Analyse von TinyML-Inferenz auf dem ESP32 im Vergleich zu High-Performance-Mikrocontrollern

Studienarbeit

des Studiengangs **Elektrotechnik - Automation**

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Samuel Geffert

02. Januar 2026

Bearbeitungszeitraum:	12 Wochen
Matrikelnummer, Kurs:	4203964, TEL23AT1
Dualer Partner:	KHS GmbH, Worms
Betreuer:	Prof Dr. Michael Arzberger

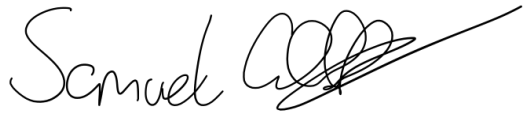
Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema:

Analyse von TinyML-Inferenz auf dem ESP32 im Vergleich zu High-Performance-Mikrocontrollern

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Dabei wurde unter anderem Generative AI (Google Gemini) als Hilfsmittel zur sprachlichen Überarbeitung und zur Einholung von Formulierungsvorschlägen genutzt. Die inhaltiliche Erarbeitung sowie die finale Auswahl und Prüfung der Texte erfolgeten eigenständig.

Mannheim, den 02. Januar 2026



Samuel Geffert

Zusammenfassung

Diese Studienarbeit evaluiert die technische Eignung des kostengünstigen ESP32 für TinyML-Anwendungen im direkten Vergleich zu leistungsstarken ARM Cortex-M7 Systemen. Um die Leistungsfähigkeit objektiv zu bewerten, wurde ein Benchmarking auf Basis des Industriestandards MLPerf Tiny (Closed Division) durchgeführt. Dabei wurden der generische ESP32, der ESP32-S3 sowie die Referenzboards Teensy 4.0 und Arduino Giga R1 unter Nutzung von TensorFlow Lite Modellen für Keyword Spotting, Image Classification und Visual Wake Words untersucht.

Die Messergebnisse belegen eine deutliche Dominanz der Cortex-M7 Architektur. Diese weist auch die höchste Energieeffizienz pro Inferenz auf, da die extrem kurze Rechenzeit den höheren momentanen Leistungsbedarf überkompensiert. Während der moderne ESP32-S3 den Leistungsrückstand durch Vektor-Instruktionen verringern konnte, stieß der generische ESP32 bei speicherintensiven Vision-Modellen an physikalische Grenzen (Speicherlimitierung), was zu einem Ausfall der Modellausführung führte.

Die Arbeit kommt zu dem Schluss, dass der generische ESP32 für komplexe Bildverarbeitungsaufgaben ungeeignet ist. Für einfachere Audio-Anwendungen zeigt sich jedoch, dass der Mikrocontroller trotz geringerer Effizienz eine funktional ausreichende Leistung zu minimalen Hardwarekosten bietet, was ihn für preissensitive Massenanwendungen qualifiziert.

Abstract

This study evaluates the technical suitability of the low-cost ESP32 for TinyML applications in direct comparison to high-performance ARM Cortex-M7 systems. To objectively assess performance, a benchmark based on the MLPerf Tiny industry standard (Closed Division) was conducted. The generic ESP32, ESP32-S3, as well as the reference boards Teensy 4.0 and Arduino Giga R1, were examined using TensorFlow Lite models for Keyword Spotting, Image Classification, and Visual Wake Words.

The measurement results demonstrate a clear dominance of the Cortex-M7 architecture. Surprisingly, this architecture also exhibits the highest energy efficiency per inference, as the extremely short computation time overcompensates for the higher instantaneous power consumption. While the modern ESP32-S3 was able to narrow the performance gap through vector instructions, the generic ESP32 reached physical limits (memory limitations) with memory-intensive vision models, resulting in a failure of model execution.

The study concludes that the generic ESP32 is unsuitable for complex image processing tasks. However, for simpler audio applications, the results show that despite lower efficiency, the microcontroller offers functionally sufficient performance at minimal hardware costs, qualifying it for price-sensitive mass-market applications.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CMSIS-NN	Common Microcontroller Software Interface Standard – Neural Network.
CPU	Central Processing Unit.
DNN	Deep Neural Network.
DRAM	Data RAM.
DS-CNN	Depthwise Seperable Convolutional Neural Network.
DSP	Digital Signal Processing.
DUT	Device Under Test.
FPU	Floating Point Unit.
GPIO	General Purpose Input/Output.
IC	Image Classification.
IoT	Internet of Things.
IPS	Inferenzen pro Sekunde.
IRAM	Instruction RAM.
JS220	Joulescope 220.
KI	Künstliche Intelligenz.
KWS	Keyword Spotting.

MAC	Multiply-Accumulate.
MCU	Microcontroller Unit.
ML	Machine Learning.
MLPerf	Machine Learning Performance.
NAS	Neural Architecture Search.
NN	Neuronales Netz.
PC	Personal Computer.
RAM	Random-Access Memory.
ResNet	Residual Network.
RTOS	Real-Time Operating System.
SIMD	Single Instruction, Multiple Data.
SMLAD	Signed Multiply Accumulate Dual.
SRAM	Static Random-Access Memory.
TCM	Tightly Coupled Memory.
TFLite	TensorFlow-Lite.
TFLM	TensorFlow Lite for Microcontrollers.
TinyML	Tiny Machine Learning.
UART	Universal Asynchronous Receiver/Transmitter.
VS Code	Visual Studio Code.
VWW	Visual Wake Words.

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In den letzten Jahren hat das Thema Tiny Machine Learning (TinyML) große Aufmerksamkeit von Industrie und Wissenschaft erhalten, getrieben von dem Ziel, neue Möglichkeiten für nachhaltige Entwicklungstechnologien zu erschließen [1]. Die zunehmende Verbreitung von Machine Learning (ML)-Technologien hat zu einem deutlichen Anstieg des Energieverbrauchs geführt. – von kleinen Internet of Things (IoT)-Geräten bis hin zu großen Rechenzentren [2]. Vor diesem Hintergrund etabliert sich TinyML als Schlüsseltechnologie: Das Thema wurde erstmals im Jahr 2019 in Forschungspublikationen diskutiert [3]. TinyML wird dabei oft definiert als die Fähigkeit, ML-Modelle bei einer Durchschnittsleistung von unter einem Milliwatt zu betreiben, sodass diese theoretisch über ein Jahr lang mit nur einer Knopfzelle laufen könnten [4].

Die Technologie kommt bereits heute in vielen Bereichen wie im Gesundheitswesen, in der Landwirtschaft oder in der Industrie zum Einsatz [3]. Ein Haupttreiber ist dabei das Internet der Dinge (IoT). Experten erwarten deshalb, dass der Markt für Mikrocontroller in den kommenden Jahren stark wachsen wird [1].

Die große Menge an vernetzten Geräten erzeugt Datenmengen, die eine zentrale Verarbeitung in der Cloud ineffizient machen. Daten werden am Ende des Netzwerks (engl. Edge) produziert, weshalb es effizienter ist, diese Daten auch dort zu verarbeiten [5]. Das klassische Cloud-Computing stößt hier an seine Grenzen: Shi et al. identifizieren die große Datenmenge, die benötigte Bandbreite, Latenzzeiten sowie Echtzeitanforderungen und Datenschutz als Probleme [5]. Auch Greco et al. betonen, dass Cloud-Computing bzw. Edge-AI für Echtzeitanwendungen und Geräte mit begrenzter Rechenleistung und Bandbreite nicht geeignet sind [6]. Hier setzt TinyML an, indem es die Inferenz – also die Ausführung trainierter Modelle – direkt auf das Endgerät verlagert. Dies schafft eine hohe Reaktionsfähigkeit und gewährleistet den Datenschutz, da sensible Daten das Gerät nicht verlassen müssen. Gleichzeitig bleiben die Energiekosten gering, insbesondere im Vergleich zur Übertragung der Daten in die Cloud [4].

1.1 Problemstellung und Forschungsfrage

Trotz der Vorteile bringt TinyML Herausforderungen mit sich, die vor allem aus den hohen Hardware-Anforderungen moderner künstlicher Intelligenz (KI)-Algorithmen resultieren. Während Cloud-Server über nahezu unbegrenzte Ressourcen verfügen, müssen TinyML-Modelle auf Mikrocontrollern mit nur wenigen Kilobyte Static Random-Access Memory (SRAM) auskommen. Dies führt zu einem direkten Konflikt zwischen der Komplexität eines Modells und der verfügbaren Hardware. Banbury et al. identifizieren in diesem Zusammenhang neben dem begrenzten Speicher außerdem die große Vielfalt unterschiedlicher Hardware-Plattformen als zentrale Hürde [4].

Der begrenzte Speicher limitiert häufig die Modellgenauigkeit sowie die Zuverlässigkeit der Systeme [7]. Hinzu kommt ein ökonomischer Faktor: Bei großen IoT-Deployments, die oft tausende von Sensoren erfassen, sind die Kosten pro Einheit ein limitierender Faktor [8]. Entwickler stehen daher oft vor dem Dilemma, eine Balance zwischen ausreichender Rechenleistung für KI-Anwendungen und minimalen Hardwarekosten zu finden.

Vor diesem Hintergrund rückt der ESP32 der Firma Espressif als sehr populärer und kostengünstiger Mikrocontroller in den Fokus. Die Forschungsfrage dieser Arbeit beschäftigt sich daher mit der Konkurrenzfähigkeit dieser Plattform:

> *Inwiefern kann der ESP32 als kostengünstige Standard-Hardware mit High-Performance-Mikrocontrollern im TinyML-Umfeld mithalten und wo liegen die Leistungsgrenzen im Vergleich?*

Diese Arbeit analysiert somit nicht nur die technische Realisierbarkeit von KI-Inferenz auf dem ESP32, sondern evaluiert auch dessen Potenzial als kosteneffiziente Alternative zu leistungsfähigeren Mikrocontrollern.

1.2 Aufbau und Vorgehensweise der Arbeit

Diese Arbeit gliedert sich in sechs Kapitel. Nach der Einleitung werden in Kapitel 2 die theoretischen Grundlagen erarbeitet. Hierbei erfolgt eine Definition von TinyML, gefolgt von einer Analyse der relevanten Hardware-Architekturen (ARM Cortex-M und Xtensa) sowie des notwendigen Software-Stacks. Zudem wird die Problematik der Vergleichbarkeit erörtert und MLPerf als Industriestandard für das Benchmarking eingeführt.

Kapitel 3 beschreibt darauf aufbauend die Methodik und den Versuchsaufbau. Es erläutert die Anwendung der Benchmarking-Strategie, definiert die untersuchten Hardware-Plattformen und legt das Verfahren zur Messung von Latenz und Energieverbrauch dar.

Die quantitativen Ergebnisse dieser Untersuchungen werden in Kapitel 4 präsentiert, gegliedert nach Durchsatz, Energieeffizienz und Preis-Leistungs-Verhältnis.

Anschließend erfolgt in Kapitel 5 die Diskussion der Ergebnisse. Hier werden die Ursachen für Leistungsunterschiede analysiert, Limitationen wie Speicherengpässe beleuchtet und die ökonomische Relevanz der Daten bewertet.

Die Arbeit schließt in Kapitel 6 mit einem Fazit zur Leistungsfähigkeit des ESP32 sowie einem Ausblick auf zukünftige Optimierungspotenziale.

2 Theoretische Grundlagen

Dieses Kapitel legt das Fundament für die Untersuchung und den Vergleich der Hardware-Plattformen. Zunächst wird das Konzept von TinyML definiert und von Edge Computing abgegrenzt. Darauf aufbauend werden die Funktionsweise neuronaler Netze sowie die Optimierungstechniken – insbesondere die Quantisierung – erläutert, um diese Modelle auf Mikrocontrollern ausführbar zu machen. Im weiteren Verlauf erfolgt eine Analyse der Hardware-Architekturen (ARM Cortex-M, Xtensa) und des Software-Stacks. Abschließend widmet sich das Kapitel der Problematik des Hardware-Vergleichs und führt MLPerf als Standard für Benchmarking ein.

