information

*Article*

# Reducing the Power Consumption of Edge Devices Supporting Ambient Intelligence Applications

**Anastasios Fanariotis *, Theofanis Orphanoudakis and Vassilis Fotopoulos**

Digital Systems and Media Computing Lab, School of Sciences and Technology, Hellenic Open University, 26334 Patras, Greece; fanis@eap.gr (T.O.); vfotop1@eap.gr (V.F.)
* Correspondence: afanariotis@eap.gr

**Abstract:** Having as a main objective the exploration of power efficiency of microcontrollers running machine learning models, this manuscript contrasts the performance of two types of state-of-the-art microcontrollers, namely ESP32 with an LX6 core and ESP32-S3 with an LX7 core, focusing on the impact of process acceleration technologies like cache memory and vectoring. The research employs experimental methods, where identical machine learning models are run on both microcontrollers under varying conditions, with particular attention to cache optimization and vector instruction utilization. Results indicate a notable difference in power efficiency between the two microcontrollers, directly linked to their respective process acceleration capabilities. The study concludes that while both microcontrollers show efficacy in running machine learning models, ESP32-S3 with an LX7 core demonstrates superior power efficiency, attributable to its advanced vector instruction set and optimized cache memory usage. These findings provide valuable insights for the design of power-efficient embedded systems supporting machine learning for a variety of applications, including IoT and wearable devices, ambient intelligence, and edge computing and pave the way for future research in optimizing machine learning models for low-power, embedded environments.

**Keywords:** Artificial Intelligence; microcontrollers; embedded systems; machine learning; microcontroller power efficiency



**Citation:** Fanariotis, A.; Orphanoudakis, T.; Fotopoulos, V. Reducing the Power Consumption of Edge Devices Supporting Ambient Intelligence Applications. *Information* **2024**, *15*, 161. https://doi.org/10.3390/info15030161

Academic Editors: Lorenzo Carnevale and Massimo Villari

Received: 10 February 2024
Revised: 29 February 2024
Accepted: 6 March 2024
Published: 12 March 2024



## 1. Introduction

During the past two years, computing has reached a significant milestone. This is the achievement of a functional AI model that can communicate with humans using natural language in a way that is almost indistinguishable from that if the result was originated by a machine or another human. This event occurred as an explosive development of various AI-based services, e.g., financial investment advice services or marketing advice services or bringing AI closer to the public and proving that the public is ready to accept any useful AI services. The key element of this event is "communication with humans in a natural manner", a fact that indicates that efficient interaction with the real world is a key element for "smart systems". A natural language may be the basic form of human communication; however, a human communicates and receives stimuli from the surrounding environment in a variety of other multimodal [1] methods. This is where ubiquitous computing rules come into play and embedded systems are required to carry the burden of running AI inference models that may offer humans a more natural and easier way of receiving services. This means that an embedded system may need to be autonomous and efficient in order to achieve its purpose. AI models' inference runtimes are notorious for their latency, which is the real time required to produce a result, and their power requirements, especially when models are complex enough to produce high-level services such as face recognition. This fact drew the attention of various semiconductor manufacturers, which rushed to include function-specific acceleration hardware to small microcontrollers, thus making them more efficient in AI inference latency-wise.

In this manuscript, we are examining if this acceleration hardware also makes these systems more efficient power-wise, thus suitable as autonomous AI systems. To achieve this, we employ a series of experiments where identical machine learning models are executed on both microcontrollers under various conditions. The study meticulously examines power consumption patterns in relation to the utilization of cache memory and vectoring instructions. The ESP32's LX6 core, known for its dual-core architecture, is evaluated against the ESP32-S3's LX7 core, which boasts enhanced computational capabilities and an improved vector instruction set.

Our findings reveal significant insights into how cache memory optimization and vectoring instructions can drastically affect the power efficiency of microcontrollers. The results demonstrate a clear correlation between these process acceleration technologies and reduced energy consumption during machine learning tasks. Additionally, the study provides a comparative analysis of the two microcontrollers' acceleration technologies, offering valuable data for engineers and researchers in selecting the most power-efficient microcontroller for machine learning applications.

## 2. Related Work

The trend to deploy cognitive services boosting the performance of intelligent edge devices has emerged since the introduction of the so-called pervasive and ubiquitous computing paradigm [2]. This paradigm, in turn, can be attributed to rapid advances in microelectronics, on the one hand, and communication technologies, on the other, which have led to packing more and more miniaturized processing, storage, and sensing components in handheld, portable, and always-connected "smart" devices that can support a wide range of environmentally adaptive and context-aware human-centric services. The latest advances in the domain of machine learning (ML) and artificial intelligence (AI) [3] build on the infrastructure of ubiquitous computing and the Internet of Things (IoT) to extend intelligence to the edge of networked systems, leading to a paradigm shift of so-called edge AI [4]. Therefore, in this quest, it is critical to understand and assess performance trade-offs in terms of processing, networking, and storage capabilities as well as autonomous operation of edge devices that support the edge AI model and can lead to enhanced system design and application performance. In this section, we review several works that have addressed the above optimization and performance benchmarking problem by proposing either improved algorithms or improved system designs to accelerate AI processing on edge devices, while maintaining the properties of mobile and autonomous operation (i.e., in terms of system size, weight, power consumption requirements, etc.).

In [5], the authors focus on the feasibility of implementing deep learning (DL) on microcontrollers for human activity recognition using accelerometer data. It compares random forests, a classical machine learning technique, with convolutional neural networks (CNNs) in terms of classification accuracy and inference speed. The study finds that random forest classifiers are faster and nearly as accurate as a custom small CNN model. The paper discusses the potential of DL for modern microcontrollers and questions its suitability compared to traditional methods, especially considering inference speed and energy consumption. Specifically, this paper mentions the use of the CMSIS-NN hardware acceleration library, which significantly impacts the performance of CNNs on microcontrollers. Enabling this library can improve CNN performance by one to two orders of magnitude. This enhancement is particularly crucial for achieving fast inferences on ARM-based microcontrollers. The impact of CMSIS-NN hardware acceleration is highlighted as more significant than mere updates to the instruction sets of newer microcontroller models, and it is considered essential when the CNN architecture is fixed.

The CMSIS-NN library is designed to accelerate ML models, particularly neural networks (NNs), on ARM Cortex-M processors, which are common in microcontrollers. This acceleration is achieved through a combination of strategies, with the usage of SIMD (single instruction, multiple data) instructions being among them.

Overall, the findings underscore the importance of balancing model complexity and power efficiency, especially for applications on resource-constrained devices like microcontrollers. The use of techniques like quantization can help in optimizing models for better power efficiency without substantially compromising performance; however, according to the authors, the CMSIS-NN library's impact is compared to the floating-point unit (FPU). For models that are fully quantized and do not require floating-point operations, the FPU has no effect on speed, whereas the CMSIS-NN library significantly improves performance. This suggests that in scenarios where models are optimized for microcontrollers (including quantization and CMSIS-NN optimization), the overall efficiency, including power efficiency, is likely to be enhanced but only if the model is quantized in a floating-point arithmetic system. This is not taking into account any SIMD or other vectoring instructions available that are likely to improve acceleration through parallelism.

The review performed in [6] provides a comprehensive examination of tools and techniques for efficient edge inference, a key element in AI on edge devices. It discusses the challenges of deploying computationally expensive and power-hungry DL algorithms in end-user applications, especially on resource-constrained devices like mobile phones and wearables. The paper covers four main research directions: novel DL architecture and algorithm design, optimization of existing DL methods, development of algorithm–hardware codesign, and efficient accelerator design for DL deployment. It reviews state-of-the-art tools and techniques for efficient edge inference, emphasizing the balance between computational efficiency and the resource limitations of edge devices. However, the paper also highlights some limitations of MCUs (microcontroller units), such as less memory, slower processing compared to CPUs (central processing units) or GPUs (graphics processing units), lack of parallelization, low clock frequency, and reliance on general-purpose processors that do not support vectorization or thread-level parallelism, indicating that vectorization and parallelism—when implemented—may have a large impact on inference latency and power efficiency.

