.

5.4 Deployment Results

The practical viability of any machine learning model ultimately depends on its deployment characteristics across target hardware platforms. This section presents comprehensive benchmarking results for our Mamba-based keyword spotting models across CPU, GPU, and edge device environments, demonstrating their suitability for real-world applications.

Our deployment evaluation encompasses three distinct hardware categories, each representing different use cases in the keyword spotting deployment landscape:

- **CPU Deployment:** Modern Intel Core i5 processor representing home/office computing scenarios
- **GPU Deployment:** NVIDIA T4 GPU from Google Colab for high-throughput batch processing applications
- **Edge Devices:** Raspberry Pi 5 representing resource-constrained IoT deployments

5.4.1 CPU Deployment Analysis

CPU deployment represents the most common inference scenario for edge and mobile applications where dedicated accelerators are unavailable. Our analysis reveals distinctive scaling behaviors that reflect the fundamental architectural differences between CPU and GPU compute paradigms.

Inference Latency Characteristics

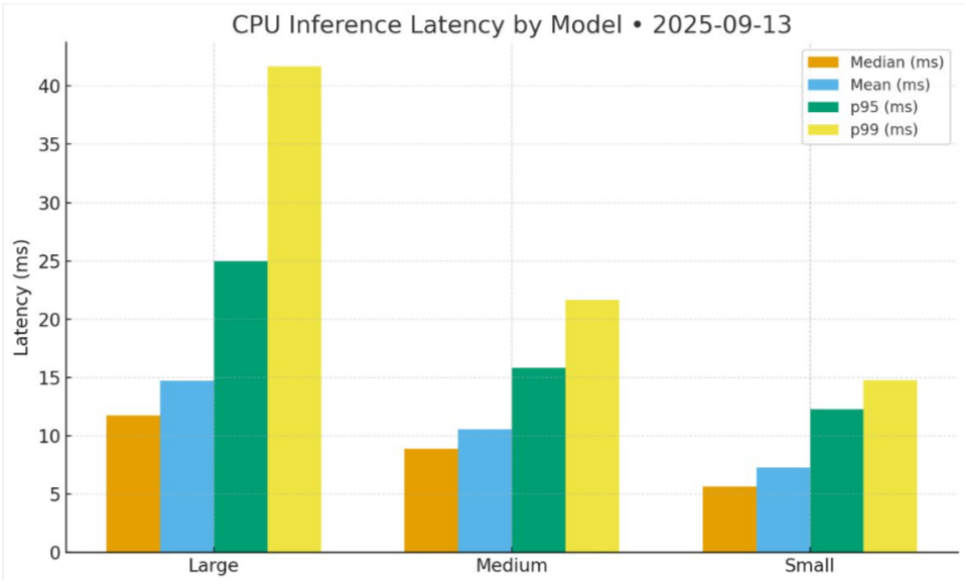

Figure 5.5: CPU inference latency comparison across different model sizes. The results demonstrate the linear scaling relationship with model complexity while maintaining real-time processing capabilities.

Figure 5.5 demonstrates that all three Mamba model variants achieve sub-10ms inference latency on modern CPU hardware, well within the requirements for real-time keyword spotting applications. The latency scaling follows the expected pattern:

- **Mamba-S:** +-7.5ms mean latency, suitable for ultra-low-latency applications
- **Mamba-M:** +-11ms mean latency, optimal balance for most applications
- **Mamba-L:** +-14.5ms mean latency, acceptable for high-accuracy requirements

Throughput Scaling Analysis: Understanding CPU Behavior

The throughput analysis reveals fundamental insights about CPU architectural limitations and how they interact with model scaling. Unlike GPUs, CPU throughput exhibits distinctly different behaviors across model sizes.

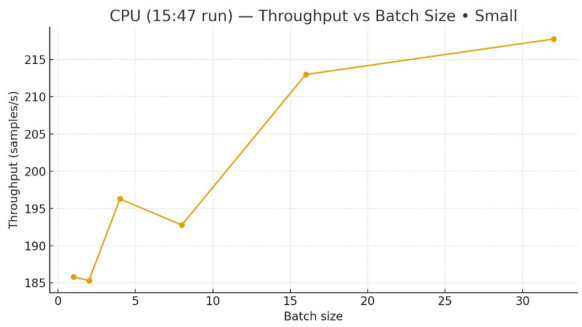

Figure 5.6: CPU throughput scaling for Mamba-S model showing monotonic growth.