2.1 TinyML und algorithmische Grundlagen

Bevor die Hardware im Detail betrachtet wird, ist ein grundlegendes Verständnis des TinyML-Konzepts sowie der algorithmischen Anforderungen erforderlich. Dieser Abschnitt definiert TinyML und erläutert sowohl die Funktionsweise als auch die Optimierung der eingesetzten neuronalen Netze.

2.1.1 Definition und Abgrenzung

Während der Begriff TinyML in der Einleitung eingeführt wurde, bedarf es für die Analyse einer Abgrenzung. Ray definiert TinyML als einen Paradigmenwechsel, der darauf abzielt, maschinelles Lernen von High-End-Systemen (Cloud/Server) auf Ressourcenbeschränkte „Low-End“-Clients zu verlagern [7].

Im Gegensatz zum Edge Computing, das oft auf Einplatinencomputern (wie Raspberry Pi) oder Edge-Servern mit Betriebssystemen wie Linux läuft, fokussiert sich TinyML auf Mikrocontroller (engl. Microcontroller Unit (MCU)). Diese Geräte operieren ohne vollwertiges Betriebssystem (Bare-Metal oder Real-Time Operating System (RTOS)) und müssen mit limitiertem Speicher (oft im Kilobyte-Bereich für SRAM) auskommen [1].

Wie in Abbildung 2.1 dargestellt, ermöglicht TinyML eine sehr gute Rechenleistung bei minimalem Energieverbrauch. Ziel ist es, die Datenverarbeitung auf dem Gerät in Echtzeit (<1 ms bis wenige ms) auszuführen, während der Energiebedarf im Milliwatt-Bereich verbleibt.

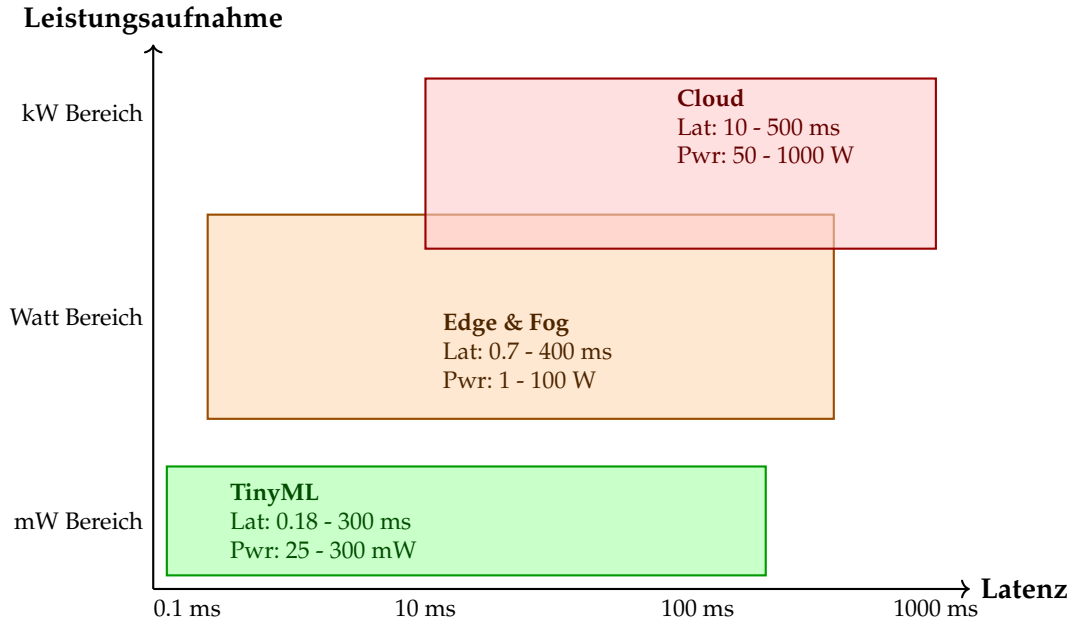


Abbildung 2.1: Vergleich der Latenz- und Leistungsbereiche von Cloud, Edge und TinyML (Datenbasis: [1]).

2.1.2 Funktionsweise Neuronaler Netze auf Mikrocontrollern

Um die Anforderungen an die Hardware zu verstehen, ist ein Blick auf die Struktur der verwendeten Algorithmen notwendig. In TinyML-Anwendungen kommen laut Lai et al. primär Deep Neural Networks (DNNs) zum Einsatz [9]. Unabhängig vom Anwendungsfall folgen diese Modelle meist demselben Aufbau.

Abbildung 2.2 visualisiert die Topologie eines solchen Netzes. Die Eingabedaten (links) durchlaufen mehrere Verarbeitungsschichten, bis am Ende (rechts) ein Ergebnis ausgegeben wird.

Ein DNN verarbeitet Informationen in einer Kette von sequenziell geschalteten Schichten (engl. Layer). Die Datenstruktur, die dabei – vom Eingang über die Zwischenschritte bis zum Ausgang – verwendet wird, ist das mehrdimensionale Array, technisch Tensor genannt [10].

Programmiertechnisch entspricht ein Tensor einer n-dimensionalen Matrix. Ein Farbbild mit 32×32 Pixeln wird beispielsweise als Input-Tensor der Dimension $32 \times 32 \times 3$

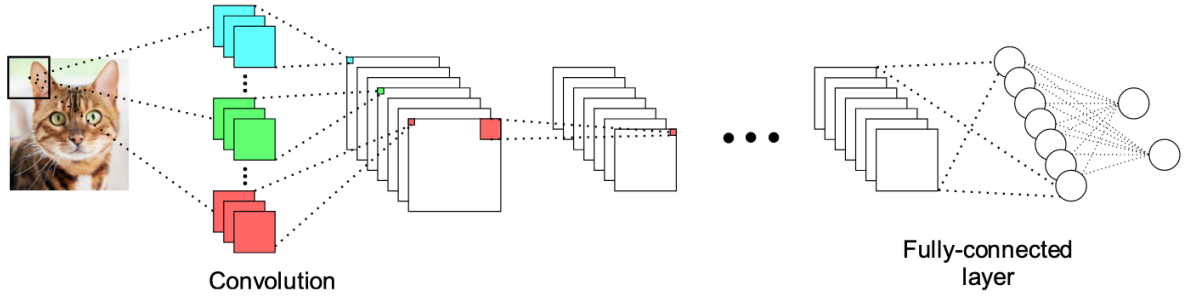


Abbildung 2.2: Struktur eines typischen Deep Neural Network (DNN) [9].

(Höhe × Breite × 3 Farbkanäle) in das Netzwerk gegeben. Jede Schicht transformiert diesen Tensor, indem sie dessen Werte mit den trainierten Parametern (Gewichten) verrechnet. Das Ergebnis wird an die nächsten Schichten übergeben, bis am Ende der Kette ein Output-Tensor mit den Ergebnissen ausgegeben wird [10].

Aus Sicht der Hardware-Architektur lassen sich zwei zentrale Kategorien von Schichten unterscheiden, die unterschiedliche Anforderungen an die Ressourcen stellen [9]:

- **Convolutional Layers (Faltungsschichten):** Diese befinden sich meist am Anfang des Netzes (in Abbildung 2.2 links). Sie sind rechenintensiv, da hier der Großteil der mathematischen Operationen (Multiply-Accumulate (MAC)) stattfindet. Hierfür sind optimierte Digital Signal Processing (DSP)-Instruktionen oder Vektor-Erweiterungen relevant (weitere Ausführungen in Abschnitt 2.2).
- **Fully Connected Layers:** Diese befinden sich am Ende des Netzes (in Abbildung 2.2 rechts). Sie sind speicherintensiv, da sie eine große Anzahl an Gewichten benötigen, die aus dem Flash-Speicher geladen werden müssen.

2.1.3 Modell-Optimierung/Quantisierung

Da moderne DNNs oft zu speicherintensiv für Mikrocontroller sind, ist eine Optimierung der Modellgröße unerlässlich. Die Methode hierbei ist die Quantisierung, bei der die Parameter von 32-Bit Fließkommazahlen (Float32) auf 8-Bit Ganzzahlen (Int8) reduziert werden [7].

Dieser Schritt verringert den Speicherbedarf um den Faktor vier und hat Auswirkungen auf die Hardware-Anforderungen: Durch die Reduktion auf Int8 können Berechnungen auch auf Prozessoren ohne dedizierte Fließkomma-Einheit (engl. Floating Point Unit (FPU)) ausgeführt werden [11].

2.2 Hardware-Architekturen für TinyML

Die Wahl der Hardware-Architektur ist für TinyML-Anwendungen von Bedeutung, da sie die technischen Grenzen für Latenz und Energieeffizienz festlegt. Banbury et al. unterscheiden grundsätzlich zwischen universellen Mikrocontrollern (engl. General Purpose MCUs) und spezialisierter Hardware. Während spezialisierte Custom-Chips Inferenz-Energien von ca. 1 µJ pro Inferenz erreichen können, bieten universelle MCUs den Vorteil der Verfügbarkeit, niedrigen Kosten und einfachen Programmierbarkeit [12]. Vor diesem Hintergrund beschränkt sich diese Arbeit auf universelle Plattformen. Es werden drei Standard-Architekturen untersucht: Der ARM Cortex-M7 als Leistungsreferenz sowie die Xtensa LX6 und LX7 Architekturen der ESP32-Serie als Herausforderer.

2.2.1 ARM Cortex-M7 Architektur

Der ARM Cortex-M7 gilt als ein leistungsstarker Kern im Mikrocontroller-Bereich [13] und bildet das Herzstück vieler High-End-Plattformen, darunter der verwendete Teensy 4.0 sowie der Arduino Giga R1.

Basierend auf der Dokumentation von ARM [14], zeichnet sich die Architektur durch ein superskalares Design aus. Das bedeutet, dass der Prozessor über parallele Ausführungseinheiten verfügt und so in der Lage ist, zwei Befehle im selben Taktzyklus abzuarbeiten, sofern diese nicht voneinander abhängen. Für TinyML ist laut Banbury et al. die integrierte DSP-Erweiterung relevant, die einfache Single Instruction, Multiple Data (SIMD)-Operationen ermöglicht [12]. Dabei wird ein standardmäßiges 32-Bit-Register logisch unterteilt, um beispielsweise zwei 16-Bit-Multiplikationen parallel mit einem einzigen Befehl auszuführen (z.B. Signed Multiply Accumulate Dual (SMLAD)). Da quantisierte Neuronale Netze (NNs) oft auf solchen 8-Bit oder 16-Bit Operationen basieren, steigert dies die Rechenperformance.

Tightly Coupled Memory (TCM): Ein Schlüsselelement ist das TCM. Da dieser Speicher an den Prozessorkern gekoppelt ist, garantiert er den Zugriff innerhalb eines einzigen Taktzyklus. Zeitkritischer Programmcode liegt dort bereit. Die Central Processing Unit (CPU) muss nicht auf Rechenanweisungen warten, was selbst beim Nachladen externer Daten einen deterministischen Rechenfluss sichert.

2.2.2 Xtensa LX6 und LX7 Architektur (ESP32-Serie)

Die Prozessoren der Firma Espressif basieren auf der Xtensa-Architektur von Cadence/ Tensilica [15]. Hier zeigen sich Unterschiede zwischen den Generationen.