The work in [7] explores the challenges and methodologies involved in deploying deep neural networks (DNNs) on low-power, 32-bit microcontrollers. It emphasizes the need for optimization in terms of power consumption, memory, and real-time constraints to facilitate edge computing. The paper introduces a framework called "MicroAI" for training, quantization, and deployment of these networks, highlighting the balance between model accuracy, energy efficiency, and memory footprint. The work also includes a comparative analysis of MicroAI with existing embedded AI frameworks and evaluates performance using various datasets and microcontroller platforms. The paper also highlights ongoing work on 8-bit quantization, which improves inference time and memory footprint but at the cost of a slight decrease in accuracy and more complex implementation. The authors focus on several optimization techniques for fixed-point on 8-bit integer inference, including per-filter quantization, asymmetric range, and non-power of two scale factors. Additionally, the use of SIMD (single instruction, multiple data) instructions in the inference engine is anticipated to further decrease inference time.

The work in [8] focuses on deploying ML models on IoT devices efficiently in terms of power consumption. It addresses the challenge of running ML algorithms on embedded devices with limited computational and power resources. The study presents experimental results on the power efficiency of well-known ML models, using optimization methods. It compares the results with the idle power consumption of the systems used. Two different systems with distinct architectures were tested, leading to insights about the power efficiency of each architecture. The paper covers topics like system setup, methodology, selected ML models, and frameworks used, and it discusses the measurements and power efficiency results obtained for both microcontroller architectures and various optimization methods that were applied on the selected ML models. The results show that current optimization methods used for reducing inference latency also reduce power consumption. While the referred work [8] is targeting mostly ML model optimization on a software level and comparing the effect on different generic architectures, we in this paper are focusing

on hardware-level optimization methods that are in fact architecture-agnostic acceleration blocks and may be present in various microcontrollers. In a way, our work is an extension of [8] in the hardware domain.

The PhD thesis of Angelo Garofalo [9] is the latest work to our knowledge that provides detailed insights into vectoring acceleration and SIMD (single instruction, multiple data) instructions, particularly in the context of RISC-V IoT processors and QNN (quantized neural network) acceleration. This work emphasized the fact that modern MCUs lack support at the ISA (instruction set architecture) level for low-bit-width integer SIMD arithmetic instructions. Most commercial ISAs support only 16-bit or 8-bit data. This limitation affects performance and energy efficiency during computation, especially in the context of DNN models. As a first architecture-aware design of AI-based applications, the thesis evaluates kernel computation optimizations by fully exploiting the target RV32IMCXpulp ISA. These kernels are specialized functions or programs/methods used in microcontrollers for processing data in neural networks, specifically in machine learning applications (e.g., matrix multiplication (MatMul) kernels that are used for performing mathematical operations where two arrays (matrices) are multiplied together). It proposed using hardware loops to accelerate 'for' statements and employing 8-bit SIMD instructions to work over multiple SIMD vector elements in parallel, thus increasing the throughput of the computation. Beyond kernel optimizations, the work in [9] discusses the support for various SIMD instructions. In addition to the dot product, the thesis mentions support for other SIMD instructions, such as maximum, minimum, and average for nibble (4 bits) and crumb (2 bits) packed operands. These are particularly useful for speeding up pooling layers and activation layers based on the rectified linear unit (ReLU) function. A range of arithmetic and logic operations (addition, subtraction, and shift) complete the set of XpulpNN SIMD instructions. Furthermore, the author in [9] introduces a multi-precision dot-product unit that computes the dot-product operation between two SIMD registers and accumulates partial results over a 32-bit scalar register in one clock cycle. This unit supports a variety of element sizes in SIMD vectors, ranging from two 16-bit to sixteen 2-bit elements, with the ability to interpret operands as signed or unsigned. To address the saturation problem of the RISC-V encoding space and avoid explicitly encoding all combinations of mixed-precision operands, the thesis proposes a power-aware design with virtual SIMD instructions. These instructions are decoded at the ID (instruction decoding) stage, and their precision is specified by a control and status register written by the processor. Finally, the thesis presents the design of an energy-efficient multi-precision arithmetic unit targeting the computing requirements of low-bit-width QNNs. This includes support for sub-byte SIMD operations (8-, 4-, and 2-bits) and is integrated into a cluster of MCU-class RISC-V cores with a new set of ISA domain-specific instructions, named XpulpNN. This aims to bridge the gap between ISA and hardware to improve computing efficiency for QNN workloads at the extreme edge of IoT.

These findings indicate a significant focus on enhancing computational efficiency and flexibility in processing neural network tasks, particularly in resource-constrained environments like IoT devices with various methods including hardware acceleration such as SIMD instructions. Specifically, we can conclude that a lot of effort is put towards fully exploiting available hardware capabilities of modern microcontrollers and/or making new proposals for the reuse of currently available technologies that may benefit ML model processing acceleration. Most of this work is focused on accelerating ML model inference times, thus making the overall systems more efficient; however, while it is stated in many cases, power efficiency improvement is seldomly—if ever—analyzed in a meaningful manner.

Power efficiency, nowadays, is a necessity for every computational system [10], but additionally in this case, since a microcontroller-based system, which is by definition a resource-constrained system, especially for ML runtime models [11], may need to be autonomous, its power efficiency is a concern that the designer has to take into account. Furthermore, the work in [12] does investigate various studies on hardware-based DNN

inference optimization techniques that include execution parallelization and caching techniques focusing on inference acceleration rather than power efficiency.

In this manuscript, we will investigate the power efficiency aspect of selected hardware acceleration methods, similar to those presented in the related works but that are currently available for modern microcontrollers and thus ready to be used. Our hypothesis is that there are hardware blocks available for current hardware that may prove useful in making ML runtimes in microcontrollers more power efficient.

## 3. Theoretical Basis

Modern microcontrollers carry a vast number of technologies and methods that are used to improve their performance in various functions. In this chapter, we will describe two technologies that are currently used on most available modern microcontrollers. These technologies are cache memory and vectoring instructions. Our work is based on measuring power efficiency of ML model runtimes on two microcontrollers under two main scenarios. One is the use of cache memory under various configurations and the second is the use of vectoring/SIMD instructions. Both of these technologies are described below.

### *3.1. Hierarchical Memory System/Cache Memory*

A hierarchical memory system [13] in a microcontroller is designed to manage data storage and access in an efficient manner [14,15]. This system consists of multiple layers of memory, each with different characteristics like size, speed, and cost. Such systems may, among other layers of memory, include caches that are relatively small but fast memory locations designed to speed up data access to frequently used data. They act as an intermediary between the slow main memory and the fast CPU and usually split into two different operation-based sub-caches. First, the instruction cache (I-Cache) specifically stores pre-fetched instructions, with the purpose of speeding up the execution of programs by minimizing the time taken to fetch instructions from the main memory. Secondly, the data cache (D-Cache) is designed to store and provide quick access to data used by the CPU, helping in reducing the delay in data retrieval from the main memory, which is crucial for efficient processing.

The data flow in this system is managed to ensure that the most frequently accessed data are stored in the fastest memory (like caches), thus reducing the average time to access data. When a microcontroller processes data, it first checks the fastest memory (registers), then the caches, and finally the main memory if the data are not found in the caches.

I-Cache is dedicated to storing pre-fetched instructions. Its primary function is to accelerate the execution of programs by reducing the time taken to fetch instructions from the slower main memory. When the CPU executes a program, it frequently accesses certain instructions. I-Cache keeps a copy of these instructions. By doing this, it minimizes the need for the CPU to repeatedly access the slower main memory for these instructions, leading to faster program execution. I-Cache exploits the principle of "temporal locality", meaning that instructions accessed recently are likely to be accessed again soon. Generally, I-Cache is implemented using high-speed memory technology, like SRAM (static random-access memory). It is closely integrated with the CPU to minimize latency. I-Cache tends to be relatively small but extremely fast. Its effectiveness lies not in its size but in how well it predicts and stores the instructions that the CPU will need next.