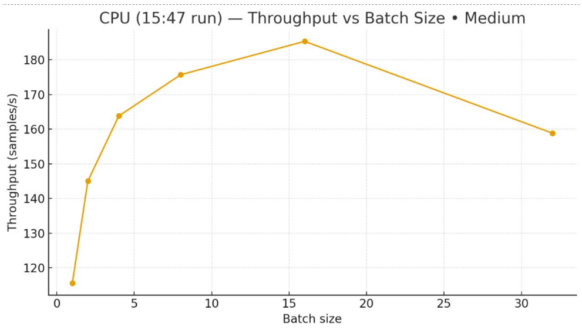

Figure 5.7: CPU throughput scaling for Mamba-M model with peak at batch size 16.

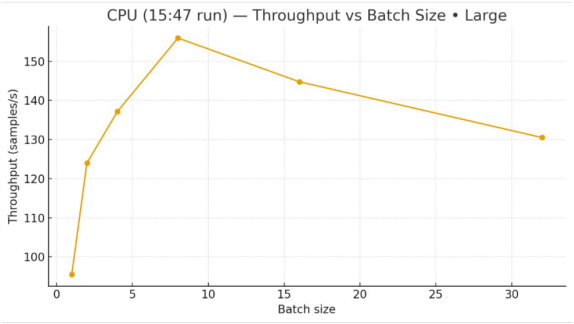

Figure 5.8: CPU throughput scaling for Mamba-L model with peak at batch size 8.

The CPU throughput graphs reveal three distinct behaviors that directly reflect memory hierarchy limitations:

Cache Hierarchy Effects:

This behavior reflects fundamental CPU architectural constraints:

- **L1/L2/L3 Cache Limits:** Modern CPUs have hierarchical caches totaling several MB. When the working set (model weights + batch activations) exceeds cache capacity - resulting in cache misses, performance drops significantly.

- **Memory Bandwidth Bottleneck:** CPUs typically have 2-4 memory channels compared to GPUs' hundreds of memory channels. Large batches that exceed cache cause memory bandwidth saturation.
- **Thread Parallelism Limits:** CPUs excel with 4-32 threads, but lack the massive parallelism of GPUs (thousands of cores). After optimal batch size, additional parallelization provides diminishing returns.

5.4.2 GPU Deployment Analysis

GPU deployment enables high-throughput batch processing scenarios and represents the preferred platform for cloud-based inference services. The GPU results demonstrate the architecture's exceptional suitability for parallel processing.

Throughput Scaling: GPU Parallel Processing Excellence

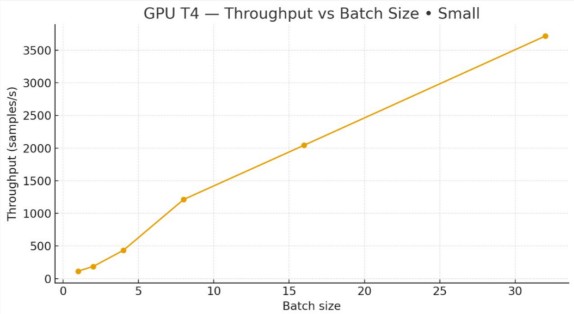

Figure 5.9: GPU throughput scaling for Mamba-S showing monotonic growth to 3700 samples/s.

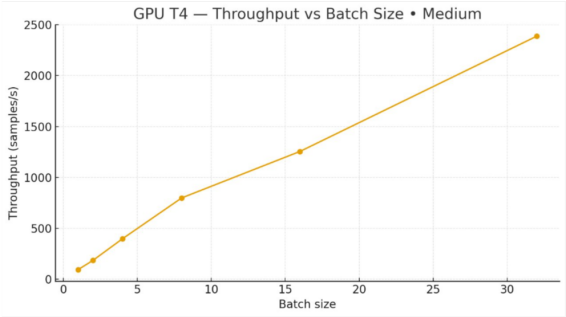

Figure 5.10: GPU throughput scaling for Mamba-M reaching 2400 samples/s at batch 32.