Xtensa LX6 (ESP32): Der generische ESP32 nutzt den LX6-Kern. Gemäß dem technischen Referenzhandbuch [16] ist dieser Kern im Vergleich zum Cortex-M7 architekturell einfacher aufgebaut. Er arbeitet Instruktionen sequenziell nacheinander ab (keine superskalare Ausführung). Zwar verfügt der LX6 über DSP-Befehle zur Audio-Verarbeitung, ihm fehlen jedoch spezielle Befehlssatzerweiterungen für neuronale Netze. TinyML-Frameworks müssen Berechnungen hier nacheinander ausführen, was die Effizienz bei großen KI-Modellen einschränkt.

Xtensa LX7 (ESP32-S3): Der Nachfolger ESP32-S3 setzt auf den neueren LX7-Kern. Laut den Spezifikationen von Espressif [17] besteht die Neuerung für KI-Anwendungen in der Einführung dedizierter Vektor-Instruktionen. Hier liegt ein Unterschied zum Cortex-M7 vor: Während der M7 existierende 32-Bit-Register unterteilt (SIMD), führt der LX7 128-Bit-Vektorregister ein. Dies erlaubt es, mehr Daten parallel zu verarbeiten – etwa mehrere Multiplikationen und Additionen in einem einzigen Schritt. Diese Erweiterung wurde entwickelt, um die in neuronalen Netzen dominanten Matrix-Operationen zu beschleunigen und schließt die Lücke zu teureren High-Performance-MCUs.

Speicheranbindung (Cache vs. TCM): Der ESP32 nutzt kein deterministisches TCM, sondern greift über Caches auf Daten zu. Da große Modelle aus dem externen Flash-Speicher nachgeladen werden müssen, entstehen Wartezeiten für die CPU.

Zusammenfassend stellt Tabelle 2.1 die technischen Unterschiede der diskutierten Architekturen gegenüber und ordnet diese in den Kontext von TinyML-Anwendungen ein.

Tabelle 2.1: Vergleich der Hardware-Architekturen

Feature	**ARM Cortex-M7** *(Teensy 4.0, Giga R1)*	**Xtensa LX6** *(ESP32)*	**Xtensa LX7** *(ESP32-S3)*
Pipeline	6-stufig Superskalar (Dual-Issue)	5-7 stufig	5-7 stufig
KI-Beschleunigung	SIMD (DSP-Extension)	Keine Vektor-Instruktionen	Vektor-Instruktionen
Speicher-Zugriff	TCM (Deterministisch, 0 Wait-States)	SRAM + Cache	SRAM + Cache (Optimierter Bus)
Relevanz für Quantisierung	Hoch (SIMD verarbeitet 2x Int16 / 4x Int8 pro Takt)	Gering (Sequenzielle Abarbeitung)	Hoch (Vektor-Befehle für Int8)

2.3 Softwareumgebung und Inferenz-Frameworks

Nachdem die Voraussetzungen analysiert wurden, widmet sich dieser Teil der Software-Architektur. Um komplexe NNs auf Mikrocontrollern auszuführen, reicht der Programmcode nicht aus. Es bedarf eines Software-Stacks – einer Schichtung aufeinander aufbauender Softwarekomponenten – der die Brücke zwischen dem mathematischen Modell und den Registern des Prozessors bildet.

Abbildung 2.3 veranschaulicht den Aufbau des Systems. Diese Struktur orientiert sich an der von Lai et al. definierten Software-Architektur für eingebettete neuronale Netze [9] und erweitert diese um die Komponenten der Espressif-Plattform.

Die Architektur gliedert sich vertikal in drei funktionale Ebenen:

1. **Application Layer:** Er steuert den Programmablauf. Er nimmt die Eingabedaten entgegen, bringt sie in das richtige Format und ruft das neuronale Netz zur Berechnung auf.
2. **Inferenz Engine:** Diese Middleware lädt das Netz und führt die definierten Rechenoperationen Schritt für Schritt aus.
3. **Kernel-Bibliotheken:** Sie stellen die mathematischen Basisoperationen bereit, optimiert für die jeweilige Hardware.

Eine Analyse der Inferenz-Engine und der Kernel-Bibliotheken erfolgt in den Kapiteln 2.3.2 und 2.3.3.

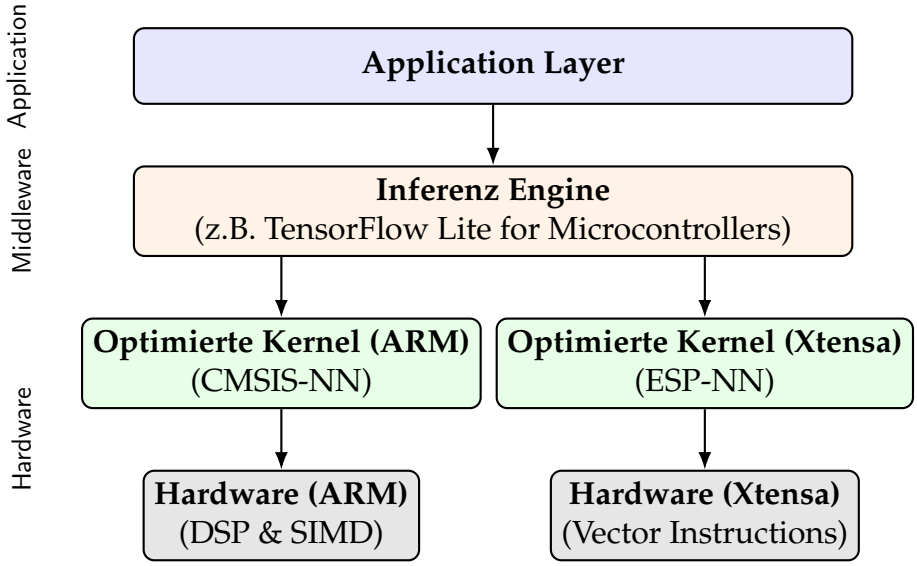


Abbildung 2.3: Schematische Darstellung des Software-Stacks.

2.3.1 Implementierungsstrategien für Embedded ML

Grundsätzlich identifiziert Soro drei Ansätze, um Machine-Learning-Modelle auf eingebetteten Systemen zu implementieren: das manuelle Programmieren, die automatische Code-Generierung und die Nutzung von Inferenz-Engines [18].

Das manuelle Schreiben von C/C++ Code für das neuronale Netz, liefert durch Low-Level-Optimierungen oft die besten Ergebnisse hinsichtlich Ausführungsgeschwindigkeit und Speicherbedarf. Banbury et al. geben jedoch zu bedenken, dass dieser Ansatz zeitaufwendig ist, die Übertragbarkeit auf andere Hardware erschwert und zu undurchsichtigen Optimierungen führt, die die Vergleichbarkeit wissenschaftlicher Ergebnisse behindern [12]. Demgegenüber steht die Code-Generierung, bei der das Modell automatisch in statischen C-Code übersetzt wird. Dies löst das Zeitproblem, bringt laut Abadade et al. jedoch ebenfalls Herausforderungen bei der Übertragbarkeit mit sich, da der generierte Code stark an die Zielhardware gekoppelt sein kann [1].

Aufgrund dieser Limitierungen hat sich in der Industrie und Forschung die Nutzung dedizierter TinyML-Frameworks als Standard durchgesetzt. Diese bieten eine Umgebung, die neben Werkzeugen zur Datenverarbeitung eine flexible Inferenz-Engine bereitstellt [1].

2.3.2 TensorFlow Lite for Microcontrollers (TFLM)

Obwohl diverse Inferenz-Engines wie MicroTVM, emlearn oder uTensor existieren, hat sich TensorFlow Lite for Microcontrollers (TFLM) als Lösung etabliert [11]. Garai hebt hervor, dass TFLM mehr als eine Inferenz-Engine ist: Es stellt die Toolchain (wie Konverter und Quantisierungs-Werkzeuge) bereit, um Modelle sowohl auf ARM Cortex-M Architekturen als auch auf den Xtensa LX6 und LX7 Kernen der ESP32-Serie auszuführen [19].

Ein Vorteil von TFLM ist seine Rolle als technische Basis für High-Level-Plattformen wie Edge Impulse, was seine Verbreitung fördert. Ein weiterer Aspekt ist die effiziente Ressourcennutzung: Experimentelle Untersuchungen von Garai zeigen, dass TFLM den Speicherbedarf im Vergleich zu herkömmlichen, nicht für Embedded-Systeme optimierten Laufzeitumgebungen (z.B. Standard-Keras auf dem Personal Computer (PC)) um bis zu 69 % senken kann [19]. Diese Reduktion bezieht sich primär auf die Größe des Modells selbst (Flash-Speicher). Erreicht wird dies durch die Konvertierung des KI-Modells in das TensorFlow-Lite (TFLite)-Format, welches Techniken wie die Quantisierung (siehe Unterabschnitt 2.1.1) nutzt [11].

Neben der statischen Modellgröße ist die Verwaltung des Arbeitsspeichers (SRAM) zur Laufzeit wichtig. Hierfür nutzt TFLM ein Konzept namens *TensorArena*. Dieser Ansatz ermöglicht es, auf dynamische Speicherallokation (wie `malloc`) zu verzichten und verhindert Speicherfragmentierung. Stattdessen muss der Entwickler einen großen und zusammenhängenden Speicherblock im SRAM reservieren – die TensorArena.

Ein Vorteil dieses Prinzips ist, dass das Modell nicht vollständig in den Arbeitsspeicher geladen werden muss. Da die statischen Gewichte im Flash verbleiben, ist die erforderliche TensorArena in der Regel kleiner als das Gesamtmodell, was die Effizienz der Speichernutzung steigert.

In diesem reservierten Bereich platziert die Inferenz-Engine die variablen Daten: die permanenten Eingabe- und Ausgabe-Tensoren sowie die temporären Zwischenergebnisse der mittleren Schichten. Da Speicher auf Mikrocontrollern knapp ist, verwendet die Arena eine Strategie zur Wiederverwendung: Speicherbereiche von Zwischenergebnissen, die für nachfolgende Berechnungen nicht mehr benötigt werden, werden für die Tensoren der nächsten Schicht überschrieben. Die Größe der TensorArena definiert die Obergrenze des Random-Access Memory (RAM)-Verbrauchs und muss auf die maximale Auslastung des Modells abgestimmt sein [11].

2.3.3 Hardware-Abstraktion durch Optimierte Kernel

Die Inferenz-Engine (TFLM) steuert den Ablauf des neuronalen Netzes und entscheidet lediglich, *welche* Operation (z. B. eine Faltung) als Nächstes ausgeführt werden muss. Für die Berechnung dieser Operation nutzt sie optimierte Kernel. Dabei handelt es sich um spezialisierte Algorithmen, die in hardware-spezifischen Bibliotheken bereitgestellt werden. Diese Kernel bilden die unterste Schicht des Software-Stacks und nutzen für die Berechnung die in Abschnitt 2.2 beschriebenen SIMD- und Vektor-Instruktionen der Prozessoren.

CMSIS-NN (ARM Architektur)

Für Mikrocontroller basierend auf der ARM Cortex-M Architektur (siehe Unterabschnitt 2.2.1) kommt die Common Microcontroller Software Interface Standard – Neural Network (CMSIS-NN) Bibliothek zum Einsatz [9]. Abadade et al. beschreiben dies als eine Kollektion optimierter neuronaler Netzwerk-Kernel, die mit dem Ziel entwickelt wurden, den Speicherbedarf zu minimieren und gleichzeitig die Rechenleistung zu maximieren [1]. Technisch wird dies durch die Nutzung der DSP-Erweiterungen und SIMD-Befehle der Cortex-M4 und M7 Prozessoren erreicht [13]. Anstatt Berechnungen einzeln durchzuführen, erlauben diese Kernel die parallele Verarbeitung mehrerer Datenpunkte in einem Taktzyklus, was für die Matrix-Operationen quantisierter Netze essenziell ist.