D-Cache is dedicated to storing data that the CPU uses. It aims to minimize the delay in data retrieval from the main memory. When the CPU processes data, it often needs to access certain data elements repeatedly. D-Cache stores copies of these data to reduce the need for time-consuming access to the main memory. D-Cache utilizes both "temporal locality" (recently accessed data are likely to be accessed again) and "spatial locality" (data near recently accessed data are likely to be accessed). D-Cache writes policies like write-through and write-back and governs how modifications to data in the cache are handled in relation to the main memory. With the write-through policy, every change in D-Cache is immediately written to the main memory, while with the write-back policy, changes in

D-Cache are written back to the main memory only when the cache line is replaced. Like I-Cache, D-Cache is also small and fast, designed for quick access to data.

The importance for microcontrollers' functionality lies in the effects that, by reducing the frequency of access to slower main memory, caches significantly speed up data processing, and faster data access means reduced CPU idle time and lower energy consumption, which are crucial in embedded systems and IoT devices.

Adding to the above reasoning that most if not all ML models are in fact just tensors densely packed to memory, thus they have increased data locality, especially when we take into account their layer-by-layer formatting and processing flow, we expect that the usage of caches will greatly improve power efficiency.

### *3.2. Vectoring and SIMD Technologies*

A microcontroller vectoring system refers to the ability to handle multiple data operands simultaneously through a single instruction. This is part of a broader concept known as SIMD (single instruction, multiple data) [16]. This system is particularly useful in microcontrollers for tasks that require processing large arrays of data efficiently, like digital signal processing; matrix operations, like ML tensor operations; and handling complex sensor data.

SIMD is a parallel processing method [17] that allows a single instruction to perform the same operation on multiple data points simultaneously. This method increases computational speed for certain tasks by parallelizing the computation on multiple data elements, reducing the total number of instructions required. SIMD is widely used in applications requiring heavy numerical computations, such as graphics processing, scientific simulations, and machine learning tasks.

Vectoring in microcontrollers is similar to SIMD, but specifically tailored for microcontroller environments. It aims to enhance the data processing capabilities of microcontrollers by allowing them to handle multiple data points with a single instruction, similar to SIMD. It is particularly useful in microcontroller-based applications like signal processing, sensor data handling, and IoT devices where efficient data processing is needed.

Both SIMD and vectoring technologies are designed to process multiple data points simultaneously under a single instruction, and they are used to increase computational efficiency, especially in tasks involving large arrays of data. SIMD is a general concept applied in various computing systems, including high-end processors and GPUs, while vectoring in microcontrollers is a more specific application within the constraints of microcontroller architecture. SIMD can be found in a broader range of hardware, from general-purpose CPUs to specialized processing units, whereas vectoring is specifically integrated into the design of certain microcontrollers. SIMD optimizations are often geared towards high-performance and complex computing tasks, while vectoring in microcontrollers is optimized for the efficiency and power constraints typical in embedded system applications. However, ARM does describe the equivalent system for their microcontrollers as SIMD and not as vectoring.

## 4. Experimental Setup

Our experimental setup and methods are focused on measuring power efficiency in ML models running on microcontrollers under various states of hardware capabilities (e.g., disabled cache, limited cache size, etc.). Therefore, our setup must include the necessary laboratory instruments, a group of microcontroller boards as the devices under investigation, and well-defined methods of producing meaningful measurement results.

In our case, we need to measure the power consumption of two microcontrollers while they are running ML models. This means that we have to connect a power measuring instrument to each of the microcontrollers and restrict its measurement sampling to the time window in which the ML model runs. To achieve precise measurements, we utilized a collection of tools comprising a laboratory-grade digital multimeter (DMM), featuring external triggering capabilities, and a bench power supply unit (PSU) capable of delivering

a stable 3.3 Volt output. Furthermore, we employed an extra-precise multimeter to confirm an accurate voltage reading when the MCU was operating under load. We also employed the serial port for each board that we used as a console output in order to obtain internal runtime information for each inference.

### *4.1. Digital Multimeter*

For our DMM, we opted for the Keysight 34465A [18]. This digital multimeter's accuracy can reach up to 0.0035% (+3 counts), depending on the function and range used. The multimeter is equipped with self-calibration and self-diagnostic features to preserve its accuracy over time. An external triggering function allows measurements to be synchronized with external events or signals. This is particularly beneficial in scenarios like data acquisition systems or when precise timing of measurements is crucial since, as we will describe further down, in order to take precise measurements, we need to limit the measurement period to that of the ML runtime and precisely time it by external triggering.

This DMM also provides a range of mathematical functions for data analysis. Users can perform basic arithmetic (addition, subtraction, multiplication, and division) and more complex calculations like square root, square, absolute value, and logarithm. We used this function in order to calculate the power consumption from current readings. The device also features data acquisition and storage capabilities, allowing for sequential measurement recording over time, crucial for later analysis. Users can customize data acquisition by setting parameters like sampling rate and reading count, controllable via the front panel or remotely. The multimeter automatically captures and saves data per these settings. Additionally, the DMM can create, display, and export trend charts, providing a comprehensive view of measurement patterns over time.

To ensure precise measurements using the methodology detailed later, we configured the DMM with the following settings:

- The input was set to the 3A current port.
- The trigger mode was adjusted to respond to external triggering on the negative edge.
- The DMM was placed in continuous acquire mode.
- A measurement delay of 50 microseconds (µS) was established post-triggering.
- The sampling rate was tailored for each model, ensuring that a minimum of three samples could be taken during the inference duration.
- The display mode was switched to show a trend chart.
- A linear mathematical function was implemented to convert the current input (measured in mA) to power input (expressed in mW).

### *4.2. Bench Power Supply Device*

The EL302T bench power supply unit (PSU) [19] is an electronic device used for providing a stable, adjustable source of direct current (DC). This power supply typically features the capability to set specific voltage and current levels within a certain range. Key characteristics of the EL302T include its precision in controlling output voltage and current and its reliability in maintaining stable power output. Additionally, the unit has safety features like overvoltage protection and thermal shutdown to ensure safe operation during extended use.

## 5. ESP32—Selected Development Boards

The ESP32 and ESP32-S3 microcontrollers that we are using are made by Espressif and are available in various forms. Their respective development boards are available from various sources including Espressif. For our experimental setup, we chose two such boards that are low cost and with an abundance of peripherals to support running ML models, such as large amounts of PS RAM (pseudo-static RAM). In order to simplify our setup, we used development boards that offer out-of-the-box connectivity options such as USB/UART and GPIO pins and not barebone modules. Barebone ESP32 comes as an SiP

(system-in-package) module with PCB finger soldering edges and it is rather hard to use it without a suitable carrier board.

### *5.1. ESP32 DevKit TTGO*

The ESP32 Devkit TTGO [20] is a development board built around the ESP32 chip designed for a wide range of IoT (Internet of Things) applications. The DevKit typically comes equipped with numerous GPIO (general-purpose input/output) pins, which are essential for interfacing with various sensors, actuators, and other peripheral devices. The board includes a USB-to-UART bridge, simplifying the process of programming the ESP32 and enabling serial communication with a computer or other devices for collecting our measurements. At its core, the ESP32 is a high-performance microcontroller, boasting significant processing power and memory capacity. Despite its power, the ESP32 is designed to be energy-efficient, a critical feature for battery-operated or power-sensitive applications. In terms of software support, the ESP32 DevKit is compatible with several development environments, including Espressif's IDF, offering flexibility in programming languages and frameworks. We opted for this board because it implemented the maximum size of PSRAM for the ESP32 architecture (4 MB). A large size of memory is required for ML applications. The basic specifications are given in the table below (Table 1).

**Table 1.** ESP32 (LX6) specifications.

| Component | Description |
|---|---|
| MCU core | Tensilica Xtensa® LX6 |
| Available cores | 2 |
| Maximum core frequency | 240 MHz |
| Integration scale | 40 nm |
| SRAM memory (internal) | 520 KB |
| Total cache size | 32 KB + 32 KB |
| External memory | 4 MB QSPI PSRAM [1] |
| Flash storage | 4 MB QSPI Flash |
| Working voltage range | 2.3 V to 3.6 V |
| Power consumption | Varies based on activity and sleep modes |

[1] Limited to 4 MB by architecture.

