UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Smart TinyML Offloading from IoT Devices to Edge Nodes Equipped with Hardware Accelerators

Diploma Thesis

Apostolos Giannousas

Supervisor: Nikolaos Bellas

September 2025

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Smart TinyML Offloading from IoT Devices to Edge Nodes Equipped with Hardware Accelerators

Diploma Thesis

Apostolos Giannousas

Supervisor: Nikolaos Bellas

September 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Έξυπνη Εκφόρτωση TinyML από Συσκευές IoT σε Κόμβους Άκρου Εξοπλισμένους με Επιταχυντές Υλικού

Διπλωματική Εργασία

Απόστολος Γιαννουσάς

Επιβλέπων: Νικόλαος Μπέλλας

Σεπτέμβριος 2025

Approved by the Examination Committee:

Supervisor **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Nikolaos Bellas, for his mentorship and invaluable guidance throughout my studies. His insightful feedback and steady support were instrumental in bringing this thesis to completion.

I am also deeply thankful to Professor Christos Antonopoulos and Professor Spyros Lalis. Their courses, together with those of Professor Bellas, sparked my interest in System Software, Computer Architecture, and High-Performance Computing, and equipped me with the tools to pursue this work.

I owe special thanks to my colleague, Anastassios Nanos, for his sustained collaboration and technical contributions. His assistance with the design, implementation, and troubleshooting of core components significantly strengthened the quality and scope of this thesis.

Finally, I extend my heartfelt gratitude to my family for their unwavering support and encouragement throughout my studies. Their belief in me has been a constant source of motivation.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Apostolos Giannousas

Diploma Thesis

Smart TinyML Offloading from IoT Devices to Edge Nodes Equipped with Hardware Accelerators

Apostolos Giannousas

Abstract

In an era where power efficiency and cost outweigh raw throughput, the deployment of small embedded microcontroller units alongside traditional power-hungry general-purpose computational units (e.g., CPUs) and accelerators (e.g., GPUs) is steadily increasing. Simultaneously, the growing demand in computing deep neural network and machine learning workloads has led to the emergence of TinyML frameworks, which enable tailoring model inference to the tight compute and memory constraints of embedded devices.

However, with the increasing requirements for more accurate and versatile models and tightening real-time targets, handling machine learning workloads solely on embedded devices becomes quite challenging. Consequently, the idea of offloading the whole or part of the computational load to nearby edge devices or cloud servers has occupied a prominent place in modern research.

This thesis proposes a lightweight offloading mechanism for full inference offloading from both generic Linux machines and ESP32 microcontrollers by leveraging the Twirp remote procedure call framework, which provides a minimal wire protocol based on HTTP/1.1 and Protocol Buffers, and the vAccel framework, which provides an abstracted API for using different machine learning frameworks and hardware accelerators. Our mechanism simplifies the process of offloading while maintaining generality and providing scalability.

Keywords:

Microcontrollers, ESP32, Hardware Accelerators, Deep Neural Networks, Machine Learning, TinyML, Computation Offloading, Edge Computing, Cloud Computing, Internet of Things, Remote Procedure Call, vAccel, Twirp, Protocol Buffers, HTTP

Διπλωματική Εργασία

Έξυπνη Εκφόρτωση TinyML από Συσκευές IoT σε Κόμβους Άκρου Εξοπλισμένους με Επιταχυντές Υλικού

Απόστολος Γιαννουσάς

Περίληψη

Σε μια εποχή όπου η ενεργειακή αποδοτικότητα και το κόστος υπερισχύουν του ρυθμού διαμεταγωγής, η ανάπτυξη μικρών ενσωματωμένων μονάδων μικροελεγκτών παράλληλα με τις παραδοσιακές, ενεργοβόρες μονάδες γενικού σκοπού (π.χ. CPUs) και τους επιταχυντές (π.χ. GPUs) αυξάνεται σταθερά. Ταυτόχρονα, η αυξανόμενη ζήτηση για την εκτέλεση φόρτων εργασίας βαθιών νευρωνικών δικτύων και μηχανικής μάθησης έχει οδηγήσει στην εμφάνιση πλαισίων (frameworks) TinyML, τα οποία επιτρέπουν την προσαρμογή του συμπερασμού μοντέλων στα αυστηρά υπολογιστικά και μνημονικά περιορισμένα όρια των ενσωματωμένων συσκευών.

Ωστόσο, με τις ολοένα αυξανόμενες απαιτήσεις για ακριβέστερα και πιο ευέλικτα μοντέλα και με την αυστηροποίηση των στόχων πραγματικού χρόνου, η διαχείριση φόρτων εργασίας μηχανικής μάθησης αποκλειστικά σε ενσωματωμένες συσκευές καθίσταται ιδιαίτερα απαιτητική. Κατά συνέπεια, η ιδέα της αποφόρτισης του συνόλου ή μέρους του υπολογιστικού φορτίου σε γειτονικές συσκευές υπολογιστικού άκρου (edge) ή σε διακομιστές νέφους (cloud) έχει καταλάβει εξέχουσα θέση στη σύγχρονη έρευνα.

Η παρούσα εργασία προτείνει έναν ελαφρύ μηχανισμό αποφόρτισης για πλήρη αποφόρτιση του συμπερασμού τόσο από μηχανές Linux γενικού σκοπού όσο και από μικροελεγκτές ESP32, αξιοποιώντας το πλαίσιο απομακρυσμένων κλήσεων διαδικασιών Twirp, το οποίο παρέχει ένα ελάχιστο wire protocol βασισμένο στο HTTP/1.1 και τα Protocol Buffers (Protobuf), καθώς και το πλαίσιο vAccel, το οποίο παρέχει μια αφαιρετική διεπαφή προγραμματισμού εφαρμογών (API) για τη χρήση διαφορετικών πλαισίων μηχανικής μάθησης και επιταχυντών υλικού. Ο μηχανισμός μας απλοποιεί τη διαδικασία αποφόρτισης, διατηρώντας τη γενικότητα και προσφέροντας κλιμακωσιμότητα.

Λέξεις-κλειδιά:

Μικροελεγκτές, ESP32, Επιταχυντές Υλικού, Βαθιά Νευρωνικά Δίκτυα, Μηχανική Μάθηση,

TinyML, Αποφόρτιση Υπολογιστικού Φορτίου, Υπολογιστική στο Άκρο, Υπολογιστική Νέφους, Διαδίκτυο των Πραγμάτων, Απομακρυσμένη Κλήση Διαδικασιών, vAccel, Twirp, Protocol Buffers, HTTP

Table of contents

List of figures

List of tables

Abbreviations

ABI	Application Binary Interface
AI	Artificial Intelligence
API	Application Programming Interface
CV	Computer Vision
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
ESP-IDF	Espressif's IoT Development Framework
FFI	Foreign Function Interface
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
I/O	Input/Output
IP	Internet Protocol
JSON	JavaScript Object Notation
MCU	Microcontroller Unit
ML	Machine Learning
ROM	Read Only Memory
RPC	Remote Procedure Call
RTOS	Real Time Operating System
SRAM	Static Random Access Memory
TCP	Transmission Control Protocol
TFLM	TensorFlow Lite for Microcontrollers
TinyML	Tiny Machine Learning
TLS	Transport Layer Security
UDP	User Datagram Protocol
VM	Virtual Machine

Chapter 1

Introduction

1.1 Motivation

Microcontroller units (MCUs) enable always-on, near-sensor processing within tight energy and cost budgets, but the memory and compute limits of such platforms restrict the size and accuracy of deployable models. Early work in mobile systems established that moving computation from constrained devices to nearby edge devices or distant cloud servers can reduce energy consumption and end-to-end latency [1, 2].

As deep neural network (DNN) workloads became widespread, inference began to be offered as a service in data centers [3], and device–edge collaboration emerged to shorten the network path and improve responsiveness. Subsequent research focused on exploiting both devices and edge/cloud resources to optimize the execution of inference tasks [4, 5, 6, 7].

Despite the extensive amount of research in this field, emphasis is usually placed on scheduling inference execution, and rarely on the transport mechanisms that move inference jobs. In addition, the need to dynamically switch models and frameworks is often overlooked. These observations motivated us to design the `twirp-vaccel` offloading mechanism, which combines the Twirp Remote Procedure Call (RPC) framework with the vAccel framework. Our objective is to build a lightweight, portable transport mechanism suitable for MCUs, while providing offloading support across multiple frameworks and exploiting hardware accelerators on the remote server.

1.2 Thesis Contribution

The contributions of this thesis can be summarized as follows:

1. We present `twirp-vaccel`, a lightweight and portable mechanism for full inference offloading from constrained devices.
2. We provide a Go server that exposes Twirp-generated service endpoints and executes requests via vAccel, suitable for generic Linux hosts and capable of exploiting available hardware accelerators.
3. We provide a C client for both generic Linux hosts and ESP32 MCUs that uses these endpoints to offload inference.
4. We show that our approach maintains a small client footprint while retaining generality across frameworks and straightforward scalability.

1.3 Thesis Structure

This thesis is organized according to the following structure:

- **Chapter 1: Introduction** - Describes how research motivated this work and outlines its contributions.
- **Chapter 2: Background & Related Work** - Presents the background concepts and related work required to understand the proposed mechanism and evaluate its contribution.
- **Chapter 3: The vAccel Framework** - Provides an overview of the vAccel framework, underlining its unique architecture and design principles.
- **Chapter 4: Proposed Mechanism** - Explains thoroughly the design and implementation of the proposed method and how it operates.
- **Chapter 5: Evaluation** - Presents experiments using the mechanism and interprets the results.
- **Chapter 6: Conclusion** - Summarizes the main findings and outlines directions for future work.

Chapter 2

Background & Related Work

2.1 Offloading Overview

In resource-constrained environments (mobile devices, IoT devices, MCUs), meeting real-time and energy constraints while performing computationally intensive workloads can be quite challenging. Research in classic mobile systems showed that moving expensive computational tasks off the device can dramatically reduce energy and end-to-end latency thanks to systems like MAUI (Fig. 2.1) [1], which enabled fine-grained energy-aware offloading to nearby infrastructure. In the same vein, later work focused on automating partitioning and migration of compute-heavy parts for execution in the cloud, which resulted in systems like CloneCloud (Fig. 2.2) [2].