ESP-NN (Espressif Architektur)

Analog dazu stellt Espressif für seine Chipsätze die ESP-NN Bibliothek bereit [20]. Diese Bibliothek fungiert als Gegenstück zu CMSIS-NN für die Xtensa-Architektur und stellt sicher, dass TFLM die Vorteile der Hardware nutzt.

Die Optimierungsstrategien unterscheiden sich je nach Chip-Generation. Beim generischen ESP32 (LX6), dem Vektor-Befehle fehlen, setzt die Bibliothek auf handoptimierte Assembler-Routinen. Diese ordnen die Befehlsabfolge so an, dass Leerlaufzeiten im Prozessor minimiert werden und die Rechenwerke ausgelastet sind. Beim neueren ESP32-S3 (LX7) greift ESP-NN auf die im technischen Referenzhandbuch beschriebenen Vektor-Instruktionen zurück [17]. Dies ermöglicht es, mathematische Operationen parallel statt sequenziell zu berechnen, was die Inferenzgeschwindigkeit gegenüber dem Vorgänger steigert.

2.4 Standardisiertes Benchmarking für Embedded ML

Nachdem die Hardware-Architekturen und der Software-Stack erläutert wurden, stellt sich die Frage, wie die Leistungsfähigkeit dieser Systeme bewertet werden kann. In der Praxis erweist sich der Vergleich von TinyML-Lösungen als komplex, da die Ergebnisse von der Kombination aus Hardware, Compiler-Optimierungen und genutzten Modell-Varianten abhängen.

2.4.1 Herausforderungen und die Notwendigkeit von Standards

Das Problem beim Vergleich von Embedded-Systemen ist das Fehlen einheitlicher Messverfahren. Banbury et al. kritisieren, dass veröffentlichte Leistungsdaten oft auf „Best-Case"-Szenarien basieren, die von Herstellern zu Marketingzwecken optimiert wurden [12]. Solche Zahlen spiegeln selten die reale Leistung in einer Applikation wider. Ein Prozessor mag theoretisch eine hohe Rechenleistung aufweisen. Wenn der Software-Stack diese nicht effizient nutzt oder der Speicherzugriff limitiert ist, ist der Wert für den Anwender nutzlos. Ohne einen standardisierten Benchmark ist es für Entwickler nicht möglich, fundierte Entscheidungen zwischen Plattformen wie dem ESP32 und dem Teensy 4.0 zu treffen.

2.4.2 MLPerf und die MLCommons Allianz

Als Antwort auf die Fragmentierung der Testlandschaft etablierte sich Machine Learning Performance (MLPerf) als Standard. Getragen wird dieser von der MLCommons Allianz, einem Konsortium aus Industrie und Forschung, das neutrale und reproduzierbare Vergleiche gewährleisten soll [21].

Für die Bewertung von TinyML-Systemen ist die *MLPerf Tiny* Suite maßgeblich. Zur Gewährleistung der Vergleichbarkeit wird häufig die darin definierte *Closed Division* herangezogen. Im Gegensatz zur *Open Division*, welche weitreichende Modelloptimierungen erlaubt, sind in der *Closed Division* die KI-Modelle fest vorgegeben, um den Fokus rein auf die Hardware-Effizienz zu legen.

Ein Merkmal des Standards ist der automatisierte Messablauf. Um menschliche Fehler auszuschließen, kommt der *EEMBC MLPerf Runner* zum Einsatz. Dieses Framework steuert die Zufuhr der Testdaten sowie die Erfassung der Metriken über eine definierte Schnittstelle und garantiert identische Protokoll-Bedingungen für alle Messungen [22].

2.4.3 Relevante Metriken für TinyML

Für die Bewertung von TinyML-Systemen definiert Banbury et al. im Rahmen der MLPerf Tiny Spezifikation drei Metriken, die auch als Grundlage für die Messungen in dieser Arbeit dienen [4]:

1. **Genauigkeit (engl. Accuracy):** Sie beschreibt den Anteil der korrekt getroffenen Vorhersagen im Verhältnis zur Gesamtzahl der durchgeführten Tests.
2. **Latenz (engl. Latency):** Sie beschreibt die Zeitspanne, die für eine einzelne Inferenz benötigt wird – gemessen vom Moment, in dem der Input-Tensor an das Modell übergeben wird, bis zu dem Zeitpunkt, an dem das fertige Ergebnis bereitsteht. Sie wird typischerweise in Millisekunden gemessen. MLPerf nutzt hierfür oft den Kehrwert, den Durchsatz (engl. Throughput), angegeben in Inferenzen pro Sekunde.
3. **Energieverbrauch (engl. Energy Consumption):** Da TinyML-Geräte oft batteriebetrieben sind, ist die Energieeffizienz ebenso wichtig wie die Latenz. Gemessen wird die benötigte Energie pro Inferenz (in Mikrojoule, µJ).

Durch die kombinierte Betrachtung dieser drei Dimensionen – Genauigkeit, Latenz und Energie – ermöglicht das MLPerf-Framework eine Bewertung, die über reine Taktraten oder theoretische Rechenleistung hinausgeht.

3 Methodik und Versuchsaufbau

Dieses Kapitel beschreibt das methodische Vorgehen und den Versuchsaufbau zur Beantwortung der Forschungsfrage. Um die Leistungsfähigkeit des ESP32 im Vergleich zu High-Performance-Mikrocontrollern objektiv zu bewerten, wurde ein standardisierter experimenteller Aufbau gewählt. Im Folgenden werden das Versuchsdesign, die ausgewählten Hardware- und Software-Komponenten sowie der physische Messaufbau erläutert.

3.1 Versuchsdesign und Metriken

Die Basis für den quantitativen Vergleich bildet der MLPerf Tiny Benchmark. Die Wahl fiel auf diesen Industriestandard, um die in Unterabschnitt 2.4.1 diskutierte Problematik zwischen theoretischen Datenblattwerten und realer Anwendungsleistung aufzulösen.

3.1.1 Benchmark-Strategie: Closed Division

Für die Untersuchung wurde die Closed Division von MLPerf gewählt. In dieser Kategorie sind die NNs vorgegeben und dürfen nicht verändert werden. Dies ist für das Ziel dieser Arbeit essenziell: Da der Einfluss der Hardware auf die Inferenzleistung isoliert untersucht werden soll, müssen alle anderen Variablen konstant gehalten werden. Eine Modifikation der Modelle (wie in der Open Division) würde die Ergebnisse verzerren, da Leistungsunterschiede dann nicht mehr eindeutig der Hardware-Architektur zugeordnet werden könnten.

Um faire Wettbewerbsbedingungen zu schaffen, wurde für jede Plattform das Ziel verfolgt, die bestmögliche Leistung zu erzielen. Dies beinhaltet die Verwendung optimierter Kernel sowie die Allokation der Tensor Arena im schnellstmöglichen RAM-Bereich.

3.1.2 Ausgewählte KI-Modelle

Aus der MLPerf Tiny Suite wurden drei repräsentative Anwendungsfälle mit vortrainierten KI-Modellen ausgewählt. Die Wahl der neuronalen Architekturen variiert dabei bewusst, um unterschiedliche Belastungsprofile für Speicher und Rechenwerk zu erzeugen. Die Speichergröße der verwendeten quantisierten TFLite-Modelle unterscheidet sich dabei deutlich (siehe Tabelle 3.1).

- **Keyword Spotting (KWS) (Keyword Spotting):** Dieses Modell dient der Erkennung von Schlüsselwörtern (z.B. „Yes“, „No“). Technisch basiert es auf einem Depthwise Seperable Convolutional Neural Network (DS-CNN). Diese Architektur ist darauf optimiert, mittels effizienter Faltungsoperationen akustische Merkmale mit minimaler Rechenlast zu extrahieren. Mit einer Größe von **52,7 kB** stellt es die geringsten Anforderungen an den Flash-Speicher.
- **Image Classification (IC) (Image Classification):** Die Klassifizierung von Bildern in 10 Kategorien erfolgt durch ein Residual Network (ResNet). Im Gegensatz zu den anderen beiden Modellen setzt diese Architektur primär auf klassische Faltungsschichten. Dies führt zu einer hohen Dichte an MAC-Operationen, was das Modell trotz seiner moderaten Speichergröße von **96,2 kB** zu einer intensiven Belastungsprobe für die Rechenleistung macht.
- **Visual Wake Words (VWW) (Visual Wake Words):** Diese binäre Bildklassifizierung („Person anwesend“ ja/nein) basiert auf der MobileNetV1-Architektur. MobileNet nutzt – ähnlich wie das KWS-Modell – optimierte Faltungsstrukturen, die weniger Rechenleistung benötigen als klassische Verfahren. Diese Technik reduziert die mathematische Komplexität, benötigt jedoch aufgrund der höheren Anzahl an Parametern mehr Speicherplatz (**325,5 kB**).

Tabelle 3.1: Übersicht der Größen der quantisierten TFLite-Modelle

Modell / Workload	**Flash-Speicherbedarf**
Keyword Spotting (KWS)	52,7 kB
Image Classification (IC)	96,2 kB
Visual Wake Words (VWW)	325,5 kB

3.1.3 Bewertungsmetriken

Da die verwendete Closed Division die Modelleigenschaften und Gewichte vorgibt, wird die Modellgenauigkeit (engl. Accuracy) in dieser Arbeit als Randbedingung vorausgesetzt.

Der quantitative Vergleich der Hardware-Plattformen konzentriert sich daher auf die folgenden drei Effizienz-Metriken:

1. **Durchsatz (IPS):** Die Anzahl der Inferenzvorgänge, die das System pro Sekunde verarbeiten kann. Diese Metrik dient als Indikator für die reine Rechengeschwindigkeit der CPU-Architektur.
2. **Energie (µJ):** Die benötigte Energiemenge für eine einzelne Inferenz. Diese Metrik ist entscheidend für die Beurteilung der Autonomie bei batteriebetriebenen IoT-Geräten.
3. **Preis-Leistungs-Verhältnis (IPS/€):** Der erzielte Durchsatz im Verhältnis zu den Hardwarekosten. Da TinyML-Komponenten oft in preissensitiven Massenprodukten eingesetzt werden, bewertet diese Metrik die ökonomische Effizienz.

3.2 Die untersuchte Systemumgebung

Für die experimentelle Untersuchung wurde eine Systemumgebung definiert, die vier unterschiedliche Mikrocontroller-Plattformen sowie einen einheitlichen Software-Stack umfasst. Diese Auswahl ermöglicht es, die Leistungsfähigkeit des generischen ESP32 gegen High-End-Lösungen und moderne Architektur-Weiterentwicklungen zu prüfen.

3.2.1 Hardware-Plattformen

Das Hardware-Testfeld setzt den ESP32 in Relation zu einer moderneren Variante seiner eigenen Familie sowie zu leistungsstarken ARM-Cortex-M7-Mikrocontrollern, die als Leistungsobergrenze dienen.

Neben der reinen Taktfrequenz ist die Speicherarchitektur für TinyML von Bedeutung. Da die *Tensor Arena* (siehe Unterabschnitt 2.3.2) im Arbeitsspeicher gehalten werden muss, entscheidet die Größe und Anbindung des RAM darüber, ob komplexere Modelle geladen werden können. Während der klassische ESP32 auf seinen internen Speicher

beschränkt ist, bieten die modernen Vergleichs-Boards Zugriff auf großen, wenn auch langsameren, externen RAM.

Zusätzlich sei angemerkt, dass alle verwendeten Plattformen über ausreichend Flash-Speicher (mindestens 2 MB) verfügen, um selbst das größte untersuchte Modell dauerhaft zu speichern.