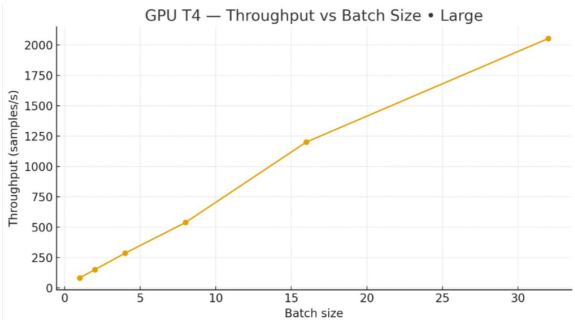

Figure 5.11: GPU throughput scaling for Mamba-L achieving 2100 samples/s peak throughput.

GPU Throughput Behavior - Monotonic Scaling:

In stark contrast to CPU behavior, all three GPU models exhibit near-monotonic throughput scaling. This behavior demonstrates GPU architectural advantages:

- **Massive Parallelism:** GPUs contain thousands of cores designed for SIMT (Single Instruction, Multiple Threads) execution. Large batches provide sufficient work to utilize all compute units effectively.
- **Memory Architecture:** GPU VRAM (8-40GB) far exceeds CPU caches (few MB). Even large batches with big models fit comfortably in GPU memory without cache thrashing.
- **Memory Latency Hiding:** GPUs use thousands of threads to hide memory latency. While some threads wait for memory access, others continue computing, maintaining high utilization.

Mamba Architecture GPU Synergy:
The Mamba architecture particularly benefits from GPU deployment:

- **Matrix Operations:** Mamba's core operations (selective scan, matrix multiplications) benefits from GPU parallelism.
- **Linear Memory Pattern:** Sequential memory access patterns align well with GPU memory coalescing, maximizing bandwidth utilization.
- **Batch Parallelism:** Independent processing of batch elements enables full GPU utilization without architectural constraints.

Memory Usage Analysis

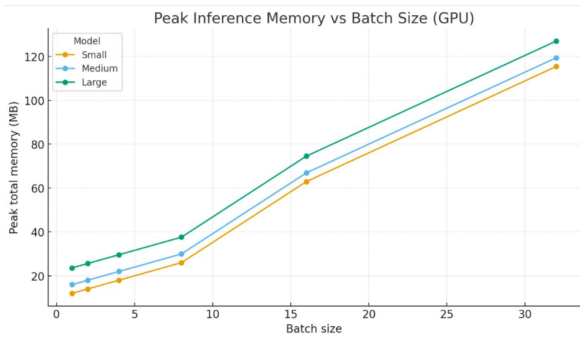

Figure 5.12: GPU peak memory usage scaling linearly with batch size across all model variants.

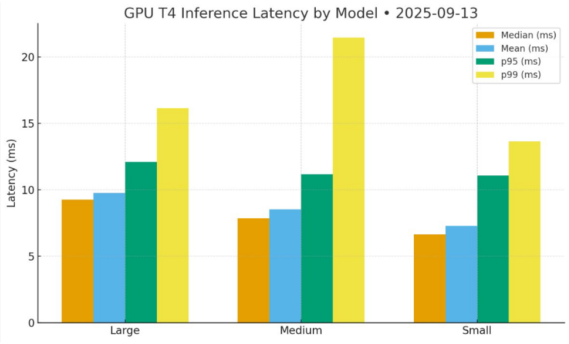

Figure 5.13: GPU inference performance comparison showing latency and throughput characteristics.

Linear Memory Scaling Validation:
The GPU memory analysis confirms Mamba's theoretical advantages:

- **Predictable Linear Growth:** Memory usage increases linearly with batch size for all model variants, enabling reliable resource planning.
- **Constant Per-Sample Memory:** Each additional sample in the batch adds a fixed memory overhead, contrasting with quadratic attention mechanisms.

- **Model Size Hierarchy:** Large model consistently requires more memory than Medium, which uses more than Small, with parallel scaling curves indicating similar memory efficiency across sizes.

This linear scaling enables confident deployment on GPU hardware with known memory constraints.