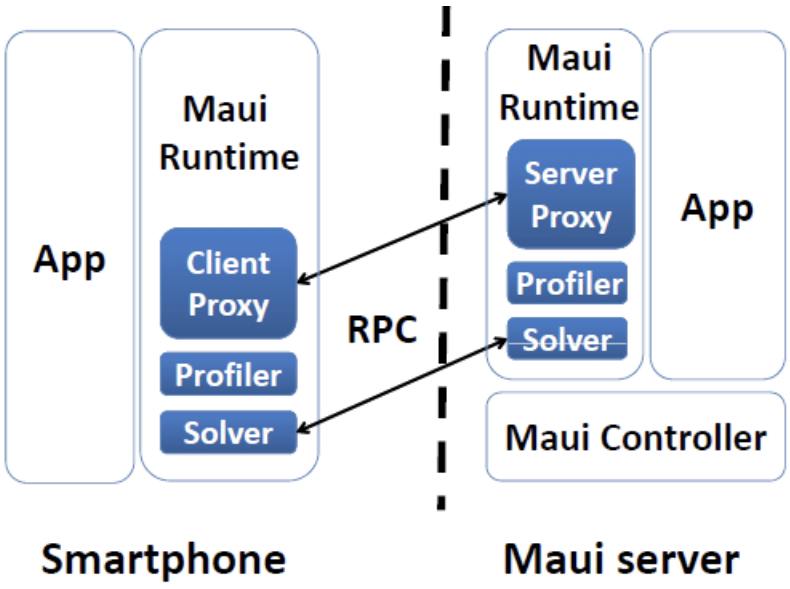

Figure 2.1: High-level view of MAUI offloading system's architecture

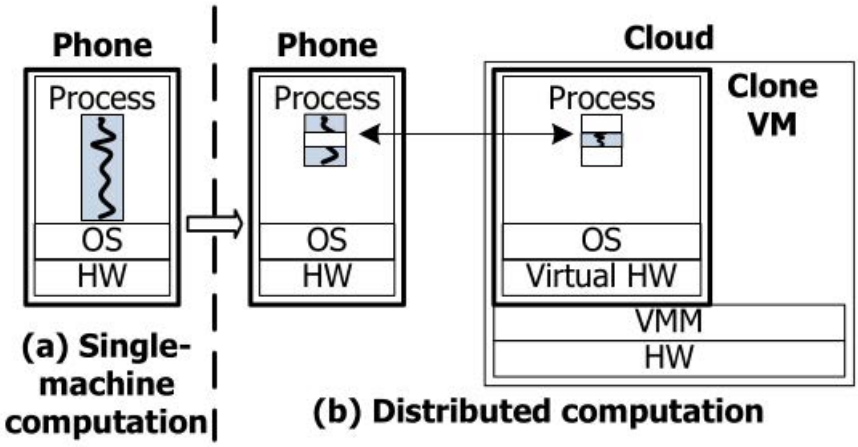

Figure 2.2: CloneCloud system model. CloneCloud transforms a single-machine execution (mobile device computation) into a distributed execution (mobile device and cloud computation) automatically.

As interest in deep learning skyrocketed, researchers identified the need to provide DNN inference as a service, which led to the first attempts to fully offload DNN forward passes to cloud servers that return the results to the user [3]. Soon after, more sophisticated tools such as the Neurosurgeon scheduler (Fig. 2.3) were designed to partition DNN computation between the mobile device and the datacenter, thus contributing to a significant reduction in latency and energy consumption [4].

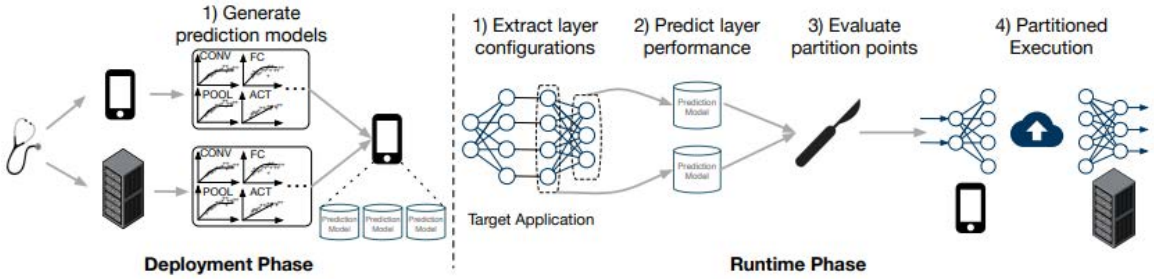

Figure 2.3: Overview of Neurosurgeon. At deployment, Neurosurgeon generates prediction models for each layer type. During runtime, Neurosurgeon predicts each layer's latency/energy cost based on the layer's type and configuration, and selects the best partition point based on various dynamic factors

To put everything into perspective, we can distinguish the following five major DNN offloading categories by examining the findings in [5, 6, 7]:

- **On-device (*no offload*):** The whole inference is executed locally on the end device.

This constitutes the baseline against which offloading schemes are evaluated

- **Full offloading (*data offloading*):** The end device is only responsible for acquiring and optionally processing the input data while the inference is computed solely by an edge/cloud node

- **Partial offloading / split computing (*feature offloading*):** The model is partitioned. The computations of some early layers are performed on the end device and their intermediate features are transmitted to an edge/cloud node to complete the remaining layers.

- **Multi-tier collaboration:** A generalization of partial offloading in which partitioning and placement are coordinated across the end device, edge, and cloud to balance proximity and elasticity.

- **Device-device collaboration:** Multiple nearby devices collaborate and share inference work directly (peer-to-peer) or through a local coordinator.

2.2 Data Serialization and Transmission

Having outlined why and when offloading is applied, we are going to delve into the mechanisms that allow formatting and transmitting the offloading requests on the wire. We are going to categorize the various protocols that work together to construct such mechanisms and point out the most prominent ones.

2.2.1 Serialization protocols

Serialization is the process of translating a data structure into a sequence of bits for storage and transmission, while enabling reconstruction of the original data structure (also called deserialization). Serialization protocols are comprised of a set of rules and formats that define certain serialization/deserialization utilities. Their goal is to facilitate the exchange of information without depending on the machine or the language.

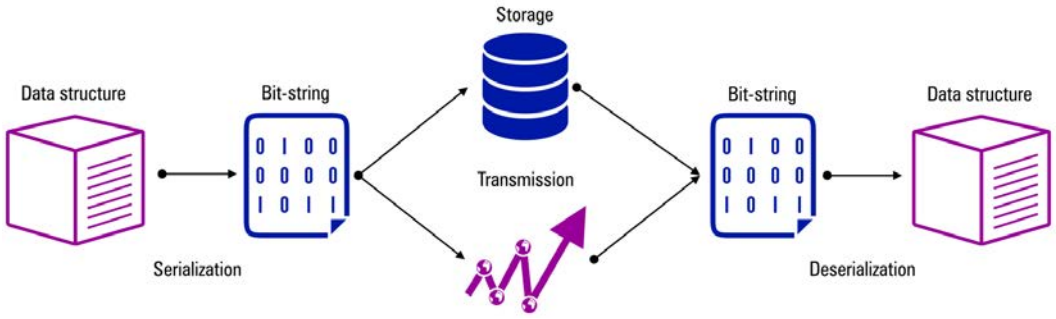

Figure 2.4: The process of translating a data structure to a bit-string is called serialization. The process of translating a bit-string back to its original data structure is called deserialization.

A widely used taxonomy classifies those protocols along two orthogonal axes: (i) **textual vs. binary** representation (human-readable vs. compact machine-oriented encodings); and (ii) **schema-driven vs. schema-less** (based on whether a formal schema describing the data structure is required) [8].

Protocol / Format	**Representation**	**Schema Category**
JSON	Textual	Schema-less
XML	Textual	Schema-less
YAML	Textual	Schema-less
Amazon Ion (text)	Textual	Schema-less
Protocol Buffers	Binary	Schema-driven
Apache Thrift (binary)	Binary	Schema-driven
Cap'n Proto	Binary	Schema-driven
FlatBuffers	Binary	Schema-driven
Apache Avro	Binary	Schema-driven
ASN.1	Binary	Schema-driven
Apache Arrow IPC/Feather	Binary	Schema-driven
MessagePack	Binary	Schema-less
CBOR	Binary	Schema-less
BSON	Binary	Schema-less
Amazon Ion (binary)	Binary	Schema-less

Table 2.1: Popular serialization protocols categorized by representation and schema requirements

The most notable example is Protocol Buffers (Protobuf) by Google [9], a language-neutral, platform-neutral, binary, schema-driven serialization format. Developers define data structures in *.proto files* using message declarations, and the Protobuf compiler generates serialization and deserialization code for a variety of languages.

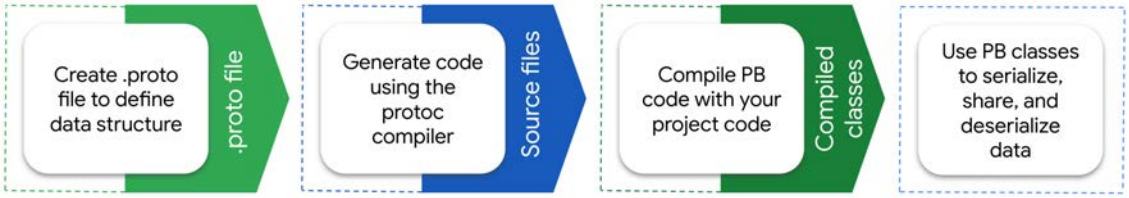

Figure 2.5: Protocol Buffers workflow

2.2.2 Wire Protocols

There are many definitions for wire protocols, and the term is often (imprecisely) used interchangeably with serialization protocols. Essentially, a wire protocol specifies the ”on-the-wire” message format and framing exchanged between a client and a service. It typically uses a serialization format (e.g., Protobuf) and may define how messages map onto an application protocol (e.g., HTTP) or a transport protocol (TCP/UDP). In summary, a wire protocol combines message structure, encoding, and framing to enable interoperable communication between remote components [10]

2.2.3 Remote Procedure Call

RPC is a programming model that exposes service operations as if they were local function calls. The client-side stub encodes the function parameters into a message and sends it to a remote server. There, the server-side stub receives and decodes the message, invokes the target operation, and, upon completion, encodes the results into a response that is sent back to the client. Finally, the client decodes the response and returns the results to the caller as if executed locally [11].

