Generación automática de código mediante microTVM para sistemas empotrados
Automatic code generation with microTVM for embedded systems

Trabajo de Fin de Máster
Curso 2024–2025

Autor
Jaime Pastrana García

Directores
Francisco Daniel Igual Peña
Luis Piñuel Moreno

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Generación automática de código mediante microTVM para sistemas empotrados
Automatic code generation with microTVM for embedded systems

Trabajo de Fin de Máster en Ingeniería Informática
Departamento de Arquitectura de Computadores y Automática

Autor
Jaime Pastrana García

Directores
Francisco Daniel Igual Peña
Luis Piñuel Moreno

Convocatoria: *Julio 2025*
Calificación: *9*

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

AGRADECIMIENTOS

En primer lugar, quiero agradecer a Fran y a Luis su acompañamiento durante la realización de este trabajo. A Fran especialmente por transmitirme confianza en cada reunión y a Luis por introducirme en el mundo de los microcontroladores. Y a los dos por hacerme ver que todos los problemas tienen solución.

En segundo lugar, a mis padres y hermanos, por su apoyo incondicional a lo largo de toda mi etapa universitaria. Por ser las personas más adecuadas para acompañarme en mi día a día.

Por último, pero no menos importante, a mis amigos, porque lo que me enseñan no está escrito en ningún libro.

RESUMEN

Generación automática de código mediante microTVM para sistemas empotrados

La inteligencia artificial ha demostrado en los últimos años ser una pieza clave en el futuro de nuestra sociedad. En su imparable avance, enfrenta serios desafíos computacionales, entre los que se encuentra su integración en dispositivos con recursos especialmente limitados. Con el propósito de facilitar la resolución de este problema, existen una serie de herramientas capaces de optimizar la generación de código adaptado a dispositivos embebidos. En este trabajo se analiza el potencial de una de estas herramientas, MicroTVM, con el fin de determinar su utilidad y aprovechamiento de las últimas innovaciones en las arquitecturas hardware. Para ello, se analiza su impacto en el rendimiento sobre dos microcontroladores ampliamente utilizados en la actualidad y basados en la arquitectura ARM.

El código empleado en la realización de este trabajo se puede encontrar en el siguiente enlace: https://github.com/Jaimepas77/TFM.

Palabras clave

MicroTVM, generación de código, inteligencia artificial, aprendizaje automático, optimización, microcontroladores, SIMD, compilación cruzada.

ABSTRACT

Automatic code generation with microTVM for embedded systems

Artificial intelligence has proven to be a key component in the future of our society in recent years. In its unstoppable advance, it faces serious computational challenges, among which is its integration into devices with especially limited resources. To help solving this problem, there are a number of tools capable of optimizing code generation adapted for embedded devices. This paper analyzes the potential of one such tool, MicroTVM, to determine its utility and its leverage of the latest innovations in hardware architectures. To do this, its impact on performance is analyzed on two widely used microcontrollers based on the ARM architecture.

The code used in this work can be found at the following link:
https://github.com/Jaimepas77/TFM.

Keywords

MicroTVM, code generation, artificial intelligence, machine learning, optimization, microcontrollers, SIMD, cross-compilation.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1

INTRODUCCIÓN

En este primer capítulo se presenta el contexto actual que justifica la realización del presente trabajo. A continuación, se exponen los objetivos concretos que se persiguen, así como la planificación desarrollada para alcanzarlos. El propósito es proporcionar una visión clara del enfoque adoptado, subrayando tanto la relevancia del problema abordado como la contribución que se pretende realizar con este estudio.

1.1. Motivación

En los últimos años, hemos sido testigos de una auténtica explosión en el desarrollo y la aplicación de técnicas de inteligencia artificial (IA). Estos avances, especialmente en el ámbito del aprendizaje automático y el aprendizaje profundo, han propiciado la aparición de aplicaciones inteligentes en una amplia variedad de sectores, desde la industria y la medicina hasta el transporte y el entretenimiento.

Paralelamente, la proliferación de dispositivos conectados ha consolidado el paradigma del Internet de las Cosas (IoT), en el que pequeños dispositivos dotados de capacidades computacionales son capaces de interactuar con el entorno, recopilar datos y tomar decisiones en tiempo real. Estos dispositivos, frecuentemente basados en microcontroladores, se encuentran en entornos frecuentes como cámaras inteligentes de semáforos, sensores de detección de personas de sistemas de alarmas o vehículos con funciones de conducción autónoma.