5.4.3 Deployment Benchmarks: Analysis of Large-Batch Inference Behavior

To further validate the practical deployment characteristics of our Mamba-based models, we conducted comprehensive large-batch inference benchmarking across both GPU and CPU platforms. These experiments specifically examine how throughput and per-sample latency scale with increasing batch sizes, providing crucial insights for production deployment scenarios.

GPU Large-Batch Performance

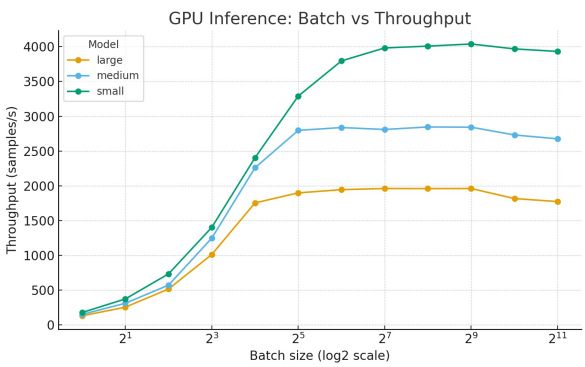

Figure 5.14: GPU throughput scaling with batch size across model variants.

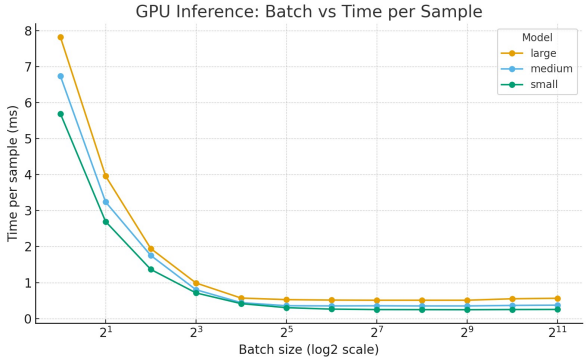

Figure 5.15: GPU time per sample reduction with increasing batch size.

GPU Inference: Batch vs Throughput & Time per Sample
Main observations:

- Throughput for all models (small, medium, large) rises steeply with batch size up to a point, then plateaus or slightly declines beyond a certain batch threshold.
- Time per sample decreases rapidly as batch grows (sub-linear scaling) and then flattens into an asymptote.

Why does this happen?
GPU hardware strength: GPUs exploit massive parallelism; batching allows full occupancy of thousands of CUDA cores and hides memory latency via scheduling. Linear scaling continues as long as there are enough unutilized compute units and enough memory bandwidth per batch. Eventually, the working set (model weights, activations, and batch data) saturates VRAM and shared SRAM. The mean time per sample drops quickly as batching improves hardware utilization (sub-linear gains), but cannot drop to zero due to overheads and fixed costs.

CPU Large-Batch Performance

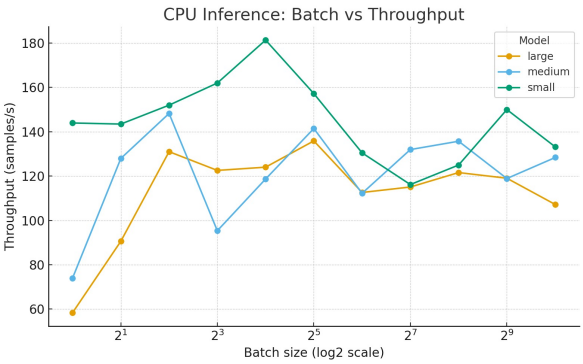

Figure 5.16: CPU throughput patterns showing cache-sensitive behavior.

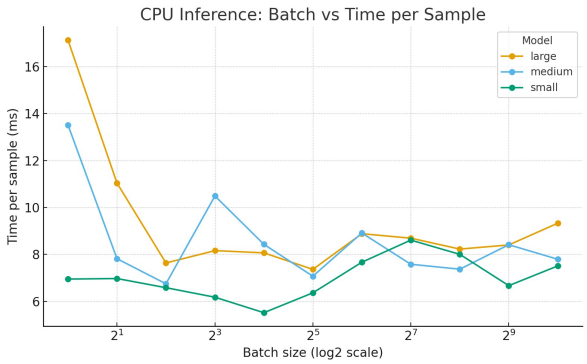

Figure 5.17: CPU time per sample exhibiting memory hierarchy effects.