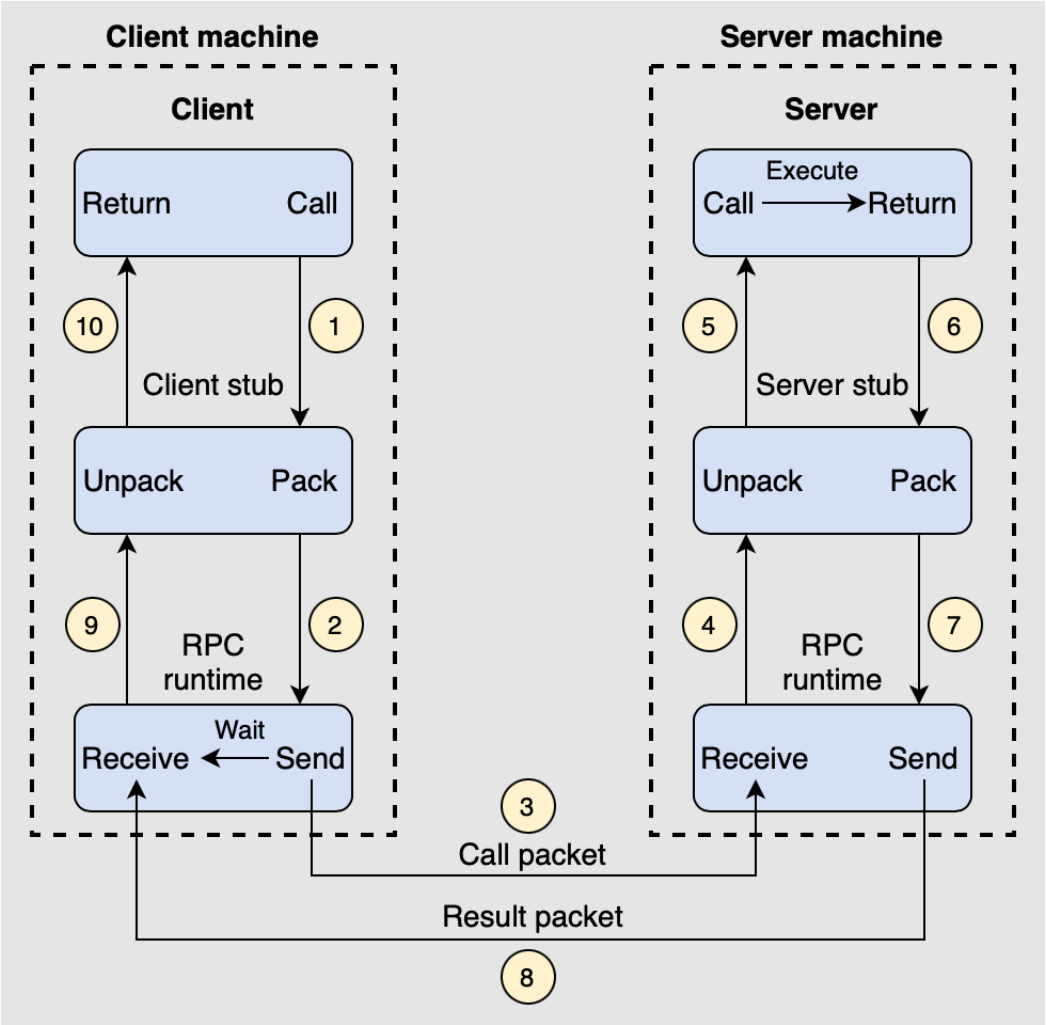

Figure 2.6: Implementation of RPC model

There are many RPC frameworks that follow this paradigm which are often described as wire protocols, because they bundle structure formatting, serialization, and a concrete message framing. The most notable ones are:

- **gRPC:** A modern, high-performance RPC framework that uses **Protobuf** by default and defines a wire format carried over **HTTP/2**. It supports unary and bidirectional streaming RPCs. Channel security is typically provided via **TLS**, and **gRPC** is widely adopted in general-purpose environments [12].
- **ttrpc:** A lightweight alternative originating in the containerd [13] ecosystem. It retains proto-defined services and Protobuf messages, but replaces **gRPC's HTTP/2** layer with a minimal custom framing to reduce overhead. It is commonly used over **Unix domain sockets** or **TCP** while security is supplied by the underlying channel (e.g. **mTLS**) rather than a built-in **HTTP/2/TLS** stack [14].

- **Twirp:** A simple RPC framework that maps proto services to **HTTP/1.1** endpoints with bodies encoded as **Protobuf** or **JSON**. It emphasizes small, easy-to-integrate clients and clear error semantics. Advanced features such as streaming are intentionally not supported due to its orientation towards embedded devices with very limited resources [15].

2.3 Microcontroller Units

A MCU is a small computer on a single integrated circuit which usually includes a processor, non-volatile memory such as Flash and Read-Only-Memory (ROM), Static/Dynamic Random Access Memory (SRAM/DRAM) and various peripherals (timers, GPIO, input/output (I/O)). In contrast to traditional general-purpose systems, MCUs were designed to tackle specific tinier tasks within larger systems, thus aiding them to meet tight cost, power and energy constraints [16].

Many MCU deployments run either bare-metal firmware or a real-time operating system (RTOS). An RTOS provides lightweight task scheduling, timing services, and synchronization primitives with an emphasis on predictability and determinism for time-sensitive workloads [17]. Popular examples include FreeRTOS, which offers a minimal kernel and libraries tailored to MCUs [17, 18].

Among the most popular MCUs are those based on Espressif's (ESP32) family [19], which combine integrated Wi-Fi and Bluetooth with multiple CPU core options (e.g., Xtensa and RISC-V variants) and a broad set of peripherals. Across the family, common on-chip features include SPI/I^2C/UART buses, ADCs and DACs, PWM/timers, capacitive touch inputs, SD/MMC interfaces, and GPIO with flexible interrupt capabilities. Many ESP32 parts also offer low-power modes, an ultra-low-power coprocessor for sensor polling, and security features such as secure boot and flash encryption, making them suitable for intricate applications.

Espressif's IoT Development Framework (ESP-IDF) [20] provides a production-grade software stack on top of FreeRTOS, including device drivers, the lwIP TCP/IP stack, TLS (mbedTLS) support, Wi-Fi and Bluetooth middleware, peripherals HALs, and common IoT services (e.g., provisioning, OTA updates, logging, NVS/flash storage). The build system relies on the *idf.py* tool which manages CMake/Ninja for configuring and building the project,

as well as the *esptool.py* tool for flashing it.

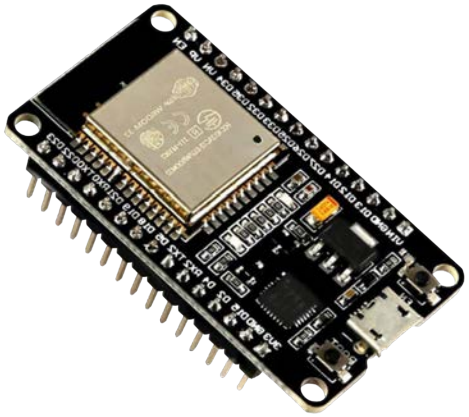

Figure 2.7: An ESP32 microcontroller device

2.4 TinyML

Tiny machine learning (TinyML) refers to executing machine learning (ML) inference on resource-constrained embedded devices, and especially low-power MCUs. TinyML is presented as the convergence of embedded systems and classic ML. This encompasses ecosystems with compact model designing techniques, toolchains and deployment workflows tailored for the memory constraints of MCUs [21]. The spurt of TinyML frameworks is attributed to the following benefits:

- **Reduced latency:** Sensor data can be consumed immediately for local inference, avoiding end-to-end delays from streaming to edge/cloud services.
- **Energy efficiency:** Carefully designed models and runtimes can operate in the milliwatt to sub-milliwatt range for certain workloads, enabling battery-powered, always-on scenarios.
- **Lower bandwidth use:** When acquisition, preprocessing, and inference occur locally, far less data must be transmitted upstream, which reduces connectivity requirements.
- **Privacy and data locality:** Keeping raw signals on device, limits exposure of sensitive data to malicious agents.

Operating at MCU scale forces specific design patterns in both models and software: small footprints in Flash and SRAM, aggressive quantization (commonly 8-bit integer) to reduce memory and compute and minimal dependencies so that inference fits alongside application code and I/O stacks. Toolchains therefore prioritize reproducibility and portability, and they often trade generality for determinism and a small surface area [22].

2.4.1 TensorFlow Lite for Microcontrollers

TensorFlow Lite for Microcontrollers (TFLM) is a runtime specifically engineered to bring the TensorFlow Lite execution model to TinyML systems. Its design targets MCU realities: kilobytes to a few hundred kilobytes of RAM, limited non-volatile storage, and the absence of operating system services such as dynamic memory allocators or file systems. The TFLM interpreter adopts a static allocation strategy, precomputing memory requirements and using a single contiguous tensor arena so that inference can proceed without heap allocation at runtime. The core is written in portable C/C++ with a small and stable application binary interface (ABI)[23].

From a deployment perspective, models are converted to TensorFlow Lite FlatBuffers and typically quantized to 8-bit integer to shrink both parameters and activations. At build time, an operator resolver selects only the operations required by the model, minimizing binary size. At runtime, the micro-interpreter performs a simple, stepwise invocation over the graph using the preallocated arena, which enables predictable latency and eases integration with real-time tasks. The TFLM design deliberately avoids large framework dependencies so that the same application code can target a range of MCU architectures.

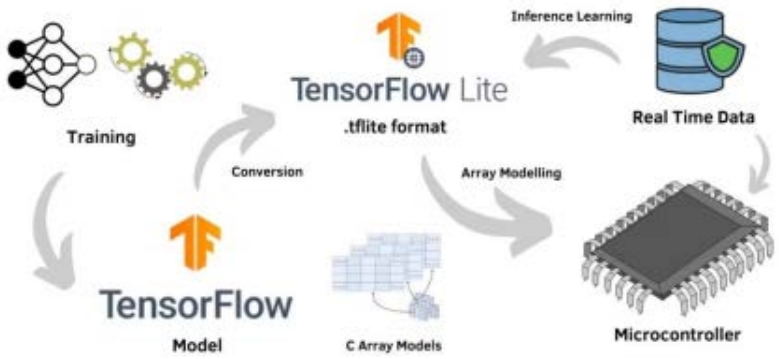

Figure 2.8: TensorFlow Lite for Microcontrollers Toolchain

Chapter 3

The vAccel Framework

3.1 Introduction

vAccel is a lightweight, modular framework designed to allow workloads in virtualized / isolated environments to benefit from available hardware accelerators. Fundamentally, vAccel decouples an application's logic from the specifics of the various hardware platforms and system configurations by providing a generic hardware-agnostic interface that can be mapped to accelerator-specific implementations. This approach enables transparent offloading of compute-intensive operations while also maintaining isolation [24].

3.2 Design Principles

vAccel is designed with several core principles in mind:

- **Portability**
  Applications built with vAccel can be deployed across systems with differing accelerator types and configurations without source changes or recompilation; this is reinforced by native support for `amd64`, `arm`, and `arm64` (including cross-compiled and heterogeneous clusters) and by several language bindings that ease integration into existing ML/AI pipelines and cloud-native workflows.

- **Security**
  vAccel supports hardware-accelerated execution within strong isolation boundaries using KVM-based VMs (e.g., QEMU, Firecracker, Cloud Hypervisor), allowing the same

application binary to run inside a VM while safely invoking host-side acceleration.

- **Modularity**

  A plugin-based architecture decouples applications from specific accelerators so hardware-specific implementations can be developed and maintained independently.

- **Efficiency**

  Communication between guest and host uses low-overhead paths such as VirtIO and RPC over VSOCK, TCP, or UNIX sockets, minimizing framework-induced latency and achieving near-native performance. Benchmarks typically show overhead below 5% relative to direct accelerator use.

- **Scalability**

  Native integration with Kubernetes enables large-scale, multi-tenant deployments of acceleration-enabled workloads, while an end-to-end CI pipeline continuously validates core components and language bindings across multiple backends, host/guest combinations, and transport modes for correctness and performance.