Um die ökonomische Effizienz zu bewerten, wurden die Marktpreise beim Distributor *Reichelt Elektronik* (Stand: Dezember 2025) ermittelt. Tabelle 3.2 fasst die technischen Eckdaten zusammen.

Tabelle 3.2: Technische Übersicht der verwendeten Hardware-Plattformen.

Board	**Architektur**	**Takt**	**Intern / Ext. RAM**	**Preis**
ESP32 DevKit V1 [16]	Xtensa LX6	240 MHz	520 kB / –	10,90 €
Arduino Nano ESP32 S3 [17]	Xtensa LX7	240 MHz	512 kB / 8 MB	18,60 €
Teensy 4.0 [23]	Cortex-M7	600 MHz	1 MB (TCM) / –	29,50 €
Arduino Giga R1 [24]	Cortex-M7	480 MHz	1 MB (TCM) / 8 MB	63,80 €

Die Rolle der einzelnen Boards im Rahmen dieser Untersuchung ist wie folgt definiert:

- **ESP32 (DevKit V1):** Dieser Controller ist das primäre Untersuchungsobjekt. Er repräsentiert die kostengünstige Standard-Klasse. Eine Besonderheit ist hier die Speicherarchitektur: Zwar verfügt der Chip nominell über 520 kB SRAM, dieser ist jedoch physisch in Instruction RAM (IRAM) und Data RAM (DRAM) unterteilt. Da TinyML-Modelle primär Datenspeicher benötigen und Systemkomponenten (wie Wi-Fi) ebenfalls Bereiche reservieren, steht effektiv nur ein reduzierter Teil des DRAM zur Verfügung. Ziel ist es zu evaluieren, ob diese Ressourcen ausreichen, um die ausgewählten KI-Modelle auszuführen.
- **Arduino Nano ESP32 S3:** Dieses Board dient als moderner Vergleichspunkt innerhalb der Espressif-Familie. Neben den integrierten Vektor-Instruktionen profitiert es von einer leistungsfähigeren Speicheranbindung. Zwar unterliegt der interne Speicher prinzipiell denselben Restriktionen wie beim Vorgänger (Trennung in IRAM/DRAM, Belegung durch weitere Systemkomponenten), diese werden jedoch durch 8 MB externen RAM kompensiert, wodurch die effektiven Speicherlimits eliminiert sind.
- **Teensy 4.0:** Ausgestattet mit 600 MHz Taktrate und einem eng gekoppelten Speicher (TCM), fungiert der Teensy als Leistungsreferenz. Durch den Verzicht auf

langsamen externen Speicher definiert er das technisch Machbare bezüglich Latenzzeiten. Der Vergleich dient dazu, den Abstand des ESP32 zur Spitzenleistung zu quantifizieren.

- **Arduino Giga R1:** Dieser Mikrocontroller dient als Referenz für eine leistungsstarke Allzweck-Plattform. Mit Cortex-M7, erweitertem Arbeitsspeicher (1 MB intern / 8 MB extern) und dem Mbed-Betriebssystem ist er auf komplexe Multitasking-Anwendungen ausgelegt. Der Vergleich prüft, wie effizient dieses vielseitige System trotz der zusätzlichen Software- und Hardware-Komplexität bei spezialisierten TinyML-Aufgaben gegenüber den kleineren Boards abschneidet.

3.2.2 Software-Stack

Als Inferenz-Engine wurde TFLM gewählt. Neben seiner Rolle als offizielles Referenz-Framework für MLPerf Tiny waren vor allem zwei Gründe ausschlaggebend: Zum einen die breite Übertragbarkeit, die den Einsatz desselben Quellcodes auf Xtensa- und ARM-Architekturen ermöglicht. Zum anderen die native Unterstützung der Hardware-Beschleuniger-Bibliotheken CMSIS-NN und ESP-NN (siehe Unterabschnitt 2.3.3).

Für die Entwicklung und Kompilierung der Software kam PlatformIO (in Visual Studio Code (VS Code)) zum Einsatz.

Um die Reproduzierbarkeit der Messergebnisse zu gewährleisten und Dritten die Validierung der durchgeführten Benchmarks zu ermöglichen, wurde der vollständige Quellcode inklusive der plattformspezifischen Anpassungen und Konfigurationsdateien veröffentlicht. Das Repository ist auf GitHub unter folgender Adresse öffentlich zugänglich: `https://github.com/SamsiGg/1.-Studienarbeit---KI-auf-ESP32.git`

3.3 Messaufbau und Datenerfassung

Die Messung der Energie- und Zeitwerte erfolgte unter kontrollierten Laborbedingungen, um externe Störfaktoren zu minimieren und vergleichbare Ergebnisse zu gewährleisten.

3.3.1 Physischer Aufbau und Energiemessung

Als Messinstrument kam das Joulescope 220 (JS220) zum Einsatz. Neben der Empfehlung durch MLPerf Tiny war primär der große Dynamikumfang der entscheidende Grund für die Verwendung. Dieser ermöglicht es, die Sprünge zwischen Ruhestrom und Volllast ohne den sonst üblichen Datenverlust durch Messbereichsumschaltungen zu erfassen.

Die Spannungsversorgung der Mikrocontroller erfolgte durch eine separate, externe Spannungsquelle. Das JS220 wurde dabei zwischen die Quelle und das Entwicklerboard geschaltet. Messtechnisch erfasst das Gerät dabei den Stromfluss sowie zeitgleich die anliegende Spannung. Durch diese synchrone Erfassung beider Größen lässt sich die momentane Leistungsaufnahme berechnen und durch Integration über die Messdauer der Energiebedarf ermitteln.

Die Stromversorgung erfolgte über die VIN-Pins der jeweiligen Boards. Durch diesen Messaufbau wird die Verlustleistung der On-Board-Spannungswandler in die Messwerte einbezogen, was eine realitätsnahe Beurteilung des Energiebedarfs des Gesamtsystems ermöglicht.

Für die Datenübertragung wurde ein Arduino Uno als serielle Brücke zwischengeschaltet. Diese Konfiguration entkoppelt den Host-PC vom zu testenden Gerät (engl. Device Under Test (DUT)), sodass keine direkte USB-Verbindung bestehen muss. Der Arduino leitet Befehle ausschließlich über die Universal Asynchronous Receiver/Transmitter (UART)-Schnittstelle weiter. Dadurch wird eine betriebsrelevante Stromversorgung des DUT über den Host-PC unterbunden.

Zusätzlich zur Datenleitung ist eine präzise zeitliche Synchronisation zwischen dem Mikrocontroller und dem Messgerät erforderlich. Hierfür ist ein General Purpose Input/Output (GPIO)-Pin des DUT direkt mit den digitalen Eingängen des Joulescope verbunden. Vor und nach jeder Inferenz setzt der Microcontroller eine steigende Flanke, wodurch das Joulescope Zeitstempel (engl. Timestamps) parallel zur Strommessung aufzeichnen kann.

Abbildung 3.1 fasst das schematische Blockschaltbild des Versuchsaufbaus zusammen.

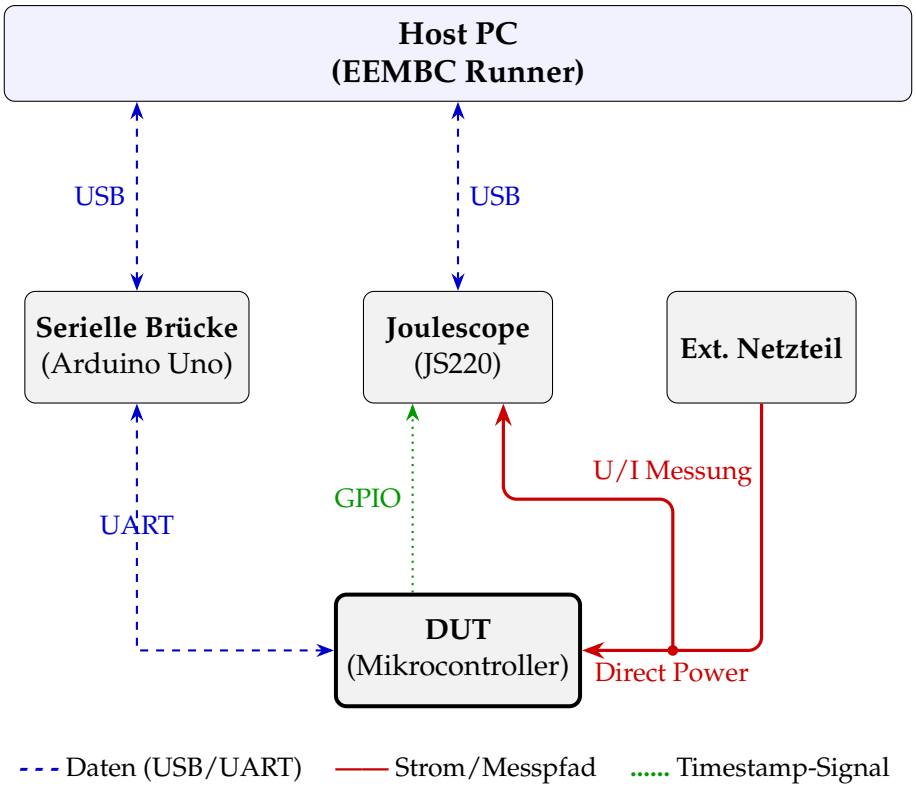


Abbildung 3.1: Verkabelungs-Blockschaltbild der Energiemessung.

3.3.2 Automatisierung und Datenverarbeitung (EEMBC Runner)

Die Steuerung des Benchmarks erfolgte über den *EEMBC MLPerf Runner*. Diese Python-basierten Skripte werden auf dem Host-PC ausgeführt und fungieren als externer Orchestrator, der den Testablauf steuert und die Messdaten erfasst.

Damit das externe Skript mit dem DUT kommunizieren kann, gibt MLPerf ein festes Kommunikationsprotokoll vor. Als Hilfestellung wird eine Software-Struktur bereitgestellt, die definiert, wie auf Befehle reagiert werden soll. Die technische Umsetzung erforderte die in dieser Vorlage nur abstrakt definierten Funktionen mit plattformspezifischem Code zu implementieren. Dazu gehörten:

- Die Initialisierung der Hardware (Clock, UART).
- Die Ausführung der Inferenz über TFLM und die optimierten Kernels.
- Das Setzen von Zeitstempeln und Triggersignalen für die Messung.

Der automatisierte Messablauf gestaltete sich wie folgt:

1. **Handshake und Identifikation:** Das Skript sendet einen Befehl über die serielle Schnittstelle mit Hilfe der seriellen Brücke (Arduino Uno). Die Benchmark-

Software auf dem Mikrocontroller antwortet mit ihrem Namen und dem geladenen Modell, sodass das Skript die passenden Parameter wählen kann.

2. **Daten-Injektion:** Das Skript überträgt die für die Inferenz nötigen Eingabedaten (den Input-Tensor) an den Mikrocontroller. Dieser empfängt den Datenstrom und puffert ihn im Arbeitsspeicher.

3. **Inferenz und Trigger:** Auf Befehl des Skripts führt die Software die Inferenz durch. Kritisch ist hierbei, dass die Software zu Beginn und am Ende der Berechnung ein Signal sendet (Zeitstempel via GPIO-Flanke), auf das das Messgerät sich synchronisiert.

4. **Energiemessung:** Das JS220 erfasst den Energieverbrauch innerhalb dieser von der Software definierten Zeitfenster.

In diesem Kontext ist auf eine Anpassung der Konfiguration hinzuweisen: Während das MLPerf-Protokoll für eine offizielle Zertifizierung längere Laufzeiten zur statistischen Validierung vorsieht, wurde die Messdauer in dieser Arbeit auf unter 10 Sekunden limitiert. Da die Inferenzzeiten und Leistungsaufnahme stabil bleiben, liefert dieses Zeitfenster valide physikalische Messwerte für Latenz und Energie, auch wenn die Ergebnisse dadurch nicht im offiziellen MLCommons-Dashboard gelistet werden können.