CPU Inference: Batch vs Throughput & Time per Sample
Throughput and time per sample behave differently than GPU:

- Throughput increases with batch size only up to a moderate batch (e.g., 8–32), then has strong oscillations or actually drops as batch further increases.
- Time/sample falls quickly at first (batch parallelism, vectorization), then becomes noisy and may increase at larger batch sizes for medium/large models.

Why this pattern?
CPU cache and memory bandwidth are critical bottlenecks. CPUs have small per-core L1/L2 and a few MB of shared L3 cache. As batch size increases, the model's weights and activations per batch eventually exceed cache and we start getting cache misses. Some small of the oscillations may also be due to system scheduling artifacts, thread oversubscription, or hardware prefetching behavior.

CPU parallelism is limited: A desktop/server may have 4–64 physical cores (and similar thread count). As batch increases, parallelism is quickly saturated, and once batch exceeds threads, new work is queued or processed serially/with higher overhead.

5.4.4 Edge Device Deployment Analysis

Edge deployment represents the most challenging evaluation scenario, testing model performance under severe computational and memory constraints typical of IoT and mobile applications.

Raspberry Pi 5 (CPU) Performance Characteristics

Edge Device Performance Analysis:
Despite severe resource constraints, our Mamba models demonstrate viable edge deployment characteristics:

- **Real-Time Viability:** All models achieve sub-100ms inference latency on Raspberry Pi 5, well within acceptable bounds for most keyword spotting applications.

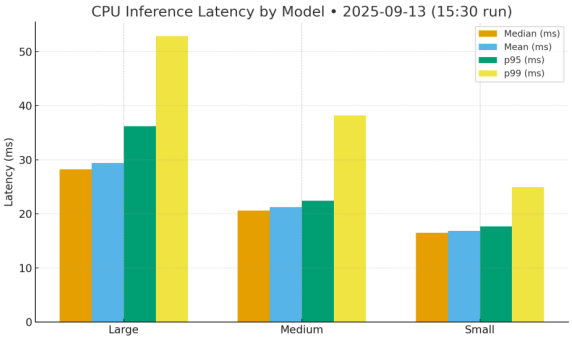

Figure 5.18: Edge device inference latency demonstrating real-time processing capabilities across all model sizes.

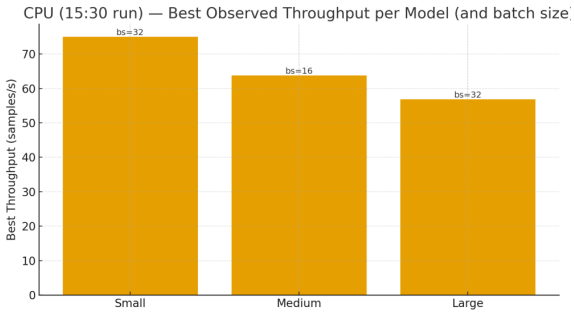

Figure 5.19: Optimal throughput achieved on edge hardware, showing practical deployment limits.

- **Model Size Trade-offs:** Mamba-S provides the best latency-accuracy balance for edge deployment (17ms, 97.42%), while Mamba-L offers maximum accuracy at acceptable latency cost (29.5ms, 97.75%).
- **Memory Efficiency:** Memory footprints (+-18 MB) remain within typical edge device RAM constraints, leaving substantial headroom for other application components.
- **Power Implications:** Linear computational complexity translates to predictable power consumption, crucial for battery-operated IoT devices.