3.3 Architecture

The modular architecture of vAccel is comprised of a core accelerator-agnostic runtime and multiple backend plugins as demonstrated in Fig. 3.1

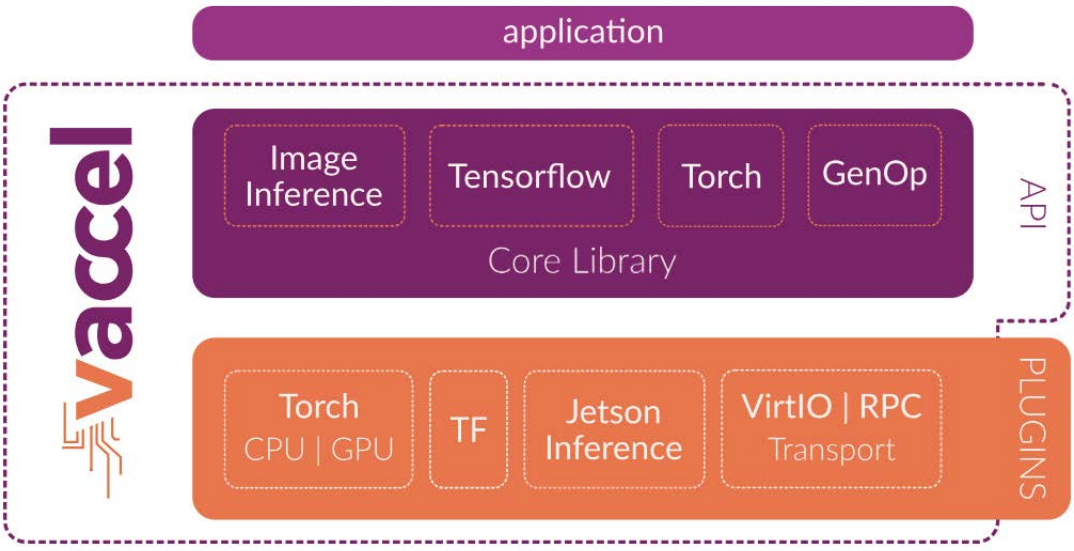

Figure 3.1: vAccel software stack

3.3.1 Core Runtime

The vAccel core runtime [24] is responsible for managing the interaction between applications and backend plugins in the following way:

- Session Management: Every operation and resource belonging to the core runtime is associated with a specific session. The runtime is responsible for creating, updating and destroying vAccel sessions.
- Resource Management: The core runtime can encapsulate binary data such as ML models and shared objects in vAccel resource structures. Their existence is crucial for offloading as they enable seamless management and utilization without moving data around pointlessly. Furthermore, it serves as a further abstraction for applications that can treat all types of binary data as a single entity and pass them to vAccel operations.
- Plugin Coordination: The core runtime's main task is to act as the middle layer that will search through its plugins and find a suitable implementation of a requested operation.

3.3.2 Plugins

The vAccel plugin model [24] provides modularity and isolation. All offload logic is packaged as dynamically loadable shared objects that conform to the vAccel Plugin ABI. This keeps application code agnostic to backend specifics while allowing plugins to be loaded on demand. The various plugins can be divided in the following three categories:

- **Bundled plugins:** they are part of the core vAccel installation and are designed to help with development and debugging
  - **NoOp plugin:** serves as a package of dummy implementations for vAccel operations and is meant for verifying the correctness of a vAccel setup.
  - **Exec plugin:** utilities for calling functions from external libraries
  - **MBench plugin:** used for simulating workloads for performance benchmarking
- **Acceleration plugins:** they execute acceleration functions using host-side libraries, drivers, or frameworks (e.g., CUDA, OpenCL, neural accelerators). They must implement the functions described by the vAccel application programming interface (API),

register their capabilities during initialization, and handle requests from the core library through the dispatch layer. Each plugin is isolated from others and loaded only when required, which reduces resource usage and improves security.

  - **Tensorflow plugin:** implements basic Tensorflow and Tensorflow Lite support for vAccel operations.
  - **Torch plugin:** implements basic PyTorch C++ API (LibTorch) support for vAccel operations.
  - **TVM plugin:** implements image inference vAccel operations with TVM

- **Transport plugins:** they implement the guest-host communication layer. They are responsible for framing, sending, and receiving requests, so that the core runtime remains decoupled from specific communication mechanisms.

  - **RPC plugin:** offers transport based on ttrpc for vaccel operations and supports VSOCK, UNIX and TCP sockets
  - **VirtIO plugin:** offers transport based on VirtIO for vaccel operations. It uses the virtio-accel [25] kernel module and is a more lightweight alternative to VSOCK.

3.3.3 Configuration and Execution Flow

vAccel application can be configured at runtime using a variety of environment variables for defining preferences such as plugin usage, higher logging level for debugging, profiling utilities etc. as depicted in table 3.1.

Table 3.1: vAccel environment variables

Variable Name	Description	Expected Values	Default
VACCEL_PLUGINS	Path(s) or filename(s) of backend plugin(s)	Absolute path, filename, or `:`-separated list	—
VACCEL_LOG_LEVEL	Controls log verbosity	1, 2, 3, 4	1
VACCEL_LOG_FILE	Filename of a log file	Filename (without extension)	—
VACCEL_PROFILING_ENABLED	Enables or disables profiling	1, 0	0
VACCEL_VERSION_IGNORE	If set, ignores plugin/lib version mismatches	1, 0	0
VACCEL_BOOTSTRAP_ENABLED	Enables or disables library auto-bootstrap	1, 0	1
VACCEL_CLEANUP_ENABLED	Enables or disables library auto-cleanup	1, 0	1

By initializing a vAccel session and trying to execute a certain vAccel operation, the application triggers an execution flow similar to that in Fig. 3.2

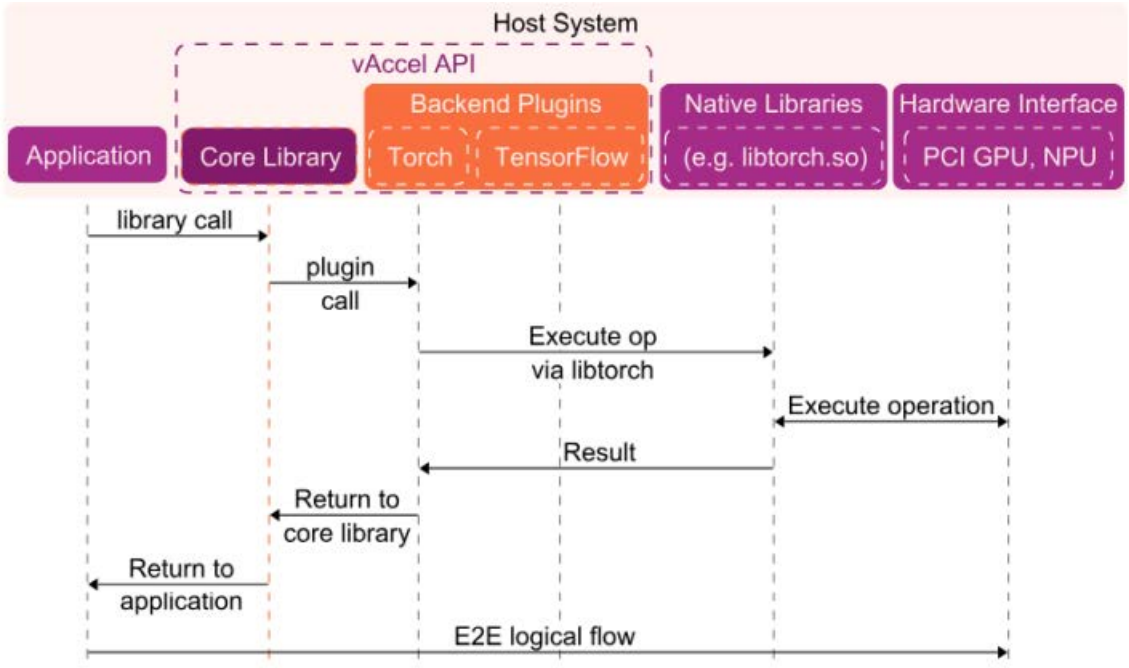

Figure 3.2: vAccel execution flow

3.4 Supported Operations

The vAccel API [24] offers a wide variety of high-level operations, including linear algebra and image processing tools, ML model management, and generic function executors. The most vital ones, especially for enabling seamless ML application offloading, are the following:

- **Image Inference:** An assortment of C functions that take a session handle and an image buffer as input. The tasks they can perform include image classification, segmentation, object detection, pose estimation, and monocular depth. For instance, the `vaccel_image_classification` function can use either a standard computer vision (CV) algorithm or an ML model, depending on the backend plugin, and return a classification tag.

- **Generic Executor:** vAccel provides the `vaccel_exec` function through which the application can execute any function symbol from a given shared object library. This is

crucial for exploiting vAccel for offloading, as the application can dynamically offload new operations without having to modify the framework.

- **Generic Operation:** In the same vein as generic execution, vAccel offers the `vaccel_genop` function, which can execute any vAccel-supported operation, broadening the generality of applications using the framework.

- **TensorFlow Operations:** By creating a session and attaching the model to a vAccel resource and the input to the appropriate vAccel tensor containers, the user can trigger the TensorFlow C runtime using vAccel's functions for loading the SavedModel resource (`vaccel_tf_model_load`), running the inference (`vaccel_tf_model_run`), and unloading the model (`vaccel_tf_model_unload`).

- **TensorFlow Lite Operations:** Similarly to TensorFlow, an equivalent collection of functions is provided for loading the TFLite model (`vaccel_tflite_model_load`), running the inference (`vaccel_tflite_model_run`), and unloading the model (`vaccel_tflite_model_delete`).

- **Torch Operations:** As with TensorFlow/TensorFlow Lite, the API offers functions for loading a Torch model (`vaccel_torch_model_load`) and running the inference (`vaccel_torch_model_run`).

Note that all these operations, along with session and resource management tools, are supported by the offloading transport mechanism proposed in this thesis and presented in the next chapter. Although this does not enable fine-grained acceleration, the generality of the offloading approach creates significant potential for smart offloading mechanisms and policies while also maintaining scalability.

3.5 Language Bindings

The core vAccel API is written in pure C, so applications written in C/C++ can use it natively by including the `vaccel.h` header and linking against `libvaccel` [26]. However, the framework also offers the following language bindings:

- **Python**

  Python bindings wrap the vAccel C API and expose native functions that mirror the C

interfaces [27] (e.g., `vaccel_image_classification(...)`), enabling scripts to invoke vAccel operations directly. The bindings are a work in progress and currently cover a subset of the API.