### *5.2. ESP32-S3-DevKitC-1*

ESP32-S3-DevKitC-1 is an advanced development board based on the ESP32-S3 chip, designed specifically for high-performance IoT applications. This board is an evolution of previous ESP32 models, offering enhanced features suitable for a wide range of applications and ML applications. Similar to ESP32 TTGO, this board features the following:

- GPIO pins: It comes with an array of GPIO pins, enabling easy connections to various sensors, actuators, and other peripherals making external triggering signal generation easy in our case.
- USB interface: The board includes a USB interface for programming and power supply, simplifying the development process that may also be used to produce any runtime data and send it to a console for review.
- Powerful microcontroller: At its core lies the ESP32-S3 chip, which offers significant improvements in processing power and memory compared to earlier model computational resources that are useful for running ML models efficiently.
- AI and machine learning capabilities: ESP32-S3 is equipped to handle AI and machine learning applications, making it suitable for advanced IoT projects.
- Flexible development options: The board supports various development environments, including the Arduino IDE and Espressif's own software development framework (ESP-IDF).

The specifications for this board are as follows (Table 2):

**Table 2.** ESP32-S3 (LX7) specifications.

| Component | Description |
|---|---|
| MCU core | Tensilica Xtensa® LX7 |
| Available cores | 2 |
| Integration scale | 40 nm |
| Maximum core frequency | 240 MHz |
| SRAM memory (internal) | 512 KB |
| Total cache size | 32 KB + 64 KB |
| External memory | 8 MB OSPI PSRAM |
| Flash storage | 8 MB QSPI Flash |
| Wireless connectivity | Bluetooth 5 BR/EDR, BLE/LR, WiFi |
| Working voltage range | 2.3 V to 3.6 V |
| Power consumption | Varies based on activity and sleep modes |

ESP32-S3-DevKitC-1 [21] stands out for its enhanced processing capabilities, AI and machine learning support, and versatile connectivity options, making it a powerful tool for developers looking to create sophisticated and interconnected IoT solutions. Our selected model comes with 8 MB OSPI PSRAM and 8 MB QSPI flash storage.

### *5.3. Evaluation of ESP32 and ESP32-S3 for Edge AI-Enabled Devices*

ESP32 comes with a Tensilica Xtensa LX6 Dual-core, 32-bit processor, which typically runs up to 240 MHz and includes integrated SPI SRAM (up to 4 MB) and SRAM (up to 520 KB). It is designed with power efficiency in mind, but is not specifically optimized for ML applications. However, as indicated in [8], it is capable of running them although with a lower power efficiency compared to other available microcontrollers.

ESP32-S3 comes with a Tensilica Xtensa LX7 Dual-core, 32-bit processor with vector extensions that has improved performance over LX6, especially in computation-heavy applications. It has the same clock as LX6, up to 240 MHz, a similar memory configuration with potential enhancements in its cache system, and its power efficiency is comparable to LX6, with possible additional efficiency gains in ML tasks due to vector extensions.

LX7 cores are more powerful, particularly in ML tasks, due to their architectural enhancements (vectoring). LX6 lacks these vector extensions, making it less efficient for parallel data processing. Both cores are designed for energy efficiency, but LX7′s optimized ML processing can lead to better energy utilization in ML tasks.

### *5.4. Available Development Tools and ML Model Selection*

In order to develop the required firmware that needs to contain both the selected ML models as well as any other signaling generation routines used during measurements, we opted to use ESP-IDF (Espressif IoT Development Framework). ESP-IDF is the official development framework for Espressif's ESP32 and ESP32-S series SoCs (system on chips). It is a comprehensive set of tools and libraries designed to facilitate the creation of applications on the ESP32 platform. At its core, ESP-IDF is built upon FreeRTOS, a real-time operating system that enables multitasking and real-time functionality. This integration of FreeRTOS allows developers to leverage its features such as task scheduling, queues, and inter-task communication mechanisms, enhancing the capability to handle complex operations and multiple processes simultaneously. ESP-IDF supports a wide range of development activities.

Programming in ESP-IDF is primarily conducted in C or C++. SDK provides a comprehensive collection of APIs and libraries, including out-of-the-box TensorFlow Micro (TinyML) [11] support, significantly reducing the complexity of developing low-level code.

ESP-IDF also includes a full ML library suite, ESP-DL, which is a specialized library designed to facilitate the deployment of deep learning (DL) algorithms on Espressif's ESP32 series of microcontrollers. ESP-DL addresses the increased processing requirements of edge AI-capable devices by enabling the integration of machine learning and deep learning

models directly into IoT devices powered by ESP32 chips. One of the key features of ESP-DL is its optimization for the ESP32 hardware architecture. Although ESP32 is a popular microcontroller, its computational resources are limited compared to typical desktop or server environments where deep learning models are usually trained and run. ESP-DL addresses this by providing a set of tools and APIs specifically optimized for ESP32's CPU and memory constraints. This allows for efficient execution of deep learning models on these devices.

ESP-DL supports a range of common neural network layers and architectures [22], making it versatile for various applications. It includes functionalities for basic layers such as convolutional, pooling, and fully connected layers, which are the building blocks of many deep learning models. This allows developers to deploy a variety of neural network models, including those used for image recognition, voice processing, and other sensory data analysis tasks common in IoT applications. Furthermore, ESP-DL aims to simplify the process of bringing pre-trained models onto the ESP32 platform. Typically, models are developed and trained on powerful computing environments using frameworks like TensorFlow or PyTorch. ESP-DL facilitates the conversion and optimization of these models for execution on the ESP32's resource-constrained environment. This library plays a crucial role in edge computing by enabling smarter IoT devices. With ESP-DL, devices can process data locally, reducing the need to constantly send data to the cloud for processing. This not only speeds up response times but also enhances privacy and reduces bandwidth usage and possibly increases energy efficiency.

The ESP32-DL library currently offers four ready-to-run example ML models that vary in computational requirements from medium to extremely high. These models are pre-trained and optimized to run in ESP32 so no other optimization methods are needed in order to run to this SoC. This gives us the opportunity of directly correlating measurement differences to the core technology used (cache/vectoring) since ML model-wise (software), there is no difference in what is running on EPS32(LX6) and ESP32-S3(LX7) SoCs. The four examples are human face detection, human face recognition, color detection, and cat face detection. However, all these models that are included in ESP-DL as examples are offered as precompiled binaries without any option to either modify them or examine their architecture, exposing only a high-level layer access that permits the users to execute various fundamental functions like loading the model's input and initiating an inference. While these models are true "black-boxes", we opted to use them instead of building our own models on TinyML or TFlite because they are compiled under the same library (TensorFlow) version with the same OpCode versions; thus, there is minimal change in compiler optimization variance between them. On the other hand, providing our own models for measurement would most likely require different OpCode versions for some of the modes with a large possibility of affecting the final runtime efficiency of each model.

The human face detection example is centered around identifying human faces within an image using a convolutional neural network (CNN). This example is particularly relevant for real-time applications such as security systems, where detecting the presence of a human face is crucial. The process involves capturing images via a camera module connected to the ESP32, or in our case feeding an image through the device's storage unit, preprocessing these data to fit the mode's requirements, and then using a CNN to detect faces. The challenge lies in optimizing the model for ESP32's limited processing power while maintaining real-time detection capability.

Building on the concept of face detection, the human face recognition example takes it a step further by not only detecting faces but also recognizing and verifying them against a known set of faces. This involves more complex processes, including feature extraction and matching, and requires a more sophisticated approach to model training and implementation. This example has significant implications for personalized user experiences and security applications, such as access control systems, where individual identification is necessary.

The color detection example serves as a demonstration of a color detection service. The code receives a static image composed of various colored blocks as its input. It then produces outputs that include the results of functions such as color enrollment, color detection, color segmentation, and color deletion. These outcomes are displayed in the terminal console.