Para responder a las demandas de rendimiento y eficiencia energética de estas aplicaciones, han surgido nuevas arquitecturas hardware optimizadas para entornos embebidos. Sin embargo, esta diversidad arquitectónica presenta un importante desafío: la adaptación y optimización del software para poder aprovechar al máximo las capacidades de

cada plataforma. En este contexto, la generación automática de código optimizado se ha convertido en una necesidad, especialmente en escenarios con recursos limitados como los microcontroladores.

En este escenario emerge MicroTVM (TVM, 2024a), una extensión del compilador TVM (Chen et al., 2018) orientada a dispositivos con capacidades reducidas. MicroTVM ofrece un marco de trabajo para generar código optimizado para distintas arquitecturas hardware, con el objetivo de facilitar el despliegue eficiente de modelos de aprendizaje automático en estos dispositivos. No obstante, dado lo reciente de estas herramientas y su vinculación con hardware en constante evolución, resulta necesario evaluar su madurez, flexibilidad y rendimiento en contextos reales.

Este trabajo se enmarca dentro de este entorno tecnológico cambiante y busca contribuir al estudio de herramientas que permitan una mejor integración de la inteligencia artificial en dispositivos IoT, con un enfoque particular en el aprovechamiento de MicroTVM como solución para la compilación eficiente de modelos en plataformas basadas en microcontroladores.

1.2. Objetivos

El objetivo del presente trabajo es realizar un análisis del potencial de MicroTVM como herramienta de generación de código para arquitecturas hardware basadas en microcontroladores. Para lograrlo de forma adecuada, se han establecido los siguientes objetivos específicos:

- Análisis de las capacidades existentes en TVM, así como aquellas que añade microTVM en el contexto de los microcontroladores.
- Estudio de los métodos de compilación disponibles, teniendo en cuenta la posibilidad de integrar distintos compiladores y parámetros de configuración que pudieran afectar al resultado final.
- Evaluación comparativa del rendimiento para diferentes arquitecturas hardware, obteniendo métricas reales haciendo uso de herramientas de benchmarking independientes.

1.3. Plan de trabajo

Para lograr los objetivos propuestos, se comenzó a trabajar con la previsión de poder finalizar el trabajo en un año. En este marco de tiempo se han ido realizando las distintas tareas necesarias, logrando completarse en el tiempo previsto. El trabajo realizado se ha estructurado en el tiempo de la siguiente manera:

1. Formación en el uso de las herramientas del proyecto. Durante los primeros meses se estudió la documentación disponible y se realizaron todos los tutoriales oficiales disponibles sobre la herramienta principal.

2. Experimentación con la herramienta, estudiando y aplicando casos de uso reales e innovadores sobre el hardware proporcionado. En aproximadamente 4 meses se estuvieron analizando los resultados de probar distintas configuraciones de la herramienta en hardware no soportado de forma predeterminada. Para ello se modificó el código interno de la herramienta, así como se crearon *scripts* de Python que proporcionan la adaptabilidad necesaria.

3. Análisis comparativo de los resultados y redacción de la memoria. En los últimos 3 meses se tomaron y analizaron los datos de rendimiento de la herramienta sobre distintos dispositivos relevantes en el contexto de los dispositivos IoT. También se redactó paralelamente la memoria del proyecto.

Para poder cumplir con la planificación y superar limitaciones técnicas, se tuvo que pivotar en el hardware empleado y los objetivos previstos. En cualquier caso, esta eventualidad ha sido manejada sin mayor inconveniente, dado que se encontraba dentro de los escenarios previstos al trabajar con hardware novedoso al que aún no se han adaptado todas las herramientas existentes.

CAPÍTULO 2

ESTADO DE LA CUESTIÓN

La ejecución de modelos de aprendizaje automático bajo términos de eficiencia computacional aceptables es uno de los principales retos en el campo de la IA. En dispositivos embebidos o de IoT, el reto es mayor, dado que los recursos en este tipo de elementos hardware acostumbran a ser más limitados. Además, se enfrenta a un ecosistema de arquitecturas de procesadores muy diverso, por lo que generalizar las optimizaciones aplicables sobre los modelos es una tarea tan compleja como necesaria. Profundizando en estos desafíos, podemos distinguir los siguientes puntos:

- Limitaciones computacionales y de memoria: Los microcontroladores a menudo carecen de unidades de punto flotante de hardware (FPUs) o están restringidos a un número limitado de aceleradores especializados. Además, la memoria disponible, tanto SRAM como Flash, se mide en kilobytes en lugar de megabytes o gigabytes, lo que impone la necesidad de modelos de IA altamente comprimidos para poder ser alojados y ejecutados.