- **Rust**

  The `vaccel-rust` [28] project offers Rust support, using Rust's Foreign Function Interface (FFI) and providing the `vaccel-bindings` crate. Based on that, the `vaccel-rust` project offers the `vaccel-rpc-proto`, `vaccel-rpc-agent` and `vaccel-rpc-agent` crates which implement ttrpc-based transport for vAccel operations.

- **Go**

  There is also a corresponding `vaccel-go` [29] project that uses `cgo` to call the vAccel C API. Although less mature than the Rust project, it was recently expanded to support the work proposed in this thesis. Currently, it covers most operations, with additional documentation planned for future releases. A Go application can use it by importing `github.com/nubificus/vaccel-go/vaccel`, provided the vAccel headers and libraries are available to the build and link process.

Chapter 4

Proposed Mechanism

In Chapter 3, we introduced the vAccel framework [24], which includes a comprehensive set of ML operations and generic executors. Offloading functions provided by the vAccel API is therefore an ideal basis for a hardware-agnostic and ML–framework-independent mechanism. Although vAccel already includes an RPC implementation based on ttrpc [14], its `vaccel_rpc_client` creates the following limitations:

- **Client-side dependency on vAccel:** The client is coupled to the vAccel API, implying that the framework (or at least a substantial subset) must be ported to the device. This can be prohibitive for extremely constrained platforms such as the ESP32.

- **Language and protocol constraints:** The available ttrpc client libraries target Go and Rust. Many MCU SDKs/IDFs do not support Rust, and there is no widely adopted C client, thus porting to other environments would require reimplementing ttrpc's custom framing/wire protocol. Although there are community efforts to bring Rust to ESP32 [30], they are not yet mature enough for projects of this scope.

These limitations motivated the `twirp-vaccel` mechanism, which retains `vaccel-rust's` Protobuf schemas for vAccel operations but eliminates client-side dependencies on vAccel, avoids language-specific runtimes, and replaces custom RPC framing with a minimal HTTP/1.1–based wire protocol (Twirp).

4.1 Serialization Schemas

The RPC interface is defined by a set of Protobuf schema files. We list their roles here, but the full source appears in Appendix 7.

- `vaccel.proto` → common status/error types (e.g., `Status`, `Error`, `ErrorType`).
- `empty.proto` → empty message wrapper.
- `session.proto` → vAccel session management data structures (`CreateRequest`, `CreateResponse`, `UpdateRequest`, `DestroyRequest`).
- `resource.proto` → vAccel resource management data structures (e.g. `RegisterRequest`, `UnregisterRequest`, `RegisterResponse`).
- `image.proto` → vAccel image inference request/response.
- `genop.proto` → vAccel generic operation interface messages (`Arg`, `Request/Response`).
- `tf.proto` → vAccel TensorFlow data types, tensor and node structures along with model management messages (e.g., `DataType`, `Tensor`, `Node`, `ModelRunRequest`).
- `tflite.proto` → vAccel TensorFlow Lite data types, tensor and model handling structures (e.g. `DataType`, `Tensor`, `ModelLoadRequest`).
- `torch.proto` → vAccel Torch data types, tensor and model handling structures (e.g. `DataType`, `Tensor`, `ModelLoadRequest`.
- `service.proto` → Imports all above packages and declares the RPC methods (e.g., TensorFlow/TFLite/Torch model load/run, resource registration, image classification, generic operation). This file is only needed on the server side for Twirp server/client stub generation.

The `service.proto` schema defines the `Vaccel service` with the following RPC:

```
syntax = "proto3";


import "session.proto";
import "resource.proto";
```

```
import "image.proto";
import "vaccel.proto";
import "genop.proto";
import "empty.proto";
import "tf.proto";
import "tflite.proto";
import "torch.proto";

package vaccel.service;
option go_package =
    "twirp-vaccel/go-server/generated/vaccel/service;service";


// VaccelService defines the RPC service interface
service Vaccel {
  // Session handling
  rpc CreateSession(vaccel.session.CreateRequest) returns
    (vaccel.session.CreateResponse);
  rpc UpdateSession(vaccel.session.UpdateRequest) returns
    (vaccel.Error);
  rpc DestroySession(vaccel.session.DestroyRequest) returns
    (vaccel.Error);

  // TensorFlow
  rpc TensorflowModelLoad(vaccel.tf.ModelLoadRequest) returns
    (vaccel.tf.ModelLoadResponse);
  rpc TensorflowModelUnload(vaccel.tf.ModelUnloadRequest) returns
    (vaccel.tf.ModelUnloadResponse);
  rpc TensorflowModelRun(vaccel.tf.ModelRunRequest) returns
    (vaccel.tf.ModelRunResponse);

  // TensorFlow Lite
  rpc TensorflowLiteModelLoad(vaccel.tflite.ModelLoadRequest)
    returns (vaccel.empty.Empty);
```

```
32    rpc TensorflowLiteModelUnload(vaccel.tflite.ModelUnloadRequest)
       returns (vaccel.empty.Empty);
33    rpc TensorflowLiteModelRun(vaccel.tflite.ModelRunRequest)
       returns (vaccel.tflite.ModelRunResponse);
34
35    // Torch
36    rpc TorchModelLoad(vaccel.torch.ModelLoadRequest) returns
       (vaccel.empty.Empty);
37    rpc TorchModelRun(vaccel.torch.ModelRunRequest) returns
       (vaccel.torch.ModelRunResponse);
38
39    // vAccel resource handling
40    rpc RegisterResource(vaccel.resource.RegisterRequest) returns
       (vaccel.resource.RegisterResponse);
41    rpc UnregisterResource(vaccel.resource.UnregisterRequest)
       returns (vaccel.empty.Empty);
42
43    rpc ImageClassification(vaccel.image.Request) returns
       (vaccel.image.Response);
44
45    rpc Genop(vaccel.genop.Request) returns (vaccel.genop.Response);
46 }
```

Listing 4.1: `service.proto` - RPC service definition

4.2 Code Generation Pipeline

4.2.1 Go/Twirp Server

In order to generate the appropriate files from the proto schemas on the server side, the Protobuf compiler (`protoc`) is required along with `protoc-gen-go` and `protoc-gen-twirp` plugins that are installed as go packages. Compiling with (`protoc`) and the flags `-go_out` and `-twirp_out`, we obtain the go files below:

- `*.pb.go`: Go files for each proto file which define their data structures

- `service.twirp.go`: This file contains serialization/deserialization handlers, interfaces with empty implementations for all RPC methods and client/server stubs.

Finally, the `server.go` file, which uses the generated Protobuf and Twirp files, provides implementations for the Twirp server stub interfaces based on the `vaccel-go` bindings. By running this server file, the server stub listens on port 8080 and exposes HTTP endpoints with the form "`http://<SERVER_IP>:8080/twirp/vaccel.service.Vaccel/<METHOD>`". A client can serialize a request message and send an HTTP POST request to the endpoint corresponding to its desired method.

The generated server stub is wrapped up in a `New<Service>Server` constructor which in our case has the following signature:

```
func NewVaccelServer(
    svc Vaccel,
    opts ...interface{},
) TwirpServer
```

Essentially, this constructor wraps your interface implementations as a TwirpServer, which is a http.Handler with a few extra features. It can be mounted like any other handler (e.g. with `http.ListenAndServe`) and then your RPC server will be up and running.

Note that the Twirp Go file also includes similar constructors for Protobuf and JSON client stubs, but we deliberately ignore them since we are going to replace them with our custom Protobuf C client.

4.2.2 Linux & ESP32 C Clients

Both C-based clients use `protobuf-c` to generate C code for all schema files except `service.proto`, producing `*.pb-c.h/.c` with message data structures and serialization/deserialization functions. In Linux, code generation is driven by `protoc` with the `protoc-gen-c` plugin (invoked by `-c_out`), as provided by `libprotobuf-c-dev`. In ESP32, the same functionality is provided by the ESP-IDF `protobuf-c` component.

The generated Protobuf-C sources are consumed by `vaccel_rpc.c`, which exposes high-level wrappers of the RPC methods to user applications. Each wrapper (i) maps plain user arguments into the appropriate Protobuf message, (ii) serializes the request, (iii) issues an HTTP POST to the server, and (iv) deserializes the response.

To keep `vaccel_rpc.c` generic, we abide by two principles:

- **Schema fidelity:** Wrappers accept only plain C types or the message types defined in the Protobuf schemas and not vAccel-specific structs as in vAccel's ttrpc client. Therefore, the API remains vAccel-agnostic.
- **Transport abstraction:** All high-level RPC wrappers delegate the actual network call to a single transport hook, `twirp_request()`. This function encapsulates the HTTP/1.1 POST to the Twirp endpoint, handles serialization/deserialization buffers, and (optionally) records profiling statistics in a caller-provided region.

Consequently, we provide two variants of `twirp_client_util.c`: a Linux implementation using `libcurl`, and an ESP32 implementation using the ESP-IDF `esp_http_client` component. The `twirp_request()` function signature is:

Listing 4.2: Transport hook used by all RPC wrappers

```
int twirp_request(const char *service_method,
                  struct prof_region *transport_plus_server_stats,
                  const void *request_data,
                  size_t request_size,
                  struct response_data *response);
```

4.3 Software Execution Stacks

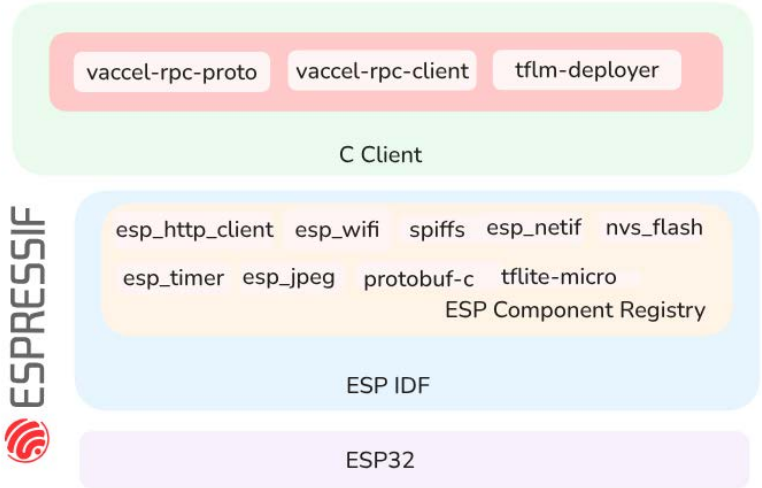

Figure 4.1: Twirp vAccel Go server execution stack

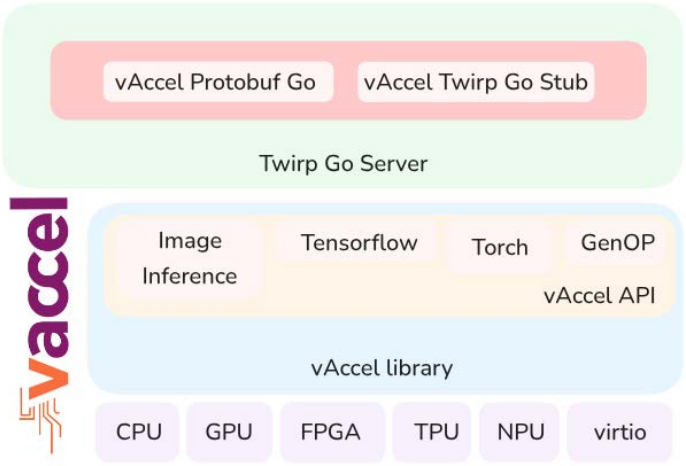

Figure 4.2: Twirp vAccel ESP32 C client execution stack

Chapter 5

Evaluation

5.1 Experimental Setup

5.1.1 Platforms

Experiments were run on a desktop host with an AMD Ryzen 5 2600 (6C/12T) CPU and an NVIDIA GeForce RTX 2060 GPU, running Ubuntu 22.04.4 (kernel 6.8.0-83). The MCU client is an ESP32-S3-DevKitC-1-N16R8 board (16 MB Flash, 8 MB PSRAM).