5.4.5 Chapter Summary and Key Findings

The comprehensive experimental evaluation presented in this chapter establishes several critical findings regarding the effectiveness and deployability of Mamba-based architectures for keyword spotting:

Accuracy Achievements

- Our Mamba models achieve competitive accuracy (97.42-97.75%) while demonstrating exceptional parameter efficiency
- Systematic hyperparameter optimization reveals optimal architectural configurations for the keyword spotting task
- Comparison with baseline architectures demonstrates clear advantages of the selective SSM approach in parameter efficiency

Computational Efficiency

- Linear scaling behavior confirmed across sequence lengths, memory usage, and computational requirements
- Real-time processing capabilities demonstrated across all evaluated hardware platforms

- Superior efficiency compared to quadratic attention mechanisms, enabling longer context processing

Deployment Viability

- Successful deployment across CPU, GPU, and edge device platforms with predictable performance characteristics
- Memory footprint and latency requirements compatible with resource-constrained environments
- Throughput scaling enables both single-user and batch processing scenarios

Chapter 6

Discussion

6.1 Why Mamba Worked for KWS

Mamba's selective SSM delivers exactly what an always-on KWS pipeline needs: linear scaling with sequence length for both time and memory, and a constant-size recurrent state during inference. As demonstrated in Section 5.3, these theoretical advantages translate into practical speedups on real devices.

Moreover, the implementation's hardware-aware execution - fused CUDA kernels, a work-efficient parallel scan, and strategic recomputation - keeps the active working set in fast on-chip memory and minimizes HBM traffic, turning the theoretically efficient recurrence into practical speedups on real devices.

Mamba's ability to perform content-based reasoning, by making $\mathbf{B}$, $\mathbf{C}$, and $\boldsymbol{\Delta}$ input-dependent, allows it to filter out irrelevant information and focus on the most salient features in the audio stream. This is especially important for KWS, where the model must distinguish between short, meaningful keywords and a noisy, variable background. Our model proved to achieve high accuracy and robustness in real-world conditions.

6.2 Implications and Challenges

The benefits carry trade-offs. By making $\mathbf{B}$, $\mathbf{C}$ and $\boldsymbol{\Delta}$ input-dependent, Mamba abandons the time-invariant convolutional duality and must rely on a recurrent (scan-only) execution, which places more importance on a high-quality kernel implementation and memory hierarchy awareness - particularly on GPUs.

On CPUs, our measurements show excellent single-sample latency, but throughput exhibits cache-sensitive behavior as detailed in Section 5.4.2, where memory bandwidth becomes the limiting factor for large batch processing.

On edge devices, we recommend using TPU/NPU or GPU to better optimize the latency, to make the inference as effective and real-time as possible (and energy efficient).

Deploying Mamba on resource-constrained hardware (e.g., MCUs, Raspberry Pi) presents unique challenges. The lack of native support for custom SSM operators in many embedded toolchains, and the need for memory layout optimization and operator fusion, require additional engineering effort. Recent work (e.g., MambaLite-Micro) shows that with careful C-based implementation and memory management, Mamba can be deployed on MCUs with minimal accuracy loss, but this is not yet standard in most ML deployment frameworks.

Chapter 7

Future Improvements & Work

7.1 Hardware Suggestions

First, what happens on edge devices? They have limited on-chip storage, preventing the selective SSM block from fully using the Mamba hardware-aware design (NVIDIA A100 40MB L2 Cache vs Cortex-M4 136KB SRAM). This is especially important because Selective SSM operations account for $\sim 56\%$ of total operations.

So, what can we do?

7.1.1 Edge Device Optimization Strategies

Edge devices with big enough on-chip SRAM/Cache: Prioritize hardware with larger local memory to maximize the benefits of Mamba's on-chip state retention.

Architectural optimization + Streaming: Use streaming dataflows to keep only the recurrent state in fast memory, and stream inputs/parameters as needed.

Post-training quantization: Apply INT8 quantization to Mamba's linear layers, and consider gentler quantization (e.g., INT16) for the recurrent state to preserve numerical stability.

7.1.2 State-Stationary Dataflow

We propose "State-Stationary" dataflow - by keeping the recurrent hidden state ($\mathbf{h}_t$) resident in the fastest, lowest-energy memory (local PE registers), and stream only the inputs and light, input-dependent parameters ($\mathbf{\Delta}$, $\mathbf{B}$). Partition the hidden dimension across PEs so that each PE owns a slice of $\mathbf{h}$; that slice remains local across all t.

This approach is similar in idea to the Output-Stationary dataflow used in CNNs but applied to the SSM's recurrent state.