- Eficiencia energética y gestión térmica: Muchos de estos dispositivos operan con baterías, lo que introduce restricciones estrictas sobre el consumo que pueden realizar las tareas de IA.

- Restricciones en tiempo real: La necesidad de respuestas rápidas en aplicaciones críticas, como drones o robots industriales, hace que la latencia sea un factor determinante. El procesamiento local de datos es esencial para eliminar los retrasos inherentes a la comunicación con servidores independientes.

- Fragmentación del ecosistema hardware: La heterogeneidad de las arquitecturas de hardware, que incluye diversas arquitecturas de conjunto de instrucciones (ISA),

soporte de FPU y la presencia o ausencia de aceleradores, dificulta enormemente la generalización de las optimizaciones de software.

- Complejidad del despliegue: A pesar de la existencia de frameworks especializados, el desarrollo para sistemas embebidos sigue requiriendo conocimientos especializados en ingeniería de sistemas embebidos, optimización de código C/C++, compresión de modelos e integración de sensores.

En las secciones siguientes se analiza el estado actual de estas cuestiones en mayor profundidad, introduciendo los retos y las soluciones existentes en el panorama.

2.1. Herramientas de generación de código

El proceso de generación de código optimizado y adaptado a una arquitectura *hardware* específica es un proceso costoso y propenso a errores. Es por esto que las grandes empresas han desarrollado entornos de compilación especializados que persiguen automatizar la resolución de este problema. Algunos de los ejemplos más representativos son los casos de TensorFlow Lite (TensorFlow, 2025), PyTorch Glow (PyTorch, 2025), TVM (Chen et al., 2018) y OpenVino (OpenVino). Todos estos entornos siguen un enfoque basado en capas de abstracción, de modo que las superiores se centran en las optimizaciones de más alto nivel (y, por tanto, más generalizables) y las inferiores en aquellos procesos dependientes de la arquitectura hardware de destino. A modo ilustrativo, se incluye el caso de TVM en la Figura 2.1. Como implementación de las capas de más bajo nivel existen dos enfoques principales: compilación adelantada en el tiempo (AoT, por sus siglas en inglés) y compilación basada en el grafo del modelo. Ambas opciones se explican con mayor detalle en el capítulo 3, en relación a la aplicación de TVM sobre microcontroladores. En cualquier caso, se puede resumir en que la compilación AoT acostumbra a proporcionar un mejor rendimiento, pero actualmente tiene un soporte escaso por su complejidad en la implementación. Esto se ejemplifica en el caso de TensorFlow Lite Micro (David et al., 2021), que soporta este tipo de compilación pero bajo unas condiciones (requiere que C++ 17 esté soportado) que limitan sus casos de uso reales.

A pesar de las diferencias entre los frameworks mencionados, todos persiguen objetivos similares: mejorar el rendimiento de ejecución de los modelos, reducir su tamaño y facilitar su despliegue en dispositivos con recursos limitados. Para ello, incorporan técnicas como cuantización, fusión de operadores y poda, que permiten disminuir el uso de memoria y acelerar la inferencia sin comprometer significativamente la precisión del modelo.

Una característica común entre estas herramientas es la posibilidad de realizar una compilación cruzada, permitiendo generar ejecutables en un entorno de desarrollo (host) destinados a ejecutarse en otro hardware (target). Esta funcionalidad es fundamental en el contexto de los dispositivos embebidos, ya que estos muchas veces no disponen de los recursos necesarios para realizar una compilación local.

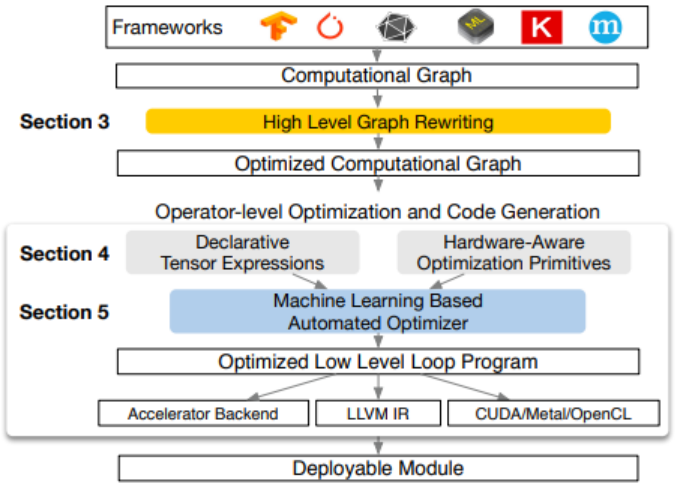

Figura 2.1: Diseño de TVM (Chen et al., 2018)