5.1.2 Software

Host-side frameworks and tools were: Python 3.10.12, TensorFlow 2.17.0 (with TFLite runtime), PyTorch 2.6.0, Go 1.25.1, and `protoc` 3.12.4 with the necessary Go and Twirp plugins. The ESP32 builds used ESP-IDF v6.0-dev and the `xtensa-esp-elf-gcc` 15.1.0 toolchain. Offloaded execution was invoked via a Twirp (HTTP/1.1 + Protocol Buffers) service on the host.

5.1.3 Model and workload

All measurements use the same custom ResNet-10 model trained on Fashion-MNIST [31]. We evaluate three inference stacks:

1. TensorFlow, TFLite, and Torch on the host under both native and vAccel execution.

2. TFLM locally on the ESP32.

3. Full inference offload from the ESP32 to CPU/GPU servers via Twirp→host frameworks.

5.1.4 Metrics and methodology

We report end-to-end inference latency in milliseconds. For offloaded runs, latency is additionally broken down into client, transport, server, and framework execution on the host. Initial warm-up executions preceded the measurements which were the result of averaging a considerable amount of iterations.

5.2 Results

From Figs. 5.1, 5.2 and Tables 5.1, 5.2, **CPU** deltas from stock to vAccel are tiny: Torch $-0.393\,\mathrm{ms}$ (-3.8%), TensorFlow $+0.611\,\mathrm{ms}$ ($+7.3\%$), TFLite $+0.171\,\mathrm{ms}$ ($+3.3\%$). On **GPU**, absolute differences are still small around $+0.277\,\mathrm{ms}$ (Torch) and $+1.078\,\mathrm{ms}$ (TensorFlow)—but percentages look large due to sub-millisecond baselines. For larger models, these largely fixed costs are amortized and the relative overhead shrinks.

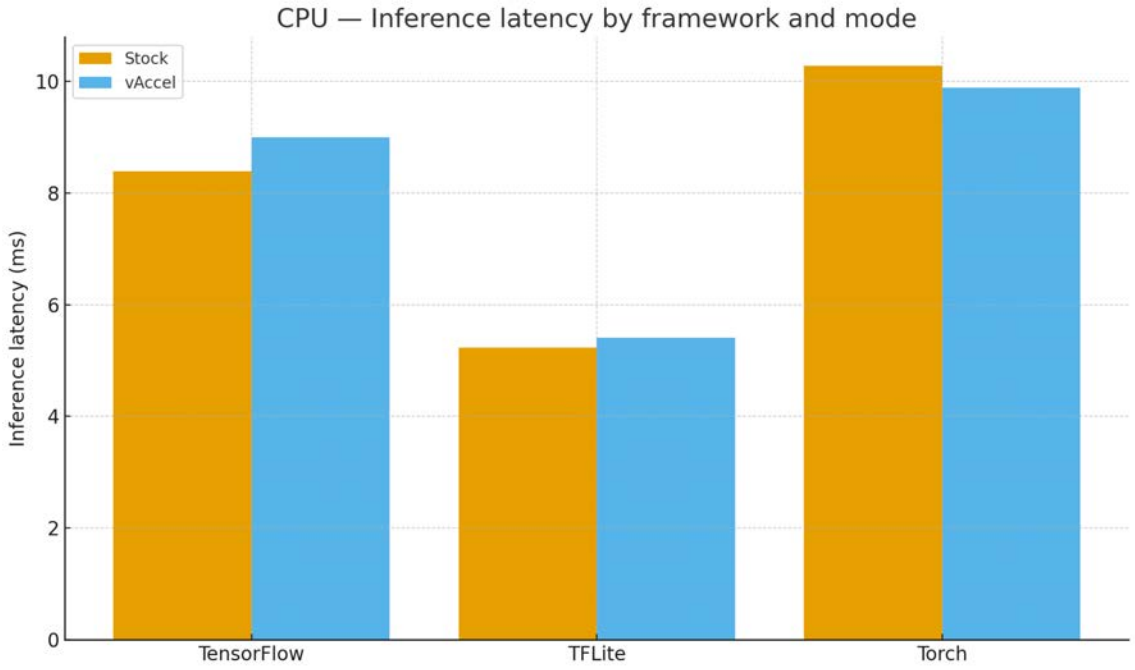

Figure 5.1: Inference latency on AMD Ryzen 5: TensorFlow, TFLite, and Torch under stock (native) execution and vAccel.

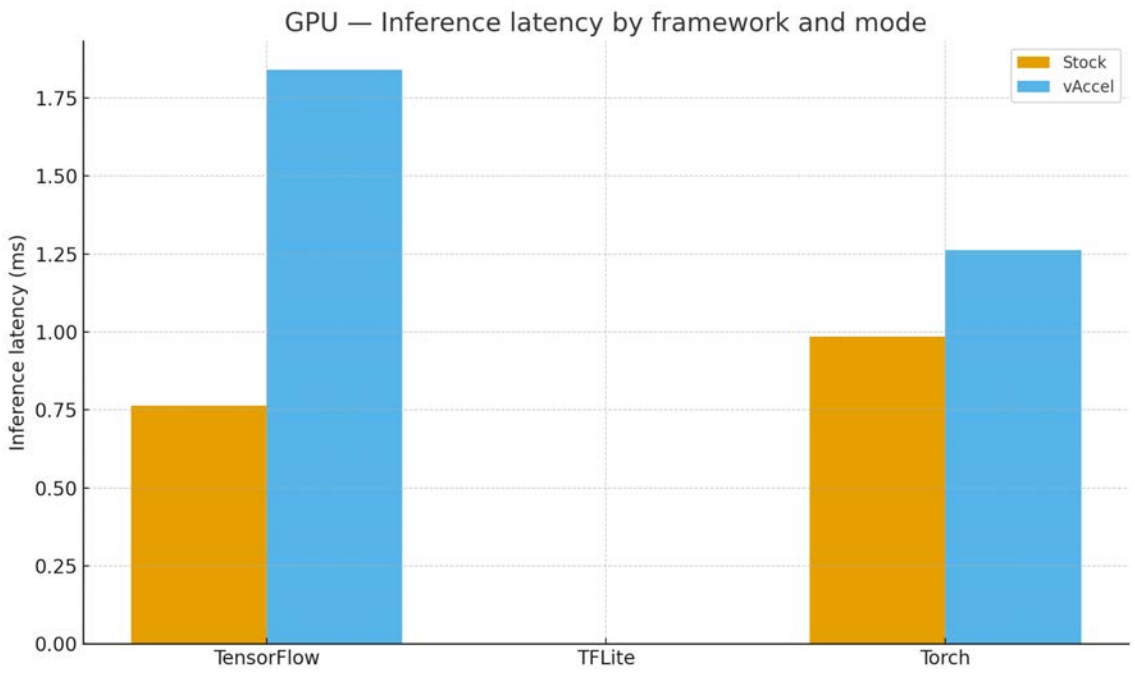

Figure 5.2: Inference latency on NVIDIA GeForce RTX 2060: TensorFlow and Torch under stock (native) execution and vAccel.

Framework	Stock	vAccel
resnet10-torch	10.2807	9.8874
resnet10-tf	8.3878	8.9989
resnet10-tflite	5.2347	5.4055

Table 5.1: CPU inference latency (ms)

Framework	Stock	vAccel
resnet10-torch	0.9852	1.2624
resnet10-tf	0.7630	1.8414

Table 5.2: GPU inference latency (ms)

From Fig. 5.3 and Table 5.3, offloading cuts ESP32 latency from seconds (local TFLM: ~6.3s) to ~90–112,ms, depending on server and framework. Even though the offloaded path traverses multiple software layers (HTTP/1.1, Protobuf, Twirp stubs, vAccel framework, ML framework), that overhead is tiny compared to on-device compute. This makes the mecha-

nism practical for real-time use and paves the way for selecting a larger model when needed to improve accuracy, while still meeting latency targets.

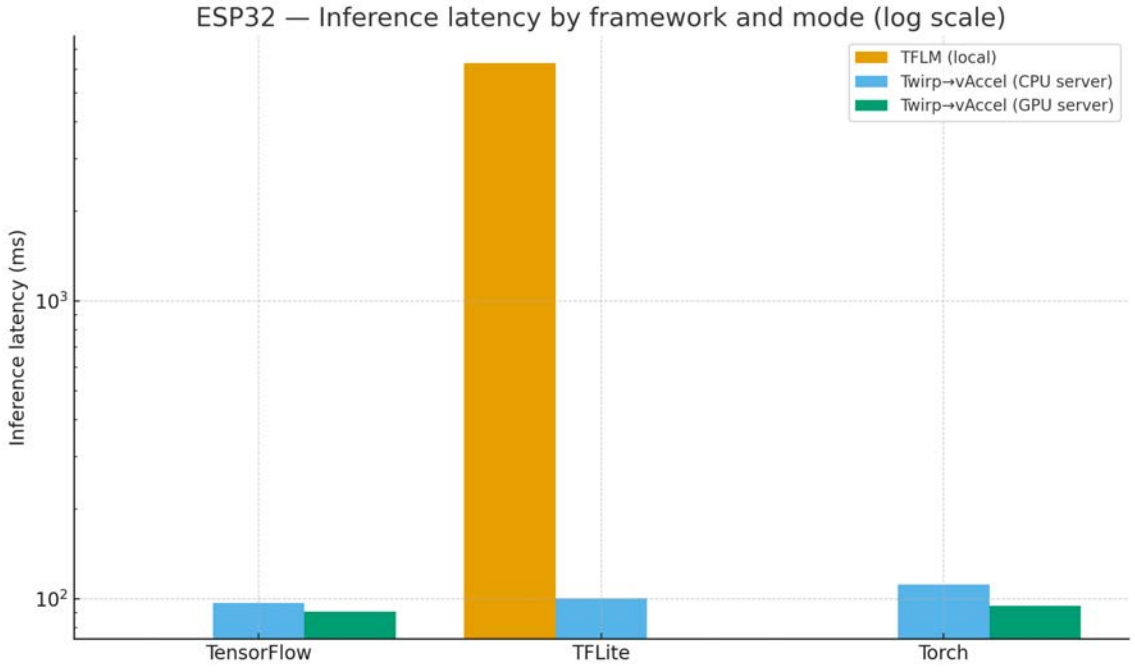

Figure 5.3: ESP32-S3-DevKitC-1-N16R8 inference latency: TFLM (local) and Twirp→vAccel offloading (TensorFlow, TFLite, Torch) to AMD Ryzen 5 / NVIDIA GeForce RTX 2060.