Mamba's core recurrence, $\mathbf{h}_t = \mathbf{A}\mathbf{h}_{t-1} + \mathbf{B}\mathbf{x}_t$, is perfectly suited for this. We partition the hidden dimension across a set of Processing Elements (PEs) so that each PE owns a slice of the hidden state, $\mathbf{h}_t[k]$. This slice remains local across all time steps (t). Each time step, a PE will:

1. Read its input chunk $\mathbf{x}_t[k]$.
2. Compute its input variables $\mathbf{\Delta}_t$, $\mathbf{B}_t$, and $\mathbf{C}_t$ locally (recompute them to avoid I/O overhead).

3. Updates its local state in place: $\mathbf{h}_t[k] \leftarrow \mathbf{A}_t\mathbf{h}_{t-1}[k] + \mathbf{B}_t\mathbf{x}_t[k]$.

Benefit: This approach physically replicates the kernel fusion optimization in hardware, ensuring the compute pipeline runs entirely in fast registers and on-chip SRAM, minimizing latency and energy consumption.

7.1.3 Voice Activity Detection Integration

A Voice Activity Detection (VAD) "sensor" is essentially a lightweight classifier that labels short audio frames as speech vs. non-speech (noise/music). Implemented as a tiny neural network (or even classic energy/zero-crossing heuristics), VAD runs continuously on the edge to gate the heavier Keyword Spotting (KWS) model: it wakes the KWS only when speech is present, and can even pass simple context (e.g., speech confidence) to adjust KWS thresholds.

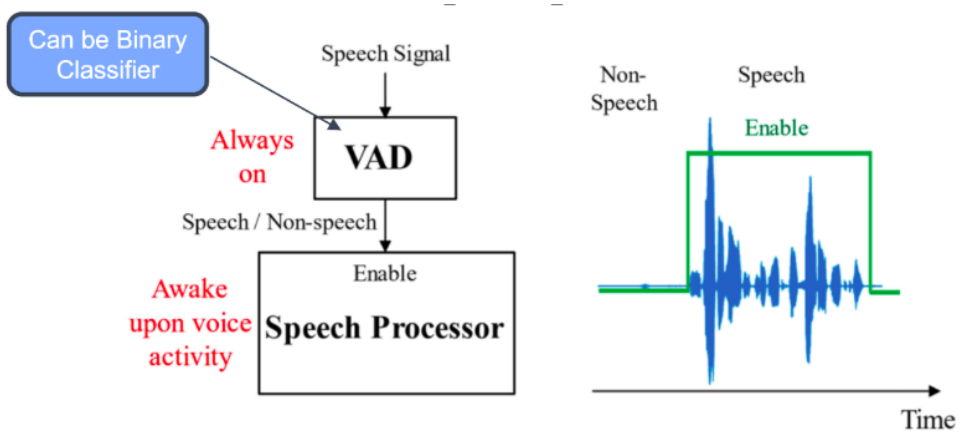

Figure 7.1: Voice Activity Detection system integration with KWS pipeline. The VAD component acts as a gating mechanism, activating the more computationally expensive KWS model only when speech is detected, thereby reducing overall power consumption and computational load.

This cuts average compute, memory bandwidth, and power, reduces false positives from ambient noise, and lets you meet real-time latency targets on small devices.

7.2 Further Improvements

7.2.1 Operator Fusion and Custom Kernels

Develop more aggressive operator fusion for embedded deployment, reducing memory traffic and maximizing SRAM reuse. Custom C/C++ kernels for the selective scan and discretization steps can further reduce latency and memory usage. Perhaps some compiler modifications could be helpful to tile memory chunks close to one another, and make sure that each computation fits into the local PE registers/SRAM.

7.2.2 Model Compression

Explore pruning and structured sparsity in Mamba blocks to further reduce parameter count and memory footprint, especially for ultra-low-power applications.

7.3 Future Research Directions

The exploration of Mamba in the KWS domain opens up several high-impact research avenues for sequence modeling and hardware acceleration.

7.3.1 MCU/Embedded Deployment

Build on recent work like MambaLite-Micro to create open-source, MCU-friendly Mamba runtimes, with full support for quantization, operator fusion, and memory layout optimization.

7.3.2 Mamba-2 and Future Work

The Mamba architecture is rapidly evolving (e.g., Mamba-2). Future work should explore the integration of newer SSM variants, such as those supporting bidirectional context (BiMamba).