The cat face detection example diversifies the application of these AI techniques to the animal kingdom, focusing on detecting cat faces. The principles remain similar to human face detection, but the model is specifically trained to recognize feline features. This example illustrates the versatility of the ESP-DL library and ESP32's capabilities, extending its use to scenarios like pet monitoring or automated animal interaction systems. The implementation involves similar steps of image capture, preprocessing, and optimized model inference to detect cat faces effectively. All four examples are not just demonstrations of technical capability but, like in this case, serve as tools for developers looking to implement AI and ML on edge devices while studying any effects during real-world runtime. They highlight the importance of optimizing deep learning models for constrained environments like ESP32 and demonstrate practical approaches to real-time data processing and output handling. Each model's expected runtime characteristics according to Espressif's preliminary analysis available with a description and file size of each example (latency and model size) [17] are presented in the table below (Table 3). The power consumption expectation is based on both model size (memory usage) and latency (computational load).

**Table 3.** Selected ML model characteristics of ESP32 and ESP32-S3 devices.

| | **Human Face Detection** | **Human Face Recognition** | **Color Detection** | **Cat Face Detection** |
|---|---|---|---|---|
| Computational Load | Moderate | High | Moderate | Low |
| Memory Usage | Moderate | High | Low | Low |
| Power Consumption | High | Extremely High | Low | Low |

### *5.5. Comparative Analysis of the Two Boards*

While ESP32 has a fixed size of cache available with 32 Kbytes of D-cache and a fixed two-way associative method, ESP32-S3 has a higher degree of configurability allowing us to define the size of both D-cache and I-Cache by offering a selection of values for D-cache of 16, 32, or 64 Kbytes and 16 or 32 Kbytes for I-cache. The n-way cache associative methods are also selectable as four- or eight-way association.

ESP32 has no vectoring support or any other operational hardware acceleration units (according to its datasheet) other than a 32-bit FPU, which only supports hardware addition and multiplication and should have no effect on 16-bit and 8-bit floating-point operations or integer operations. On the other hand, ESP32-S3 has the support of SIMD/vectoring instructions and, according to its datasheet [23], a similar 32-bit FPU unit. Vectoring support is a core functionality and consists of an expanded set of instructions aimed at enhancing the efficiency of certain tasks. These instructions include general-purpose registers with a wide bit range, alongside a variety of special registers and processor interfaces. By using the SIMD (single instruction, multiple data) approach, the instruction set facilitates vector operations across 8-bit, 16-bit, and 32-bit formats, significantly boosting the speed of data handling. Additionally, arithmetic operations like multiplication, shift, and addition are optimized to conduct data manipulations and transfers concurrently, thereby increasing the performance. The presence of SIMD may affect the usage of the FPU as a 32-bit-only FPU may be capable now of simultaneously running lower data-length FPU instructions by leveraging SIMD functionality, thus expanding FPU hardware acceleration to a 16-bit or even an 8-bit data payload. While under normal function this would have minimal effect, in the case of 8-bit or 16-bit quantized ML models, we expect to see significant improvements in inference time and maybe in power efficiency. Since ESP32-S3 presented such flexibility,

we opted to take multiple measurements based on both D-cache and I-cache size with the hope of concluding how cache type and size affect power efficiency of microcontrollers running ML models. To compare the SIMD/vectoring effects in power efficiency, we ran ESP32-S3 with a similar cache setup as the fixed setup of ESP32.

### *5.6. Measurement Methods and Connectivity*

To be able to observe the functionality of the code without skewing the power measurement results, we opted to use each board's UART output as a debugger console. We achieved this by accessing this port only when the MCU was not running ML model inference.

In order to achieve the most accurate and meaningful results from our experiments, we specifically designed our measurement methodology (in terms of software implementation and hardware setup) by applying the techniques that are detailed in this section. Throughout our code, we have not implemented any sleep or reduced power modes. This decision is based on the fact that this kind of power saving gains are straightforward for end developers to calculate if needed. Additionally, implementing a low power state is not feasible while the core processes the ML model data. Therefore, in our scenario, the use of sleep modes is not particularly important since our focus is on presenting per-inference power measurements, which exclusively include the power used during the inference process itself.

The PSU was adjusted to deliver an output of 3.3 Volts. Following this, the current port of the digital multimeter (DMM) was connected in series with both the PSU and the board undergoing measurement. To capture sample readings exclusively during the inference phase, an output pin on the board was designated as a trigger output. We selected GPIO5 for both boards since this pin was available and not used for any secondary function during the booting of both ESP32 and ESP32-S3. GPIO5 was set as output floating with an active low setting. This minimized any extra power consumption from GPIO peripherals since driving an output high requires, practically, more power than driving it low since the output is connected to the measurement instrument, and although the instrument input theoretically has infinite impedance, in reality a small current flow exists. The output was set to a low level just prior to each inference in code execution in the MCU. This pin was then linked to the external trigger port of the DMM, thereby creating a negative edge trigger event for the DMM. The setup of these connections is illustrated in the diagrams below (Figure 1).

To gain a better understanding of the power efficiency of each MCU, we measured its power consumption during a no-operations loop with all necessary peripherals active. The default clock speeds were left unchanged, meaning that ESP32 operated at 240 MHz across both cores. ESP32's power usage was recorded at 133.75 mW, which is typical for this microcontroller when its WiFi and Bluetooth capabilities are turned off (Figure 2), while for ESP32-S3, the idle power consumption was 153.1 mW (Figure 2). For the inference tests, we enabled the triggering mode. Each measurement's trigger was integrated into the code, marking the start of a new inference (Figure 3). We recorded the total inference time for each device and model, along with the power used for a single inference and the total power consumption for an inference. We then calculated the power consumption only during the time interval of an inference by subtracting idle power from total power consumption. We used the term/method "pure power consumption" as described in [8] to reflect the efficiency of an MCU core in processing the ML model for inference. Real-world power usage is closer to the total inference power since pure power does not account for necessary components like clocks, phase-locked loop subsystems (PLLs), or internal bus power management and consumption, which are all vital for the core's proper functioning.

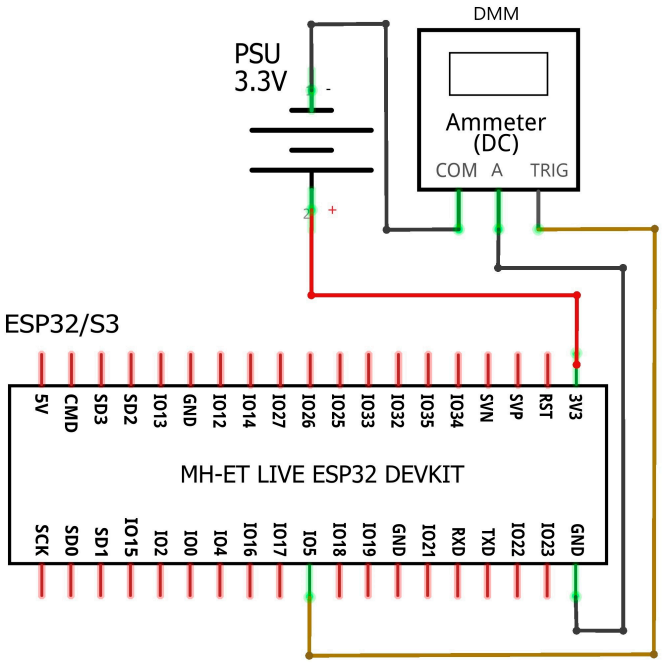
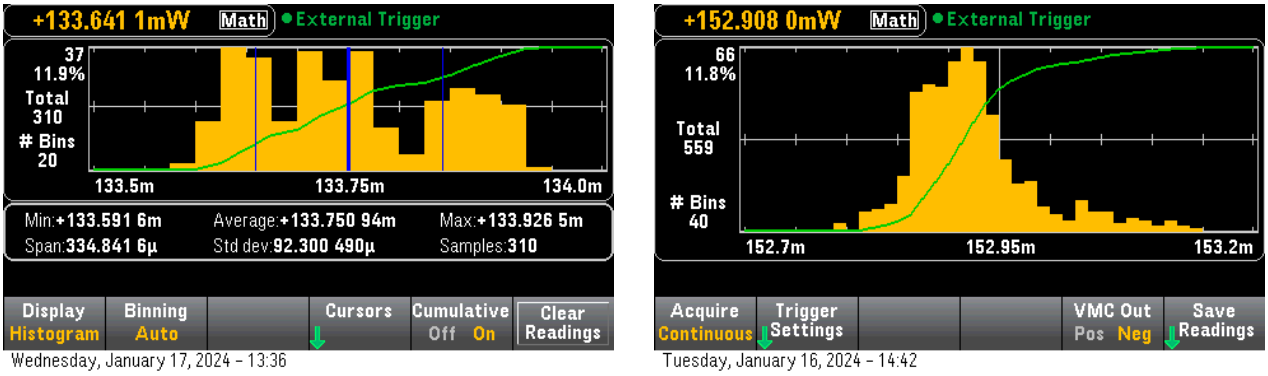