Die Ergebnisse wurden automatisch in `.txt`-Logdateien sowie im `.json`-Format gespeichert. Letzteres ermöglichte eine maschinelle Weiterverarbeitung der Daten und bildete die Grundlage für die in Kapitel 4 präsentierten grafischen Auswertungen.

4 Messergebnisse

In diesem Kapitel werden die quantitativen Ergebnisse der durchgeführten Benchmarks präsentiert. Die Darstellung erfolgt gegliedert nach den drei in der Methodik definierten Metriken: Durchsatz, Energieverbrauch pro Inferenz sowie das daraus abgeleitete Preis-Leistungs-Verhältnis (siehe Unterabschnitt 3.1.3).

Da nicht alle Hardware-Modell-Kombinationen erfolgreich realisiert werden konnten, enthalten die Diagramme Lücken. Die Analyse dieser Abweichungen erfolgt in der Diskussion (Kapitel 5).

4.1 Durchsatz und Latenz

Abbildung 4.1 zeigt den gemessenen Durchsatz in Inferenzen pro Sekunde (IPS). Höhere Werte entsprechen einer besseren Leistung.

Über alle Benchmarks hinweg zeigt sich eine Leistungshierarchie: Der Teensy 4.0 (Cortex-M7 @ 600 MHz) erzielt konstant die höchsten Durchsatzraten, gefolgt vom Arduino Giga R1 und dem ESP32-S3. Der generische ESP32 bildet in diesem Vergleich das Schlusslicht.

Beobachtungen:

- **Spitzenleistung:** Beim KWS erreicht der Teensy 4.0 mit über 100 IPS den höchsten Wert der Versuchsreihe.
- **ESP32 Performance:** Obwohl der klassische ESP32 hinter den modernen Architekturen zurückbleibt, erzielt er im KWS-Szenario einen Durchsatz von 4.23 IPS.
- **Auffälligkeit bei (VWW):** Beim VWW-Modell fällt der Durchsatz des S3 (9.03 IPS) im Vergleich zum Durchsatz beim IC-Modell ab, während er beim Teensy 4.0 steigt.

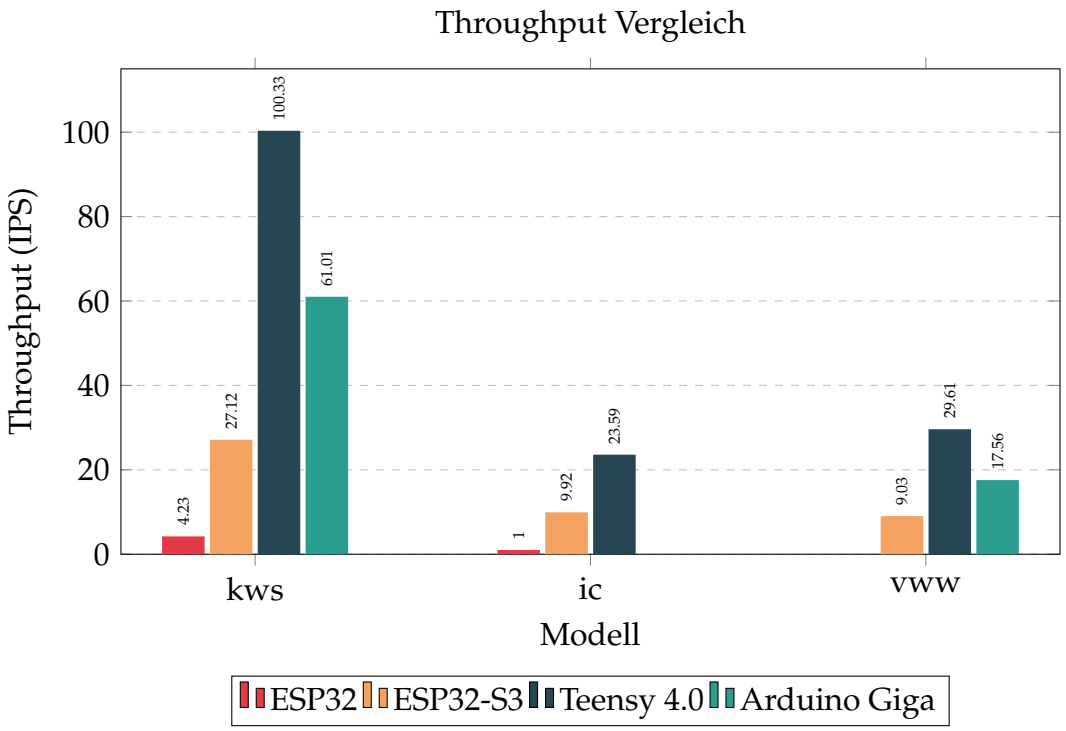


Abbildung 4.1: Throughput in IPS aller Mikrocontroller.

4.2 Energieverbrauch

Abbildung 4.2 stellt den Energiebedarf in Mikrojoule (µJ) pro Inferenz dar. In dieser Darstellung sind niedrigere Werte besser.

Die Ergebnisse im Detail:

- **Effizienzsieger:** Der Teensy 4.0 weist über alle Benchmarks hinweg den geringsten Energieverbrauch pro Inferenz auf.
- **ESP32-S3:** Der S3 positioniert sich im Mittelfeld, zeigt aber eine bessere Effizienz als der Vorgänger ESP32.
- **Verbrauch beim ESP32:** Der generische ESP32 benötigt für das IC-Modell mit ca. 145.000 µJ die größte Energiemenge im Testfeld.

Zur Ermittlung dieser Energiewerte wurde die Leistungsaufnahme im aktiven Inferenz-Betrieb gemessen. Die durchschnittlichen Ergebnisse dieser Messung sind in Tabelle 4.1 zusammengefasst.

Ein Vergleich von Leistungsaufnahme und tatsächlichem Energiebedarf offenbart einen Zusammenhang: Obwohl leistungsstärkere Controller wie der Teensy 4.0 eine höhere momentane Leistungsaufnahme aufweisen (370 mW vs. 150 mW beim ESP32), verbrauchen sie pro Inferenz weniger Energie.

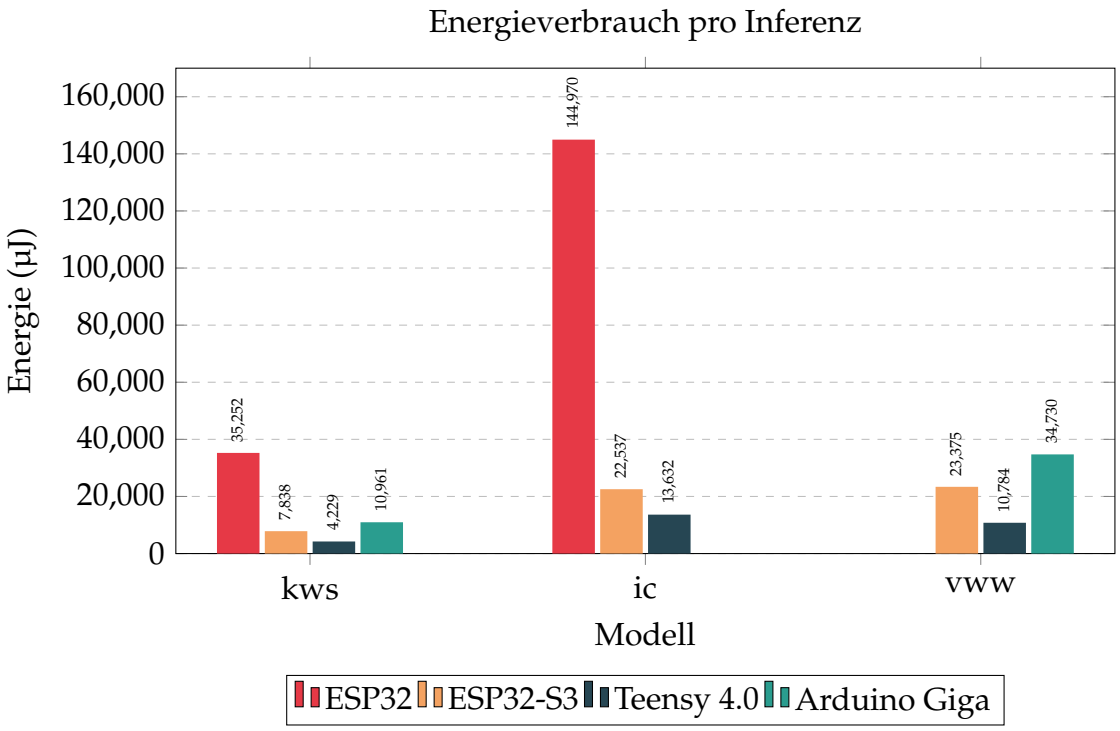


Abbildung 4.2: Energiebedarf in Mikrojoule aller Mikrocontroller.

Tabelle 4.1: Durchschnittliche Leistungsaufnahme während der Inferenz.

Mikrocontroller-Board	**Leistung**
ESP32	150 mW
ESP32-S3	210 mW
Teensy 4.0	370 mW
Arduino Giga R1	680 mW

4.3 Preis-Leistungs-Verhältnis

Um die ökonomische Effizienz der Plattformen zu bewerten, setzt Abbildung 4.3 den gemessenen Durchsatz ins Verhältnis zum Anschaffungspreis der Hardware. Dargestellt ist, wie viel Leistung (in IPS) pro investiertem Euro erhalten wird.

Die Ergebnisse lassen sich wie folgt zusammenfassen:

- Der Teensy 4.0 bietet das beste Preis-Leistungs-Verhältnis (3,40 IPS/€ bei KWS).
- Der ESP32-S3 zeigt sich als Kompromiss und bietet bei KWS (1,46 IPS/€) eine bessere Ökonomie als der generische ESP32 (0,39 IPS/€) oder der Arduino Giga R1 (0,96 IPS/€).
- Der generische ESP32 schneidet in dieser Metrik am schlechtesten ab, da der niedrige Hardwarepreis die geringe Performance nicht kompensieren kann.

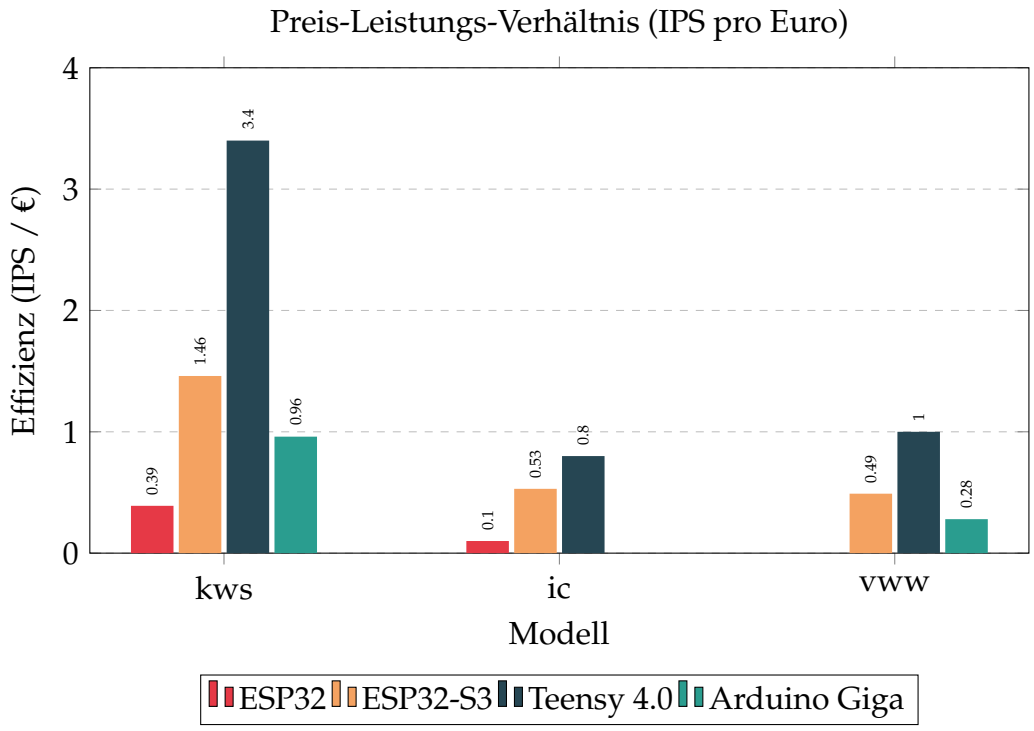


Abbildung 4.3: Preis-Leistung in IPS pro Euro aller Mikrocontroller.