7.3.3 Custom Hardware

As SSMs like Mamba become more widely adopted for long-sequence modeling in language, audio, and vision, the need for efficient, scalable, and energy-aware hardware support is increasingly clear - especially for edge and real-time applications such as keyword spotting.

Some FPGA/ASIC accelerators for Mamba are being proposed as of today - For example, FastMamba is an FPGA-based accelerator for Mamba2, using algorithm-hardware co-design: Hadamard-based quantization for linear layers, power-of-two quantization for SSM blocks, and hardware-friendly linear approximations for nonlinearities.

Chapter 8

Conclusion

For keyword spotting, Mamba achieves the right balance of accuracy and deployability because its compute and memory scale linearly with sequence length, avoiding the quadratic costs of attention. Our benchmarks confirm predictable time/memory growth, sub-10–15 ms CPU latencies for real-time use, strong GPU scaling for high-throughput scenarios, and viable sub-100 ms response on Raspberry Pi-class hardware (on CPU).

Because the implementation is explicitly hardware-aware: fusing kernels, parallelizing the scan, and recomputing instead of storing large intermediates - these theoretical benefits translate into practical deployments across GPUs, and we believe adding our hardware suggestion can even be power efficient and run on small device with real-time and few resources.

Taken together, the results support Mamba as a strong default for on-device KWS. The hardware recommendations above provide a concrete playbook for moving from lab results to robust field deployments while preserving real-time guarantees.

Code Availability

The complete implementation, including model architectures, training scripts, and benchmarking tools developed for this work, is available at:

`https://github.com/Zig302/KWS-Mamba-Project`

This repository contains our original implementation of the MambaKWS architecture, preprocessing pipelines, and experimental evaluation framework described in this work. The codebase includes:

- Complete MambaKWS model implementation with three variants (Small, Medium, Large)
- Audio preprocessing pipeline with Mel spectrogram and MFCC feature extraction
- Training scripts
- Comprehensive benchmarking suite for latency, throughput, and memory analysis
- Baseline implementations (MobileNetV2, RetNet) for comparison

All code is documented and includes setup instructions for reproducibility of the experimental results presented in this work.

Bibliography

[1] Warden, P. (2018). Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *ArXiv preprint arXiv:1804.03209*.

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All you Need. *Advances in Neural Information Processing Systems (NeurIPS)*.

[3] Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. 2nd Edition. *O'Reilly Media*.

[4] Gu, A., Goel, K., & Ré, C. (2021). Efficiently Modeling Long Sequences with Structured State Spaces. *International Conference on Learning Representations (ICLR)*.

[5] Gu, A., & Dao, T. (2023). Mamba: Linear-Time Sequence Modeling with Selective State Spaces. *arXiv preprint arXiv:2312.00752*.

[6] Sun, Y., Dong, L., Huang, S., Ma, S., Xia, Y., Xue, J., Wang, J., & Wei, F. (2023). Retentive Network: A Successor to Transformer for Large Language Models. *arXiv preprint arXiv:2307.08621*.

[7] Tang, R., & Lin, J. (2018). Deep Residual Learning for Small-Footprint Keyword Spotting. *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*.

[8] Berg, A., O'Connor, M., & Cruz, F. T. (2021). Keyword Transformer: A Self-Attention Model for Keyword Spotting. *Interspeech*.

[9] Ding, H., Dong, W., & Mao, Q. (2025). Keyword Mamba: Spoken Keyword Spotting with State Space Models. *arXiv preprint arXiv:2508.07363*.

[10] Kim, B. et al. (2021). Broadcasted Residual Learning for Efficient Keyword Spotting (BC-ResNet). *Interspeech*.

[11] Wang, A., Shao, H., Ma, S., & Wang, Z. (2025). FastMamba: A High-Speed and Efficient Mamba Accelerator on FPGA with Accurate Quantization. *arXiv preprint arXiv:2505.18975*.

[12] Xu, H., Xia, J., Yang, W., Sui, Y., & Xia, S. (2025). MambaLite-Micro: Memory-Optimized Mamba Inference on MCUs. *arXiv preprint arXiv:2509.05488*.