TVM, en particular, destaca por su enfoque modular y su capacidad de generación de código altamente especializado a través de su componente AutoTVM. Esta herramienta permite buscar automáticamente configuraciones de ejecución óptimas para operadores específicos, utilizando técnicas de búsqueda por refuerzo o aprendizaje automático. Esto lo convierte en una herramienta particularmente potente en entornos heterogéneos, donde las características del hardware varían significativamente de un dispositivo a otro.

Por su parte, PyTorch Glow adopta un enfoque intermedio entre la representación de alto nivel y la optimización de bajo nivel. Glow transforma los modelos PyTorch en una representación intermedia (IR) que puede ser optimizada antes de ser mapeada a la arquitectura destino. Esta estrategia le proporciona cierta flexibilidad y eficiencia, pero su soporte para arquitecturas menos convencionales (como microcontroladores) es limitado en comparación con TVM.

Finalmente, OpenVINO de Intel está diseñado específicamente para arquitecturas de esta empresa, como CPUs, GPUs integradas y VPUs. Esto le otorga una ventaja en cuanto a rendimiento en estos dispositivos, pero restringe su aplicabilidad a otros entornos más abiertos o heterogéneos.

Además de los frameworks anteriores, existen otros proyectos relevantes que exploran distintas aproximaciones al problema de generación de código optimizado:

- MLIR (Multi-Level Intermediate Representation) (Lattner et al., 2021): Desarrollado inicialmente por Google, propone una infraestructura flexible y extensible para representar transformaciones de alto y bajo nivel en compiladores modernos. Es la base actual del compilador de TensorFlow XLA y se encuentra en rápida expansión.

- Halide (Ragan-Kelley et al., 2017): Lenguaje de dominio específico (DSL) centrado en la generación eficiente de cadenas de procesamiento de imágenes, pero extendido

a tareas de cómputo general. Su diseño separado entre algoritmo y planificación lo hace muy adecuado para explotar la jerarquía de memoria y paralelismo.

- Exo (Ikarashi et al., 2022): Lenguaje desarrollado por el MIT para escribir y transformar kernels de bajo nivel con precisión, pensado para facilitar la generación de código personalizado para hardware especializado. En el contexto de la compilación de un modelo, permite decidir de forma manual qué optimizaciones aplicar sobre la representación intermedia de este. Es una propuesta reciente, pero con alto potencial para sistemas embebidos.

En resumen, aunque existe una amplia variedad de herramientas orientadas a la generación de código optimizado para modelos de aprendizaje automático, la elección de una u otra depende en gran medida del hardware de destino, los requisitos de rendimiento y los recursos disponibles durante el desarrollo. Las soluciones actuales, si bien potentes, todavía presentan limitaciones a la hora de generalizar sus técnicas a plataformas embebidas con restricciones extremas, lo que motiva la necesidad de seguir explorando alternativas más ligeras, eficientes y portables.

Entre todas las opciones disponibles, en este trabajo se ha seleccionado TVM para analizar en mayor profundidad su potencial en el mundo de la compilación de modelos de aprendizaje automático para dispositivos embebidos. La extensión de TVM que afronta este objetivo es MicroTVM (TVM, 2024a). A continuación, se analizan los desafíos que enfrenta, así como sus perspectivas de desarrollo futuro.

2.1.1. MicroTVM

Existen ciertas limitaciones y complejidades que marcan el estado actual de su desarrollo y adopción:

- Curva de aprendizaje pronunciada: MicroTVM, como parte del ecosistema TVM, puede presentar una curva de aprendizaje considerable en comparación con otras herramientas similares (Ortiz et al., 2025), lo que representa una barrera para nuevos usuarios o equipos sin experiencia previa en compiladores de modelos de aprendizaje automático.

- Características en desarrollo activo: Algunas de las funcionalidades de MicroTVM aún se encuentran en fases de desarrollo activo. Esto puede implicar cambios en la API, posibles errores o la necesidad de soluciones alternativas para ciertas características.