Framework	**TFLM (local)**	**Twirp→vAccel (CPU)**	**Twirp→vAccel (GPU)**
resnet10-torch	–	111.59327	94.74776
resnet10-tf	–	96.75123	90.7364
resnet10-tflite	6287.03345	100.61403	–

Table 5.3: ESP32 inference latency (ms)

Finally, the component breakdown in Fig. 5.4 and Table 5.4 shows that client and server overheads are small and consistent across frameworks and hosts. The client side is slightly higher than the server, primarily due to request serialization and response deserialization. In general, the logistics of the client and the server add only minimal cost, and, as expected, the transport time dominates the end-to-end latency.

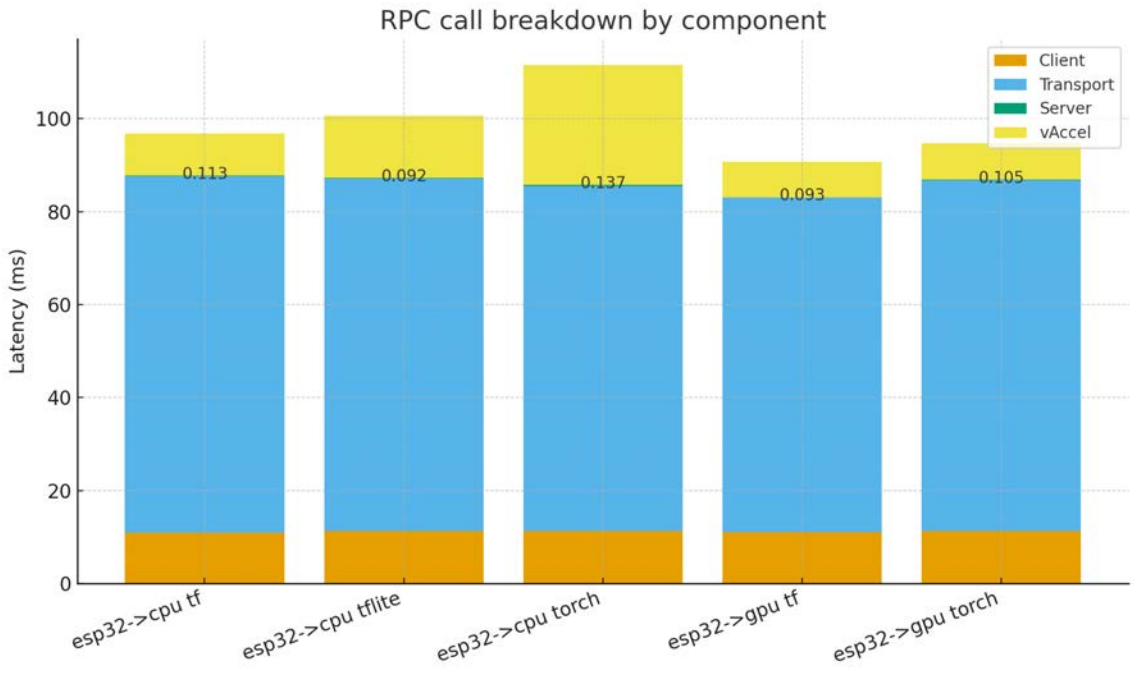

Figure 5.4: ESP32→Edge RPC latency breakdown by component (Client, Transport, Server, vAccel) for TensorFlow, TFLite, and Torch on CPU/GPU servers.

Run	Client	Transport	Server	vAccel	Total
esp32→cpu tf	10.97028	76.69195229	0.11274880	8.97624891	96.75123
esp32→cpu tflite	11.29599	75.82097121	0.09156625	13.40550254	100.61403
esp32→cpu torch	11.23891	74.33658818	0.13650002	25.88127180	111.59327
esp32→gpu tf	11.05669	71.95307490	0.09278736	7.63384774	90.73640
esp32→gpu torch	11.29236	75.48900537	0.10450563	7.86188900	94.74776

Table 5.4: ESP32 → Edge RPC breakdown (ms)

Chapter 6

Conclusion

6.1 Summary

In this thesis, we combined the vAccel framework with the Twirp wire protocol to build a general yet simple full, on-demand offloading mechanism. While the approach applies to standard Linux clients, our focus was on microcontrollers, and specifically ESP32, where resource constraints make local inference hard for larger or more accurate models. Unlike much related work, we prioritized simplifying the transport path (HTTP/1.1 + Protocol Buffers, no heavyweight client runtime) and keeping the client API vAccel-agnostic, while still allowing the application to switch models and frameworks dynamically. On the server, vAccel lets us leverage heterogeneous accelerators (CPU/GPU) behind a stable RPC interface, offering capabilities often left outside the offloading discussion.

Our evaluation with a ResNet-10 (FMNIST), shows that offloading reduces ESP32 latency from seconds to approximately 100ms, depending on framework and host, enabling real-time use and leaving headroom to pick larger models when needed. The client and server overheads are small and stable, but as expected, transport time dominates the offloaded path. All in all, the mechanism delivers a substantial performance boost for MCUs while adding minimal additional complexity and overhead, meeting the original goals of practicality, portability, and scalability.

6.2 Future Work

Our work can be extended in several directions:

1. **Reduce transport latency:** Systematically study the sensitivity of end-to-end latency to the network stack and hardware (e.g. TCP congestion control, HTTP keep-alive, CPU frequency, interrupt coalescing), and apply targeted tuning to reduce transport time.

2. **Partial offloading mechanism:** Integrate model partitioning to support feature offloading, instead of just full offloading. This can increase throughput and better exploit end–edge resources, especially for bandwidth-heavy inputs.

3. **Client-server RPC optimization:** Optimize Twirp usage via persistent connections and request pipelining. Explore client-side and server-side batching to cut down per-call costs, or even an entirely new custom framing protocol tailored to our needs.

4. **Adaptive offloading policy:** Design a runtime policy that chooses between local, partial, and full offload based on live telemetry data.

5. **Energy and resource profiling:** Add power measurements on the MCU and host, memory/flash footprints, and network bytes transferred to characterize latency, energy and accuracy trade-offs.

Bibliography

[1] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *ACM MobiSys 2010*. Association for Computing Machinery, Inc., June 2010.

[2] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 301–314, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 27–40, New York, NY, USA, 2015. Association for Computing Machinery. Also appears in *SIGARCH Computer Architecture News* 43(3S), June 2015.

[4] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 615–629, New York, NY, USA, 2017. Association for Computing Machinery. Also appears in *SIGARCH Computer Architecture News* 45(1), April 2017, pp. 615–629.

[5] Wei-Qing Ren, Yu-Ben Qu, Chao Dong, Yu-Qian Jing, Hao Sun, Qi-Hui Wu, and Song Guo. A survey on collaborative dnn inference for edge intelligence. *Machine Intelligence Research*, 20(3):370–395, May 2023.

[6] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. Split computing and early exiting for deep learning applications: Survey and research challenges. *ACM Computing Surveys*, 55(5):1–30, December 2022.

[7] Di Xu, Xiang He, Tonghua Su, and Zhongjie Wang. A survey on deep neural network partition over cloud, edge and end devices, 2023.

[8] Juan Cruz Viotti and Mital Kinderkhedia. A survey of json-compatible binary serialization specifications, 2022.

[9] Protocol buffers. `https://protobuf.dev`.

[10] Understanding Wire Protocol `https://risingwave.com/blog/understanding-wire-protocol/`.

[11] Remote Procedure Call (RPC) API Explained `https://medium.com/codenx/remote-procedure-call-rpc-api-explained-3d4a494ff28b`.

[12] The gRPC Framework `https://grpc.io/`.

[13] An industry-standard container runtime with an emphasis on simplicity, robustness and portability `https://containerd.io/`.

[14] The ttrpc Framework `https://deepwiki.com/containerd/ttrpc`.

[15] The Twirp Framework `https://twitchtv.github.io/twirp/`.

[16] What is a Microcontroller (MCU)? `https://www.techtarget.com/iotagenda/definition/microcontroller`.

[17] What is a real-time operating system (RTOS)? `https://www.ibm.com/think/topics/real-time-operating-system`.

[18] FreeRTOS `https://www.freertos.org/`.

[19] Espressif `https://www.espressif.com/`.

[20] Espressif IoT Development Framework (ESP-IDF) `https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html`.

[21] EdgeAI Foundation `https://www.edgeaifoundation.org/`.

[22] P. Warden and D. Situnayake. Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers, 2019. https://www.oreilly.com/library/view/tinyml/9781492052036/.

[23] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. Tensorflow lite micro: Embedded machine learning for tinyml systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 800–811, 2021.

[24] vAccel Framework Docs https://docs.vaccel.org/latest/.

[25] virtio-accel Kernel Module https://github.com/nubificus/virtio-accel.

[26] vAccel Framework https://github.com/nubificus/vaccel.

[27] vAccel Python Bindings https://github.com/nubificus/vaccel-python.

[28] vAccel Rust Bindings https://github.com/nubificus/vaccel-rust.

[29] vAccel Go Bindings and RPC components https://github.com/nubificus/vaccel-go.

[30] Raw Rust bindings for the ESP IDF SDK https://github.com/esp-rs/esp-idf-sys.