5 Diskussion der Ergebnisse

In diesem Kapitel werden die Messergebnisse analysiert und in den wissenschaftlichen Kontext eingeordnet. Ziel ist es, die Ursachen für die gemessenen Leistungsunterschiede zu ergründen, aufgetretene technische Anomalien zu erklären und die ökonomische Relevanz der Ergebnisse für die Praxis zu bewerten.

5.1 Analyse der Inferenzleistung und Architektur

Die Analyse verdeutlicht, dass spezifische Architekturmerkmale die Leistungsunterschiede definieren. Dabei zeigen sich zwei Erkenntnisse:

Dominanz der ARM Cortex-M7 Architektur: Der Teensy 4.0 dominiert erwartungsgemäß. Dies ist nicht allein auf die höhere Taktfrequenz zurückzuführen, sondern auch auf die Architekturmerkmale, die in Abschnitt 2.2 beschrieben wurden. Die Nutzung des TCM eliminiert Speicherlatenzen fast vollständig, während die superskalare Pipeline des M7-Kerns und die effektive Nutzung der CMSIS-NN-Kernel den Durchsatz maximieren.

Generationssprung beim ESP32 (LX6 vs. LX7): Ein Ergebnis ist der Leistungssprung des ESP32-S3 im Vergleich zu seinem Vorgänger: Dank der Vektor-Instruktionen erreicht er beim KWS-Modell einen 6,4-fachen höheren Durchsatz als der generische ESP32. Angesichts dieser Effizienzsteigerung fällt der Preisunterschied (18,60 € zu 10,80 €) kaum ins Gewicht. Vielmehr demonstriert der S3, wie architektonische Erweiterungen die Distanz zu High-End-Systemen überbrücken können.

5.1.1 Einfluss der Speicherhierarchie und Modellkomplexität

Ein Vergleich der Benchmarks VWW und IC offenbart ein Verhalten, das die Rolle der Speicheranbindung im Zusammenspiel mit der Modell-Architektur verdeutlicht.

Betrachtet man den Teensy 4.0, so ist dieser beim VWW-Modell schneller (29,61 IPS) als beim IC-Modell (23,59 IPS). Dies erscheint zunächst kontraintuitiv, da die Größe des IC-Modells (96,2 kB) deutlich kleiner ist als die des VWW-Modells (325,5 kB). Die Ursache liegt in der internen Struktur der DNNs. Das für VWW genutzte *MobileNet*-Modell wurde spezifisch für mobile Anwendungen entwickelt. Seine Architektur ist darauf optimiert, die Anzahl der in den Grundlagen beschriebenen rechenintensiven MAC-Operationen in den Faltungsschichten (siehe Unterabschnitt 2.1.2) zu reduzieren. Das für IC genutzte *ResNet*-Modell hingegen setzt auf klassische, rechenaufwendigere Faltungsstrukturen. Da beim Teensy beide Modelle vollständig in den schnellen Arbeitsspeicher passen, spielt die Größe der Modelle eine untergeordnete Rolle. Der Teensy kann stattdessen den Vorteil der geringeren mathematischen Komplexität des VWW-Modells ausspielen und erreicht so trotz eines größeren Modells einen höheren Durchsatz.

Beim ESP32-S3 kehrt sich dieses Verhältnis jedoch um: Hier ist VWW (9,03 IPS) langsamer als IC (9,92 IPS), obwohl das Modell eigentlich effizienter wäre. Der Grund hierfür ist eine erzwungene Auslagerung des Speichers. Das VWW-Modell benötigt aufgrund der größeren Eingabedaten und Zwischenspeicher eine *Tensor Arena* von ca. 250 kB. Da der interne SRAM des S3 durch Systemprozesse belegt oder fragmentiert ist, musste die *Tensor Arena* für VWW im externen Arbeitsspeicher platziert werden (ähnlich wie beim generischen ESP32, siehe Abschnitt 5.3). Das kleinere IC-Modell (Tensor Arena ca. 150 kB) fand hingegen im internen SRAM Platz. Auf dem S3 wird der Vorteil der geringeren Rechenlast beim VWW-Modell also durch die langsame Datenübertragung aus dem externen Arbeitsspeicher negiert. Dies belegt den Unterschied zwischen Systemen, die durch Rechenleistung limitiert sind (wie Teensy 4.0 und Arduino Giga R1), und solchen, die auch durch den Speicherzugriff limitiert sind (wie ESP32-S3 und ESP32).

5.2 Analyse der Energieeffizienz

Der leistungsstärkste Chip (Teensy 4.0) weist trotz der höchsten Leistungsaufnahme die beste Energieeffizienz auf. Dies resultiert aus der Inferenzdauer: Die Berechnung erfolgt so schnell, dass die Zeitersparnis den hohen Leistungsbedarf überkompensiert. Der ESP32 benötigt für denselben Rechenprozess ein Vielfaches der Zeit. Diese lange Dauer führt dazu, dass die Gesamtenergie pro Inferenz trotz geringerer momentaner Leistungsaufnahme deutlich höher ausfällt.

Beim Arduino Giga R1 zeigt sich ein Sonderfall. Trotz höherer Rechenleistung als beim ESP32-S3 ist seine Energieeffizienz geringer. Dies ist der Systemarchitektur geschuldet: Als komplexes Board, das für viele verschiedene Anwendungsfälle ausgelegt ist,

besitzt der Giga zahlreiche aktive Peripheriekomponenten und ein im Hintergrund laufendes Mbed-Betriebssystem. Dieser hohe Grundverbrauch kann auch durch den hohen Durchsatz nicht vollständig ausgeglichen werden.

5.3 Technische Limitationen und Anomalien

Wie in Kapitel 4 angedeutet, traten bei zwei Messungen Probleme auf, die somit nicht in den Vergleich aufgenommen wurden, um die Validität der Ergebnisse zu gewährleisten. Die Analyse dieser Fehler liefert jedoch wichtige Erkenntnisse für die praktische Implementierung von TinyML.

Speicherlimitierung beim generischen ESP32 (VWW): Die Nicht-Ausführbarkeit des VWW-Modells auf dem generischen ESP32 bestätigt die diskutierte Speicherproblematik. Im Gegensatz zum S3 verfügt der generische ESP32 meist nicht über externen Arbeitsspeicher (bzw. dieser wird auf dem getesteten DevKit V1 nicht genutzt). Da der interne SRAM nicht genügend zusammenhängenden Platz für die 250 kB große Tensor Arena bot, schlug die Allokation fehl. Dies zeigt, dass komplexe Vision-Modelle auf dem Standard-ESP32 ohne Speichererweiterung kaum realisierbar sind.

Latenzproblematik durch externen RAM beim Arduino Giga: Der für das IC-Modell auf dem Arduino Giga R1 ermittelte Messwert von lediglich 9,81 IPS wich vom erwarteten Leistungsmuster ab. Während der Giga bei anderen Modellen ca. 60 % der Leistung des Teensy 4.0 erreichte, fiel er hier auf unter 42 % zurück. Die Ursache lag in einer fehlerhaften Speicherkonfiguration: Im Programmcode wurde die Größe der *Tensor Arena* versehentlich auf 1024 kB festgelegt. Da der interne SRAM diese Datenmenge nicht aufnehmen kann, allozierte die Software den Speicherbereich stattdessen automatisch im externen Arbeitsspeicher. Der Zugriff auf diesen externen Speicher ist im Vergleich zum via TCM intern angebundenen SRAM signifikant langsamer, was die Inferenzgeschwindigkeit um etwa ein Drittel verlangsamte. Dies dient als ungewolltes, aber lehrreiches Beispiel dafür, wie kritisch das manuelle Speichermanagement bei Cortex-M7-Systemen ist.

5.4 Preis-Leistungs-Bewertung und Einordnung

Sudharsan et al. heben den ESP32 besonders für sein hervorragendes Preis-Leistungs-Verhältnis hervor. Ihr Argument: Mit einem Preis von ca. 3 USD bietet der Chip eine

wirtschaftliche Lösung für KI-Anwendungen, auch wenn teurere High-End-Mikrocontroller schneller rechnen [25].

Die vorliegenden Messergebnisse erfordern jedoch eine differenzierte Betrachtung. Bei rechenintensiven Aufgaben wie der Bildklassifizierung (IC) disqualifiziert sich der ESP32 durch hohe Latenz und schlechte Energieeffizienz. Hier bieten der ESP32-S3 und der Teensy 4.0 trotz höherer Anschaffungskosten das faktisch bessere Preis-Leistungs-Verhältnis.

Anders verhält es sich bei einfacheren TinyML-Anwendungen wie dem KWS. Zwar arbeiten die leistungsstärkeren Mikrocontroller auch hier effizienter, doch der ESP32 liefert mit über 4 Inferenzen pro Sekunde eine für Sprachsteuerungen (z.B. „Licht an“) funktional ausreichende Leistung. Selbst unter Berücksichtigung der hier zugrunde gelegten, höheren Preise (10,80 €) im Vergleich zu Sudharsans Annahme, bestätigt sich dessen These für einfache Modelle: Der ESP32 bietet in diesem Szenario eine akzeptable Performance zu den niedrigsten verfügbaren Einstiegskosten.

6 Fazit und Ausblick

Diese Arbeit evaluierte die technische Eignung des generischen ESP32 sowie des neueren ESP32-S3 für TinyML-Anwendungen im direkten Vergleich zu leistungsstarken ARM Cortex-M7 Systemen. Das primäre Ziel war dabei nicht nur ein ökonomischer Vergleich, sondern die Bestimmung der faktischen Leistungsgrenzen: Es galt zu klären, ob der generische ESP32 überhaupt die notwendigen Ressourcen besitzt, um moderne KI-Aufgaben zu bewältigen, und für welche Anwendungsklassen er eine valide Option darstellt.

Die MLPerf-Benchmarks belegen eine Überlegenheit der ARM Cortex-M7 Architektur (Teensy 4.0 und Arduino Giga R1). Begünstigt durch Hardware-Features wie Tightly Coupled Memory (TCM) und superskalare Pipelines übertreffen Mikrocontroller dieser Architektur den generischen ESP32 bei der Inferenzgeschwindigkeit um Faktoren zwischen 14 (Arduino Giga R1) und 20 (Teensy 4.0).

Ein zentraler Befund der Untersuchung ist, dass eine hohe momentane Leistungsaufnahme nicht zwangsläufig zu einem höheren Energiebedarf pro Inferenz führt. Der Teensy 4.0 bestätigt hier das „Race-to-Sleep“-Prinzip: Zwar benötigt er im aktiven Betrieb den höchsten Strom, kompensiert dies jedoch durch die extrem kurze Berechnungsdauer. Die Inferenz ist in einem Bruchteil der Zeit abgeschlossen, wodurch die absolute Energiemenge pro Rechenoperation geringer ausfällt als beim langsameren ESP32. In der Praxis bedeutet dies, dass leistungsstärkere Controller eine höhere Energieeffizienz aufweisen können, sofern sie die durch die schnelle Verarbeitung gewonnene Zeit in einem energiesparenden Ruhemodus (engl. Deep Sleep) verbringen.