**Figure 1.** Block diagram of device's connections.




(**a**) (**b**)

**Figure 2.** Idle power consumption measurement for: (**a**) ESP32 and (**b**) ESP32-S3.

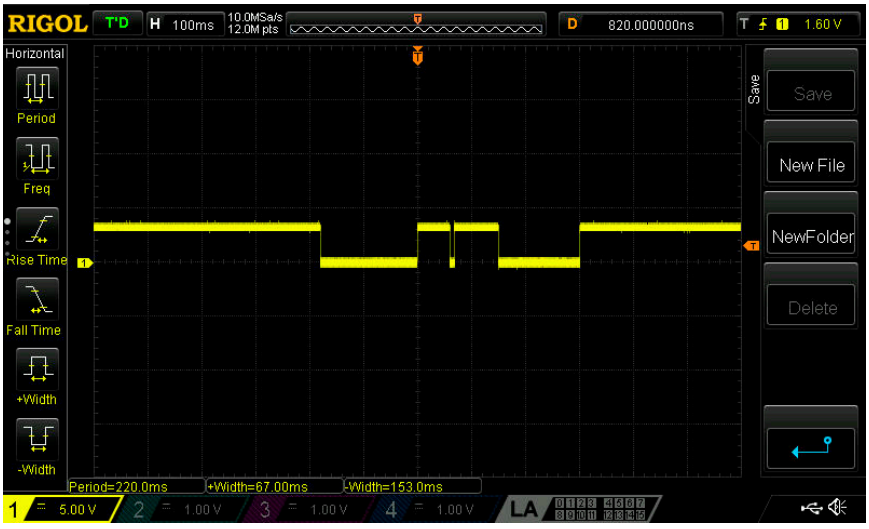



**Figure 3.** Captured trigger signal of color detection model.

## 6. Measurement Results

To draw conclusions about how memory caching influences power efficiency while running ML models, we ran all four available models with various I-Cache and D-Cache combinations in ESP32-S3. The older ESP32 did not have user-definable cache size so it was run in its default state of 32 Kbyte of D-Cache. On ESP32-S3, we ran each model with 16, 32, and 64 Kbytes of D-Cache and 16 and 32 Kbytes of I-Cache. We noticed that increasing the I-Cache from 16 Kbytes to 32 Kbytes had minimal, if any, impact on power efficiency and the acceleration of the ML model calculated by the inference time (or inference latency); thus, we only included the combination of 32 Kbyte I-Cache with 16 Kbyte D-Cache. Cache association also had zero impact on efficiency; thus, we left it at a default of four-way for ESP32-S3. ESP32 was fixed at two-way.

Vectoring influence results were drawn from a comparison of ESP32-S3 running on 32 Kbyte of D-Cache against the ESP32's default state, which has 32 Kbyte of D-Cache. Both inference instances were limited to one core runtime only.

Finally, for human face detection example, which gave us the option to define runtime as one-stage or two-stage inference, with one-stage being a less accurate single inference function and two-stage being a computationally more demanding and accurate model, we ran both types. For the face recognition model, which had the option to run the same model as 16-bit quantized or 8-bit quantized, we took extra measurements for both ESP32-S3 and ESP32.

Some of the examples ran more than one inference per code loop. For each such example, we either present one inference in our results, when all the inferences were of the same load (same power consumption result for each one of them), or we present the sum of our measurements for each cycle when different inferences—at least two of them—were of significantly different computational load. This means that during an example execution loop, there may be multiple trigger events, as depicted in Figure 3. The color detection model in fact runs, in a single run, three different inferences, generating three trigger events that are indicated on the figure as three falling edges. The low-level period after each falling edge is the inference latency time of each inference.

The measurement results include stand-by power, cache usage, inference average power, inference time, and energy used per inference. The latter is the "pure power" [8] result of the total energy consumed during the inference time minus the stand-by power. Below, we summarize our results in four different tables, one for every selected ML model.

The human face detection model (to which the results shown in Table 4 are related) is considered a "medium size/load". From these results, it is apparent that the I-Cache size has minimal effect on both inference time and inference power consumption, while the D-Cache size has a large impact on both. In fact, going from 16 Kbyte to 32 Kbyte increases efficiency by nearly 18%, while increasing cache size to 64 Kbytes halves the power consumption, indicating 53% better power efficiency. Keeping the best cache result of 64 Kbytes of D-Cache and switching to single-stage inference quadruples the power efficiency by dropping the power consumption to 0.461 μWh per inference from 3.843 μWh with 16 Kbyte of D-cache and dual-stage inference.

ESP32 against ESP32-S3 with a similar cache setup but with vectoring support enabled displayed a better power efficiency result for ESP32-S3 by 63%. The power efficiency improvement on ESP32-S3 is apparent for both two-stage and one-stage runtime models, and this is attributed to vectoring support with the SIMD arithmetic instructions [23] enabling the MCU's core to take better advantage of the available resources by running arithmetic instructions concurrently.

The face recognition model is the largest and most computational-resource-demanding model of our experiment; it runs five inferences of similar computational load per loop of code (cycle), and the results (Table 5) are presented for one inference per cycle.

**Table 4.** Human face detection model results *.

| | Stand-By Power (mW) | I-Cache (KB) | D-Cache (KB) | Stages | Inference Average Power (mW) | Inference Time (µS) | Total Energy Used per Inference (µWh) |
|---|---|---|---|---|---|---|---|
| ESP32-S3 | 153.1 | 16 | 16 | 2-Stage | 239.5 | 160,112 | 3.843 |
| ESP32-S3 | 153.1 | 32 | 16 | 2-Stage | 240 | 159,534 | 3.851 |
| ESP32-S3 | 153.1 | 16 | 32 | 2-Stage | 240 | 131,160 | 3.166 |
| ESP32-S3 | 153.1 | 16 | 64 | 2-Stage | 258 | 61,726 | 1.799 |
| ESP32-S3 | 153.1 | 32 | 64 | 2-Stage | 259 | 61,275 | 1.803 |
| ESP32-S3 | 153.1 | 16 | 64 | 1-Stage | 246 | 17,887 | 0.461 |
| ESP32 | 133.75 | 16 | 32 | 2-Stage | 207.183 | 419,556 | 8.558 |
| ESP32 | 133.75 | 16 | 32 | 1-Stage | 208.334 | 153,914 | 3.189 |

* This model may run in one-stage or two-stage inference.

**Table 5.** Face recognition model results *.

| | Stand-By Power (mW) | I-Cache (KB) | D-Cache (KB) | Inference Average Power (mW) | Inference Time (µS) | Total Energy Used per Inference (µWh) | Quantization |
|---|---|---|---|---|---|---|---|
| ESP32-S3 | 153.1 | 16 | 16 | 234.959 | 9,361,445 | 212.866 | S16 |
| ESP32-S3 | 153.1 | 32 | 16 | 231.919 | 11,019,321 | 241.259 | S16 |
| ESP32-S3 | 153.1 | 16 | 32 | 246.452 | 2,510,397 | 65.097 | S16 |
| ESP32-S3 | 153.1 | 16 | 64 | 266.216 | 945,365 | 29.704 | S16 |
| ESP32-S3 | 153.1 | 16 | 16 | 224.562 | 4,726,509 | 93.824 | S8 |
| ESP32-S3 | 153.1 | 16 | 32 | 234.618 | 1,063,099 | 24.073 | S8 |
| ESP32-S3 | 153.1 | 16 | 64 | 259.347 | 320,698 | 9.465 | S8 |
| ESP32 | 133.75 | 16 | 32 | 220.439 | 5,412,302 | 130.330 | S16 |
| ESP32 | 133.75 | 16 | 32 | 219.7 | 13,055,390 | 311.697 | S8 |

* This model may run as 16-bit signed quantized or 8-bit signed quantized.