[31] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

APPENDICES

Appendix A

Protobuf Schemas

```
syntax = "proto3";

package vaccel;
option go_package =
    "twirp-vaccel/go-server/generated/vaccel;vaccel";

message Status {
  uint32 code = 3;
  string message = 4;
}

enum ErrorType {
  FFI = 0;
  FFI_WITH_STATUS = 1;
  INVALID_ARGUMENT = 2;
  UNINITIALIZED = 3;
  OUT_OF_BOUNDS = 4;
  EMPTY_VALUE = 5;
  CONVERSION_FAILED = 6;
}

message Error {
  ErrorType type = 1;
```

```
  uint32 ffi_error = 2; // optional
  Status status = 3; // optional
}
```

Listing A.1: `vaccel.proto` - Core vAccel message types

```
syntax = "proto3";

package vaccel.image;
option go_package =
    "twirp-vaccel/go-server/generated/vaccel/image;image";

message Request {
  int64 session_id = 1;
  bytes image = 2;
}

message Response {
  bytes tags = 1;
}
```

Listing A.2: `image.proto` - Image classification messages

```
syntax = "proto3";

package vaccel.genop;
option go_package =
    "twirp-vaccel/go-server/generated/vaccel/genop;genop";

message Arg {
  uint32 argtype = 1;
  uint32 size = 2;
  bytes buf = 3;
  uint32 parts = 4;
  uint32 part_no = 5;
}
```

```
13
14 message Request {
15   int64 session_id = 1;
16
17   repeated Arg read_args = 2;
18   repeated Arg write_args = 3;
19 }
20
21 message Response {
22   repeated Arg write_args = 1;
23 }
```

Listing A.3: `genop.proto` - Generic operation messages

```
1  syntax = "proto3";
2
3  package vaccel.torch;
4  option go_package =
       "twirp-vaccel/go-server/generated/vaccel/torch;torch";
5
6
7  enum DataType {
8    UNUSED = 0; // For vAccel value compatibility
9    BYTE = 1;
10   CHAR = 2;
11   SHORT = 3;
12   INT = 4;
13   LONG = 5;
14   HALF = 6;
15   FLOAT = 7;
16 }
17
18 message Tensor {
19   bytes data = 1;
20   repeated int64 dims = 2;
```

```
  DataType type = 3;
}

message ModelLoadRequest {
  int64 session_id = 1;
  int64 model_id = 2;
}

message ModelRunRequest {
  int64 session_id = 1;
  int64 model_id = 2;

  bytes run_options = 3; // optional
  repeated Tensor in_tensors = 4;
  uint64 nr_out_tensors = 5;
}

message ModelRunResponse {
  repeated Tensor out_tensors = 1;
}
```

Listing A.4: `torch.proto` - Torch model messages

```
syntax = "proto3";

package vaccel.resource;
option go_package =
    "twirp-vaccel/go-server/generated/vaccel/resource;resource";

enum BlobType {
  FILE = 0;
  BUFFER = 1;
  MAPPED = 2;
}

```

```
message Blob {
  BlobType type_ = 1;
  string name = 2;
  bytes data = 3;
  uint32 size = 4;
}

enum ResourceType {
  LIB = 0;
  DATA = 1;
  MODEL = 2;
}

message RegisterRequest {
  repeated string paths = 1;
  repeated Blob blobs = 2;
  ResourceType resource_type = 3;
  int64 resource_id = 4;
  int64 session_id = 5;
}

message RegisterResponse {
  int64 resource_id = 1;
}

message UnregisterRequest {
  int64 resource_id = 1;
  int64 session_id = 2;
}
```

Listing A.5: `resource.proto` - Resource registration messages

```
syntax = "proto3";

package vaccel.session;
```

```
option go_package =
    "twirp-vaccel/go-server/generated/vaccel/session;session";

message CreateRequest {
  uint32 flags = 1;
}

message CreateResponse {
  int64 session_id = 1;
}

message UpdateRequest {
  int64 session_id = 1;
  uint32 flags = 2;
}

message DestroyRequest {
  int64 session_id = 1;
}
```

Listing A.6: `session.proto` - Session management messages

```
syntax = "proto3";

import "vaccel.proto";

package vaccel.tflite;
option go_package =
    "twirp-vaccel/go-server/generated/vaccel/tflite;tflite";

enum DataType {
  UNUSED = 0; // For vAccel value compatibility
  NOTYPE = 1;
  FLOAT32 = 2;
  INT32 = 3;
```

```
  UINT8 = 4;
  INT64 = 5;
  STRING = 6;
  BOOL = 7;
  INT16 = 8;
  COMPLEX64 = 9;
  INT8 = 10;
  FLOAT16 = 11;
  FLOAT64 = 12;
  COMPLEX128 = 13;
  UINT64 = 14;
  RESOURCE = 15;
  VARIANT = 16;
  UINT32 = 17;
  UINT16 = 18;
  INT4 = 19;
}

message Tensor {
  bytes data = 1;
  repeated int32 dims = 2;
  DataType type = 3;
}

message ModelRunRequest {
  int64 session_id = 1;
  int64 model_id = 2;

  repeated Tensor in_tensors = 3;
  uint64 nr_out_tensors = 4;
}

message ModelLoadRequest {
  int64 session_id = 1;
```

```
  int64 model_id = 2;
}

message ModelUnloadRequest {
  int64 session_id = 1;
  int64 model_id = 2;
}

message ModelRunResponse {
  repeated Tensor out_tensors = 1;
  vaccel.Status status = 2; // optional
}
```

Listing A.7: `tflite.proto` - TensorFlow Lite model messages

```
syntax = "proto3";

import "vaccel.proto";

package vaccel.tf;
option go_package =
   "twirp-vaccel/go-server/generated/vaccel/tf;tf";

enum DataType {
  UNUSED = 0; // For vAccel value compatibility
  FLOAT = 1;
  DOUBLE = 2;
  INT32 = 3;
  UINT8 = 4;
  INT16 = 5;
  INT8 = 6;
  STRING = 7;
  COMPLEX = 8;
  INT64 = 9;
  BOOL = 10;
```

```
  QINT8 = 11;
  QUINT8 = 12;
  QINT32 = 13;
  BFLOAT16 = 14;
  QINT16 = 15;
  QUINT16 = 16;
  UINT16 = 17;
  COMPLEX128 = 18;
  HALF = 19;
  RESOURCE = 20;
  VARIANT = 21;
  UINT32 = 22;
  UINT64 = 23;
}

message Tensor {
  bytes data = 1;
  repeated int64 dims = 2;
  DataType type = 3;
}

message Node {
  string name = 1;
  int32 id = 2;
}

message ModelLoadRequest {
  int64 session_id = 1;
  int64 model_id = 2;
}

message ModelLoadResponse {
  bytes graph_def = 1;
  vaccel.Status status = 2; // optional
```

```
}

message ModelUnloadRequest {
  int64 session_id = 1;
  int64 model_id = 2;
}

message ModelUnloadResponse {
  vaccel.Status status = 1; // optional
}

message ModelRunRequest {
  int64 session_id = 1;
  int64 model_id = 2;

  bytes run_options = 3; // optional
  repeated Node in_nodes = 4;
  repeated Node out_nodes = 5;
  repeated Tensor in_tensors = 6;
}

message ModelRunResponse {
  repeated Tensor out_tensors = 1;
  vaccel.Status status = 2; // optional
}
```

Listing A.8: `tf.proto` - TensorFlow model messages

```
syntax = "proto3";

package vaccel.empty;
option go_package =
   "twirp-vaccel/go-server/generated/vaccel/empty;empty";

message Empty {}
```

Listing A.9: `empty.proto` - Empty placeholder message

Appendix B

Twirp Request Utilities

Listing B.1: `twirp_client_util.c` - Implementation for Linux

```c
#include "twirp_client_util.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>

static size_t write_callback(void *contents, size_t size, size_t
    nmemb, void *userp) {
  size_t real_size = size * nmemb;
  struct response_data *response = (struct response_data *)userp;

  response->data = realloc(response->data, response->size +
   real_size + 1);
  if (!response->data) {
    fprintf(stderr, "Not enough memory (realloc returned NULL)\n");
    return 0;
  }

  memcpy(&(response->data[response->size]), contents, real_size);
  response->size += real_size;
  response->data[response->size] = 0;

```

```
21   return real_size;
22 }
23
24 int twirp_request(const char *service_method,
25         struct prof_region *transport_plus_server_stats,
26         const void *request_data,
27         size_t request_size,
28         struct response_data *response) {
29   CURL *curl = curl_easy_init();
30   if (!curl)
31     return -1;
32
33   char url[256];
34   snprintf(url, sizeof(url), "%s/twirp/vaccel.service.Vaccel/%s",
      SERVER_URL, service_method);
35
36   struct curl_slist *headers = NULL;
37   headers = curl_slist_append(headers, "Content-Type: application/
      protobuf");
38
39   curl_easy_setopt(curl, CURLOPT_URL, url);
40   curl_easy_setopt(curl, CURLOPT_POSTFIELDS, request_data);
41   curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, request_size);
42   curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
43   curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_callback);
44   curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void *)response);
45
46   prof_region_start(transport_plus_server_stats);
47
48   CURLcode res = curl_easy_perform(curl);
49
50   prof_region_stop(transport_plus_server_stats);
51
52   curl_slist_free_all(headers);
```

```
  curl_easy_cleanup(curl);

  return (res == CURLE_OK) ? 0 : -1;
}
```

Listing B.2: `twirp_client_util.c` - Implementation for ESP32

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "esp_log.h"
#include "esp_http_client.h"
#include "twirp_client_util.h"
#include "alloc.h"

esp_err_t _http_event_handler(esp_http_client_event_t *evt) {
  struct response_data *response = evt->user_data;

  switch (evt->event_id) {
    case HTTP_EVENT_ON_DATA:
      if (!esp_http_client_is_chunked_response(evt->client)) {
        void *new_ptr = esp_realloc(response->data, response->size
   + evt->data_len + 1);
        if (!new_ptr) {
          ESP_LOGE("HTTP", "Failed to realloc response buffer");
          return ESP_FAIL;
        }
        response->data = new_ptr;
        memcpy(response->data + response->size, evt->data, evt->
   data_len);
        response->size += evt->data_len;
        response->data[response->size] = '\0';
      }
      break;
    default:
```

```
27        break;
28    }
29    return ESP_OK;
30  }
31
32  int twirp_request(const char *service_method,
33          struct prof_region *transport_plus_server_stats,
34          const uint8_t *request_data,
35          size_t request_size,
36          struct response_data *response) {
37    char url[128];
38    snprintf(url, sizeof(url), "%s/twirp/vaccel.service.Vaccel/%s",
       SERVER_URL, service_method);
39
40    esp_http_client_config_t config = {
41      .url = url,
42      .event_handler = _http_event_handler,
43      .user_data = response,
44      .timeout_ms = 100000,
45    };
46
47    response->data = NULL;
48    response->size = 0;
49
50    esp_http_client_handle_t client = esp_http_client_init(&config);
51    if (!client) {
52      ESP_LOGE("HTTP", "Failed to init HTTP client");
53      return -1;
54    }
55
56    esp_http_client_set_method(client, HTTP_METHOD_POST);
57    esp_http_client_set_post_field(client, (const char *)request_data
       , request_size);
58    esp_http_client_set_header(client, "Content-Type", "application/
```

```
    protobuf");
59  esp_http_client_set_header(client, "Accept", "application/
    protobuf");
60
61  prof_region_start(transport_plus_server_stats);
62
63  esp_err_t err = esp_http_client_perform(client);
64
65  prof_region_stop(transport_plus_server_stats);
66
67  esp_http_client_cleanup(client);
68
69  if (err != ESP_OK) {
70    ESP_LOGE("HTTP", "HTTP request failed: %s", esp_err_to_name(err
    ));
71    esp_free(response->data);  // cleanup if failed
72    response->data = NULL;
73    response->size = 0;
74    return -1;
75  }
76
77  return 0;
78 }
```