Der ESP32-S3 zeigt als moderne Iteration der Espressif-Architektur, dass sich dieser Leistungsabstand durch Hardware-Optimierung verringern lässt. Dank der integrierten Vektor-Instruktionen konnte er die Lücke zur ARM Cortex-M7 Architektur schließen und die Effizienz gegenüber dem Vorgänger steigern.

Hinsichtlich der Forschungsfrage, ob der ESP32 mit High-Performance-Hardware mithalten kann, muss basierend auf den Ergebnissen differenziert werden:

- **Für komplexe Vision-Anwendungen** erweist sich der generische ESP32 als ungeeignet. Die hohen Latenzzeiten schließen einen produktiven Einsatz aus, während die Speicherlimitierungen die Implementierung größerer KI-Modelle verhindern – wie die Nicht-Ausführbarkeit des VWW-Modells belegt.
- **Für Audio- und Sensor-Anwendungen** stellt der ESP32 eine praktikable Lösung dar. Mit über 4 Inferenzen pro Sekunde beim KWS-Modell werden akzeptable Reaktionszeiten erreicht. Damit eignet er sich für kostensensitive Szenarien, in denen ein niedriger Stückpreis Vorrang vor maximaler Energieeffizienz hat.

Bei der Einordnung dieser Ergebnisse ist jedoch eine kritische Betrachtung der Methodik erforderlich. Die Wahl des MLPerf-Benchmarks in der „Closed Division“ erwies sich als zielführend, um die reine Hardware-Leistung zu isolieren und die Ergebnisse direkt mit anderen wissenschaftlichen Arbeiten vergleichen zu können. Gleichzeitig zeigt dieser Ansatz jedoch nur die Leistungsgrenzen bei Nutzung von Standard-Frameworks (TFLM) auf. Die Frage, „wie viel KI“ auf dem ESP32 bei maximaler Optimierung theoretisch möglich wäre, bleibt offen.

Zudem bilden die gemessenen Werte isolierte Inferenz-Szenarien ab. Da in realen Embedded-Anwendungen fast immer Ressourcen für Kommunikation oder Peripherie benötigt werden, ist die hier ermittelte Leistungsfähigkeit als idealisiertes Szenario zu betrachten. Ungeachtet der formalen Abweichung von den strikten MLPerf-Zertifizierungsregeln sind die ermittelten Daten jedoch physikalisch valide und reproduzierbar. Sie definieren somit präzise die heute faktisch nutzbaren Grenzen des ESP32 für generische Anwendungsfälle.

Für zukünftige Entwicklungen bieten sich jenseits reiner Hardware-Upgrades Optimierungspotenziale durch Software-Stacks und Algorithmen. Saha et al. demonstrieren, dass der Wechsel auf optimierte Frameworks wie Edge Impulse den Speicherbedarf gegenüber der Standard-Implementierung senken kann [26]. Ein weitergehender Ansatz ist das *System-Algorithm Co-Design*: Forschungsarbeiten wie MCUNet (Lin et al.) zeigen, dass mittels Neural Architecture Search (NAS) automatisch Modellstrukturen generiert werden können, die auf die Speicherrestriktionen des Ziel-Chips zugeschnitten sind [27].

Die Kombination aus speichereffizienten Laufzeitumgebungen und automatisierter Architektursuche birgt das Potenzial, komplexe Anwendungen wie Visual Wake Words, die an den Speicherlimitierungen des ESP32 scheiterten, auch auf dieser kostengünstigen Hardware-Klasse zu realisieren.

Literatur

[1] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki und A. S. Hafid, „A Comprehensive Survey on TinyML," *IEEE Access*, Jg. 11, S. 96 892–96 922, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3294111. besucht am 21. Dez. 2025. Adresse: https://ieeexplore.ieee.org/document/10177729/.

[2] A. Tschand u. a., *MLPerf Power: Benchmarking the Energy Efficiency of Machine Learning Systems from Microwatts to Megawatts for Sustainable AI*, 6. Feb. 2025. DOI: 10.48550/arXiv.2410.12032. arXiv: 2410.12032[cs]. besucht am 8. Okt. 2025. Adresse: http://arxiv.org/abs/2410.12032.

[3] H. Han und J. Siebert, „TinyML: A Systematic Review and Synthesis of Existing Research," in *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, Feb. 2022, S. 269–274. DOI: 10.1109/ICAIIC54071.2022.9722636. besucht am 21. Dez. 2025. Adresse: https://ieeexplore.ieee.org/document/9722636/.

[4] C. Banbury u. a., *MLPerf Tiny Benchmark*, 24. Aug. 2021. DOI: 10.48550/arXiv.2106.07597. arXiv: 2106.07597[cs]. besucht am 3. Okt. 2025. Adresse: http://arxiv.org/abs/2106.07597.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li und L. Xu, „Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, Jg. 3, Nr. 5, S. 637–646, Okt. 2016, ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2579198. besucht am 21. Dez. 2025. Adresse: https://ieeexplore.ieee.org/document/7488250/.

[6] L. Greco, G. Percannella, P. Ritrovato, F. Tortorella und M. Vento, „Trends in IoT based solutions for health care: Moving AI to the edge," *Pattern Recognition Letters*, Jg. 135, S. 346–353, Juli 2020, ISSN: 01678655. DOI: 10.1016/j.patrec.2020.05.016. besucht am 21. Dez. 2025. Adresse: https://linkinghub.elsevier.com/retrieve/pii/S0167865520301884.

[7] P. P. Ray, „A review on TinyML: State-of-the-art and prospects," *Journal of King Saud University - Computer and Information Sciences*, Jg. 34, Nr. 4, S. 1595–1623, Apr. 2022, ISSN: 13191578. DOI: 10.1016/j.jksuci.2021.11.019. besucht am 21. Dez. 2025. Adresse: https://linkinghub.elsevier.com/retrieve/pii/S1319157821003335.

[8] S. Kaliner, S. Kawanaka und S. van der Jagt, „Combining Technology Innovations to Make IoT Implementations Cost-Effective and Sustainable," Deutsche Telekom, Murata und Nexperia, White Paper, März 2023, Revision Date: 20 March 2023.

[9] L. Lai, N. Suda und V. Chandra, *CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs*, 19. Jan. 2018. DOI: 10.48550/arXiv.1801.06601. arXiv: 1801.06601[cs]. besucht am 26. Dez. 2025. Adresse: http://arxiv.org/abs/1801.06601.

[10] F. Chollet, *Deep Learning with Python*, 2nd. Shelter Island, NY: Manning Publications, 2021, ISBN: 9781617296864.

[11] R. David u. a., „TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems," in *Proceedings of Machine Learning and Systems (MLSys)*, Bd. 3, 2021, S. 800–811. Adresse: https://proceedings.mlsys.org/paper_files/paper/2021/hash/d2ddea18f00665ce8623e36bd4e3c7c5-Abstract.html.

[12] C. R. Banbury u. a., *Benchmarking TinyML Systems: Challenges and Direction*, 29. Jan. 2021. DOI: 10.48550/arXiv.2003.04821. arXiv: 2003.04821[cs]. besucht am 27. Dez. 2025. Adresse: http://arxiv.org/abs/2003.04821.

[13] H. P B, S. R. Anireddy, J. F T und V. R, „Introduction to ARM processors & its types and overview to cortex m series with deep explanation of each of the processors in this family," in *2022 International Conference on Computer Communication and Informatics (ICCCI)*, Coimbatore, India: IEEE, 25. Jan. 2022, S. 1–8, ISBN: 978-1-6654-8035-2. DOI: 10.1109/ICCCI54379.2022.9740768. besucht am 26. Dez. 2025. Adresse: https://ieeexplore.ieee.org/document/9740768/.

[14] ARM Limited, *ARM cortex-m7 processor technical reference manual*, Revision r1p2, DDI 0489F, Accessed via ARM Developer Documentation, ARM Limited, 2018. Adresse: https://developer.arm.com/documentation/ddi0489/latest.

[15] Cadence Design Systems, *Xtensa instruction set architecture (ISA) reference manual*, Tensilica Processor IP, Cadence Design Systems, Inc., San Jose, CA, 2020.

[16] Espressif Systems, *ESP32 technical reference manual*, Version 5.3, Document ID: ESP32-TRM, Espressif Systems, 2024. Adresse: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[17] Espressif Systems, *ESP32-S3 technical reference manual*, Version 1.2, Document ID: ESP32-S3-TRM, Espressif Systems, 2023. Adresse: https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf.

[18] S. Soro, „TinyML for Ubiquitous Edge AI," *arXiv preprint arXiv:2102.03725*, 2021, Preprint. Adresse: https://arxiv.org/abs/2102.03725.

[19] S. Garai und S. Samui, „Exploring TinyML frameworks for small-footprint keyword spotting: A concise overview,“ in *2024 International Conference on Signal Processing and Communications (SPCOM)*, Bangalore, India: IEEE, 1. Juli 2024, S. 1–5, ISBN: 979-8-3503-5045-6. DOI: 10.1109/SPCOM60851.2024.10631631. besucht am 26. Dez. 2025. Adresse: https://ieeexplore.ieee.org/document/10631631/.

[20] *espressif/esp-nn*, original-date: 2022-01-10T10:47:21Z, 6. Dez. 2025. besucht am 26. Dez. 2025. Adresse: https://github.com/espressif/esp-nn.

[21] „MLCommons,“ besucht am 26. Dez. 2025. Adresse: https://mlcommons.org/.

[22] *eembc/energyrunner*, original-date: 2021-01-11T17:23:47Z, 22. Sep. 2025. besucht am 29. Dez. 2025. Adresse: https://github.com/eembc/energyrunner.

[23] PJRC.com, LLC. „Teensy 4.0 Development Board,“ PJRC, besucht am 30. Dez. 2025. Adresse: https://www.pjrc.com/store/teensy40.html.

[24] Arduino SRL, *Arduino GIGA R1 WiFi User Manual*, SKU: ABX00063. Modified: 2025-12-22, Arduino SRL, Monza, Italy, Dez. 2025. Adresse: https://store.arduino.cc/products/giga-r1-wifi.

[25] B. Sudharsan u. a., „TinyML Benchmark: Executing Fully Connected Neural Networks on Commodity Microcontrollers,“ in *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, Juni 2021, S. 883–884. DOI: 10.1109/WF-IoT51360.2021.9595024. besucht am 29. Sep. 2025. Adresse: https://ieeexplore.ieee.org/document/9595024/.

[26] S. S. Saha, S. S. Sandha und M. Srivastava, „Machine Learning for Microcontroller-Class Hardware: A Review,“ *IEEE Sensors Journal*, Jg. 22, Nr. 22, S. 21 362–21 390, Nov. 2022, ISSN: 1558-1748. DOI: 10.1109/JSEN.2022.3210773. besucht am 2. Jan. 2026. Adresse: https://ieeexplore.ieee.org/document/9912325/.

[27] J. Lin, W.-M. Chen, Y. Lin, j. cohn john, C. Gan und S. Han, „MCUNet: Tiny Deep Learning on IoT Devices,“ in *Advances in Neural Information Processing Systems*, Bd. 33, Curran Associates, Inc., 2020, S. 11 711–11 722. besucht am 2. Jan. 2026. Adresse: https://papers.neurips.cc/paper_files/paper/2020/hash/86c51678350f656dcc7f490a43946ee5-Abstract.html.