Increasing the D-Cache from 16 Kbytes to 32 Kbyte improves power efficiency by 69%, while going to 64 Kbytes of D-Cache drops the power consumption per inference to 29.7 µWh from the initial 212.8 µWh for a total gain of 86%.

Using 8-bit quantization further improves power efficiency by up to 95%; however, comparing the initial power consumption of the 8-bit quantized model with the lowest 16 Kbytes of D-cache to 32 and 64 Kbytes, we notice that power efficiency improvements are statistically the same as the 16-bit quantization as cache size increases.

ESP32 with disabled vectoring is consuming almost 13 times more energy (vectoring increases power efficiency by 92%) in the 8-bit quantized model, indicating that SIMD instructions of ESP32-S3 vastly increase performance when lower width (8-bit) instruction payloads are used; however, in the 16-bit quantized model, the difference is much smaller, with ESP32-S3 being just 50% more power efficient than ESP32. We did the same measurement multiple times, with the same result each time. We had the counter-intuitive result that the 16-bit quantized model was not only more power efficient but also ran faster in comparison with the 8-bit model for ESP32.

The color detection model is a small and non-demanding in computational resources model, as it is indicated in the results (Table 6) that increasing the cache size had minimal effect on this model, leading us to conclude that 16 Kbytes is more than enough to achieve the maximum performance for its size.Going from the non-vectoring ESP32 to vectoring-enabled ESP32-S3 in similar cache setup increased power efficiency by almost 21%.

The cat face detection model is the most lightweight model of our experiment. Again, because of its small size, increasing the cache size has no real effect on the power efficiency of the model; however, vectoring has a large impact on power consumption, improving it by almost 85%. This power efficiency improvement is a result of increased data processing efficiency because of ESP32's vectoring instructions. In fact, this capability enables ESP32-

S3 to complete the inference function nearly seven times faster that its ESP32 counterpart (Table 7).

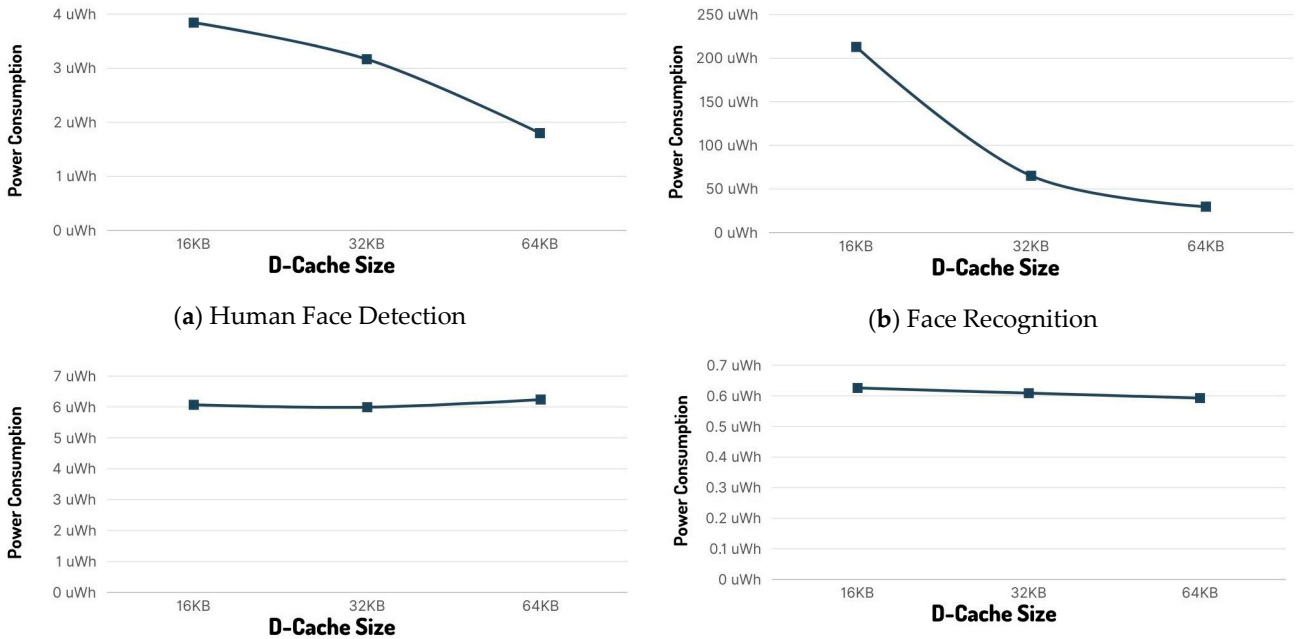
**Table 6.** Color detection model results.

| | Stand-By Power (mW) | I-Cache (KB) | D-Cache (KB) | Inference Average Power (mW) | Inference Time (µS) | Total Energy Used per Inference (µWh) |
|---|---|---|---|---|---|---|
| ESP32-S3 | 153.1 | 16 | 16 | 220 | 326,758 | 6.072 |
| ESP32-S3 | 153.1 | 32 | 16 | 222 | 317,053 | 6.068 |
| ESP32-S3 | 153.1 | 16 | 32 | 222 | 313,181 | 5.994 |
| ESP32-S3 | 153.1 | 16 | 64 | 222 | 326,201 | 6.243 |
| ESP32-S3 | 153.1 | 32 | 64 | 224 | 315,635 | 6.216 |
| ESP32 | 133.75 | 16 | 32 | 188.49 | 497,524 | 7.565 |

**Table 7.** Cat face detection model results.

| | Stand-By Power (mW) | I-Cache (KB) | D-Cache (KB) | Inference Average Power (mW) | Inference Time (µS) | Total Energy Used per Inference (µWh) |
|---|---|---|---|---|---|---|
| ESP32-S3 | 153.1 | 16 | 16 | 252.838 | 22,583 | 0.626 |
| ESP32-S3 | 153.1 | 32 | 16 | 256.159 | 22,096 | 0.633 |
| ESP32-S3 | 153.1 | 16 | 32 | 251.088 | 22,388 | 0.609 |
| ESP32-S3 | 153.1 | 16 | 64 | 259.5 | 20,077 | 0.593 |
| ESP32 | 133.75 | 16 | 32 | 214.226 | 146,592 | 3.277 |

Power consumption per inference diagrams for ESP32-S3 for each ML model are presented below (Figure 4), while the effects of vectoring usage are easily identifiable in Figure 5.




(**a**) Human Face Detection

(**b**) Face Recognition

(**c**) Color Detection

(**d**) Cat Face Detection

**Figure 4.** Power consumption vs. cache size.

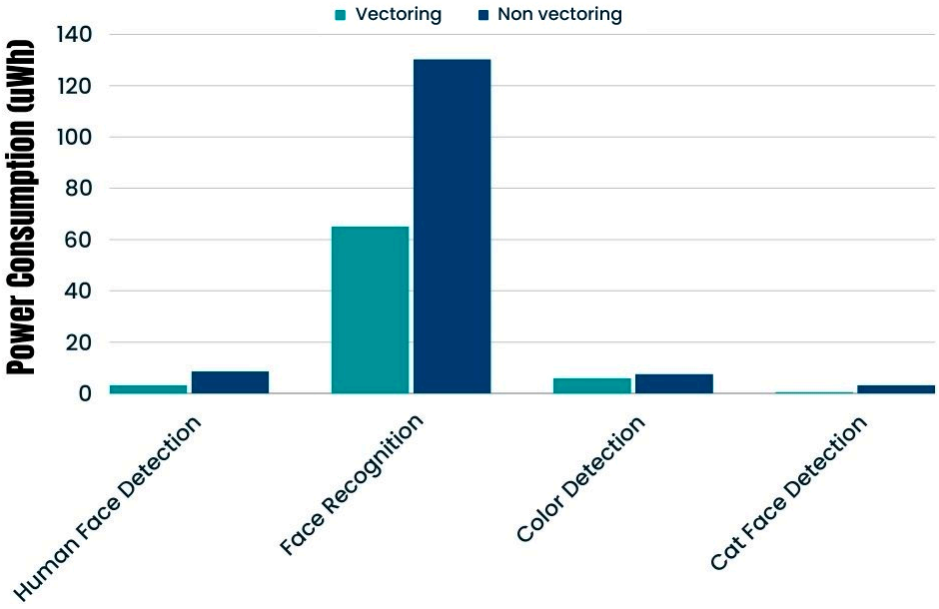



**Figure 5.** Power consumption with and without SIMD/vectoring operation extensions.

## 7. Conclusions and Future Work

In this work, we measured and compared the effects of two hardware acceleration methods on MCUs running ML models. We looked at the usage of cache memory and the usage of SIMD/vectoring technologies.

Both methods are extremely useful in increasing the power efficiency of MCUs in AI runtimes and especially in complex models. On-chip caches show extremely good results and are available on virtually all modern microcontrollers; however, only D-Cache affects ML model efficiency. This is logical, since the ML model runtimes run mostly tensor operations like array multiplications and additions; subsequently, the need for an instruction cache is minimal since the same small set of instructions are always used. On the other hand, trained ML models usually come as packaged data containing large sets of arrays, and thus as tightly packed highly localized large datasets. This formatting is an almost ideal state for a D-Cache to function with a high hit ratio; thus, we expect to observe a large impact on latency and power consumption.

Also, in our measurements, correlation between model complexity/size and cache size is evident, as in smaller models, increasing the cache size did not affect functionality in any way, but in more complex models, the results were impressive with at least one experiment showing one order of magnitude better power efficiency.

On the other hand, gains from vectoring/SIMD enabling were smaller but not insignificant. The size and complexity of each model did not seem to affect the vectoring efficiency, making this method even more useful as a model-agnostic efficiency improvement technique. Of course, the combination of both returns an optimal result for power efficiency, at least for the ESP32 and ESP32-S3 MCUs, which have a peculiar architecture and use the rather slow SPI bus for connectivity between core and external RAM (PSRAM) and flash. It would be extremely interesting to compare these results against a more classic architecture (such as ARM Cortex) that uses much faster AHB and APB busses for connectivity among the core, flash memory, and SRAM.

Since most of the models we used are visual recognition models, it would also be interesting to include necessary peripherals, like a camera, and enable wireless connectivity to draw more conclusions on how a full system may work as a true stand-alone system, at least power-wise. After all optimizations, would the major problem continue to be inference power consumption?

**Author Contributions:** Conceptualization, A.F.; methodology, A.F.; validation, T.O. and V.F.; investigation, A.F.; resources, V.F.; writing—original draft preparation, A.F.; data curation, A.F.; writing—review and editing, A.F., T.O. and V.F.; supervision, T.O. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** All Details are included in the manuscript.

**Acknowledgments:** The present work was undertaken in the context of the "Aerial Surveillance and Measurement of Vessel Emissions Around Ports" (IERAX) supported by the European Union and the Greek Operational Program Attica (project code: ATTP4-0346379).

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Conti, M.; Das, S.K.; Bisdikian, C.; Kumar, M.; Ni, L.M.; Passarella, A.; Roussos, G.; Tröster, G.; Tsudik, G.; Zambonelli, F. Looking ahead in pervasive computing: Challenges and opportunities in the era of cyber–physical convergence. *Pervasive Mob. Comput.* **2012**, *8*, 2–21. [CrossRef]
2. Poslad, S. *Ubiquitous Computing: Smart Devices, Smart Environments and Smart Interaction*; Wiley: Hoboken, NJ, USA, 2009.
3. Russell, S.J.; Norvig, P. *Artificial Intelligence: A Modern Approach*, 4th ed.; Hoboken: Pearson, GA, USA, 2021.
4. Raghubir, S.; Sukhpal, S.G. Edge AI: A survey. *Internet Things Cyber-Phys. Syst.* **2023**, *3*, 71–92.
5. Elsts, A.; McConville, R. Are microcontrollers ready for deep learning-based human activity recognition? *Electronics* **2021**, *10*, 2640. [CrossRef]
6. Shuvo, M.M.H.; Islam, S.K.; Cheng, J.; Morshed, B.I. Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. *Proc. IEEE* **2022**, *111*, 42–91. [CrossRef]
7. Novac, P.E.; Hacene, G.B.; Pegatoquet, A.; Miramond, B.; Gripon, V. Quantization and deployment of deep neural networks on microcontrollers. *Sensors* **2021**, *21*, 2984. [CrossRef] [PubMed]
8. Fanariotis, A.; Orphanoudakis, T.; Kotrotsios, K.; Fotopoulos, V.; Keramidas, G.; Karkazis, P. Power Efficient Machine Learning Models Deployment on Edge IoT Devices. *Sensors* **2023**, *23*, 1595. [CrossRef] [PubMed]
9. Garofalo, A. Flexible Computing Systems for AI Acceleration at the Extreme Edge of the IoT. Ph.D. Thesis, Department of Electrical, Electronic and Information Engineering, University of Bologna, Bologna, Italy, 2022.
10. Desislavov, R.; Martínez-Plumed, F.; Hernández-Orallo, J. Compute and energy consumption trends in deep learning inference. *arXiv* **2021**, arXiv:2109.05472.
11. Alajlan, N.N.; Ibrahim, D.M. TinyML: Enabling of inference deep learning models on ultra-low-power IoT edge devices for AI applications. *Micromachines* **2022**, *13*, 851. [CrossRef] [PubMed]
12. Mazumder, A.N.; Meng, J.; Rashid, H.A.; Kallakuri, U.; Zhang, X.; Seo, J.S.; Mohsenin, T. A survey on the optimization of neural network accelerators for micro-ai on-device inference. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2021**, *11*, 532–547. [CrossRef]
13. Jacob, B.; Wang, D.; Ng, S. *Memory Systems: Cache, DRAM, Disk*; Morgan Kaufmann: Burlington, MA, USA, 2010.
14. Kim, D.; Hida, I.; Fukuda, E.S.; Asai, T.; Motomura, M. Reducing Power and Energy Consumption of Nonvolatile Microcontrollers with Transparent On-Chip Instruction Cache. *Circuits Syst.* **2014**, *5*, 253. [CrossRef]
15. Popovic, M. Improving the Energy Efficiency of a Microcontroller Instruction Fetch Using Tight Loop Cache. Master's Thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2015.
16. Hughes, C.J. *Single-Instruction Multiple-Data Execution*; Springer Nature: Berlin, Germany, 2022.
17. Flynn, M.J.; Rudd, K.W. Parallel architectures. *ACM Comput. Surv.* **1996**, *28*, 67–70. [CrossRef]
18. Keysight. *34465A 6.5 Digit Multimeter, Truevolt DMM*; Keysight: Santa Rosa, CA, USA. Available online: https://www.keysight.com/us/en/product/34465A/digital-multimeter-6-5-digit-truevolt-dmm.html (accessed on 8 February 2024).
19. TTi "EL-R Series Bench DC Power Supply, Linear Regulation | Aim-Tti". Available online: https://www.aimtti.com/product-category/dc-power-supplies/aim-el-rseries (accessed on 8 February 2024).
20. LilyGO TTGO. *LilyGO/TTGO-T8-ESP32: Esp32-I2S-Sdcard-Wav-Player*; GitHub: San Francisco, CA, USA. Available online: https://github.com/LilyGO/TTGO-T8-ESP32 (accessed on 29 February 2024).
21. Espressif. ESP32-S3-DEVKITC-1 v1.1. ESP32-S3. Available online: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html (accessed on 29 February 2024).

22. Espressif. *ESPRESSIF/ESP-DL: Espressif Deep-Learning Library for AIOT Applications*; GitHub: San Francisco, CA, USA; Available online: https://github.com/espressif/esp-dl (accessed on 8 February 2024).
23. Espressif. ESP32-S3 Series—Espressif Systems. Available online: https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf (accessed on 8 February 2024).