- Dependencia de compiladores externos: Aunque la flexibilidad de MicroTVM para integrarse con compiladores específicos es una ventaja, la compatibilidad y la explotación de las optimizaciones proporcionadas por herramientas externas pueden resultar un desafío complejo.

Incluso con la existencia de estas limitaciones, la visión de MicroTVM al comienzo del proyecto resultaba prometedora a la par que ambiciosa. El planteamiento propuesto mostraba una línea de desarrollo estable que se beneficiara de las innovadoras capacidades de TVM y abriera nuevas posibilidades en el despliegue de modelos de IA sobre microcontroladores. Sin embargo, y por desgracia para todos los interesados en hacer uso de sus funcionalidades, la falta de mantenimiento general de la herramienta ha motivado que en diciembre de 2024 la herramienta se retirara del desarrollo y soporte general de la comunidad existente en torno a TVM. Esto se comunicó a través del foro oficial de TVM (TVM, 2024b).

2.2. Arquitecturas principales

A la hora de mapear un modelo de aprendizaje automático sobre un microcontrolador, es importante tener en cuenta la arquitectura del mismo y la potencial existencia de aceleradores integrados. La relevancia de estas variaciones en el hardware reside en el uso de primitivas distintas para cada caso. Estas diferencias se deben no sólo a las peculiaridades en el diseño de cada fabricante, sino también a la existencia de capacidades computacionales exclusivas de arquitecturas especializadas. En esta última línea se distinguen, como variantes de mayor relevancia para aplicaciones de IA, la existencia separada de CPUs, GPUs y TPUs. Como se puede observar en la Figura 2.2, se diferencian en la organización de la memoria o en las unidades funcionales de computación, entre otras. Para este proyecto se acota la experimentación a arquitecturas de tipo CPU, aunque sería interesante ampliarlo en un futuro a otras arquitecturas más especializadas.

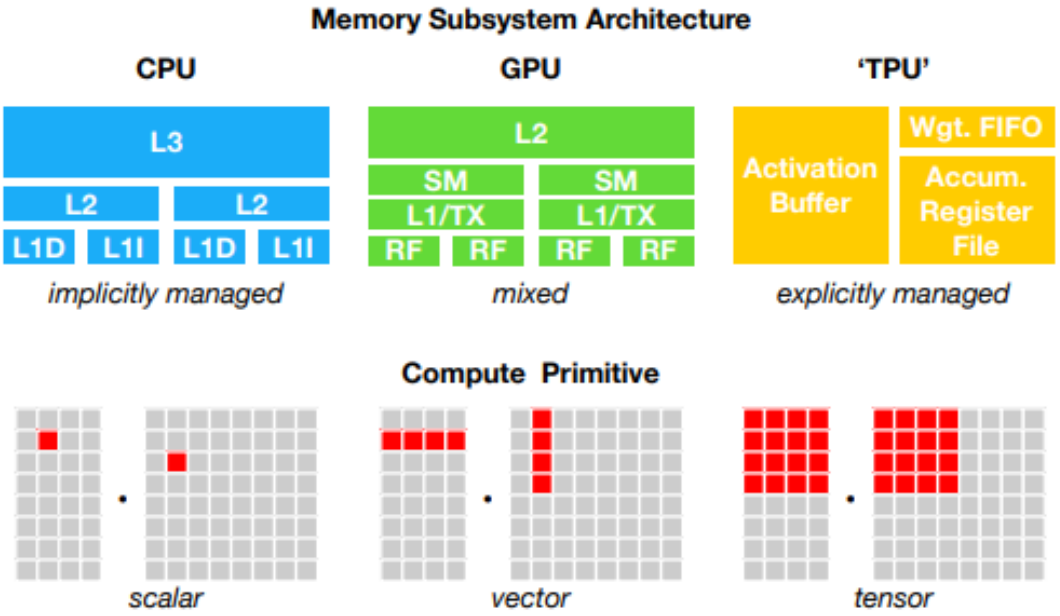

Figura 2.2: Arquitecturas de destino (Chen et al., 2018)

En el contexto de las arquitecturas de las CPUs, podemos clasificarlas en dos tipos:

- Arquitecturas CISC (del inglés Complex Instruction Set Computing). Se basan en la idea de poder agrupar más carga computacional en cada unidad funcional de computación o instrucción. Estas arquitecturas cuentan con un conjunto de instrucciones más amplio y complejo, lo que permite realizar operaciones más elaboradas con una sola instrucción. Esto puede simplificar la programación en lenguaje ensamblador y reducir el número de instrucciones necesarias para realizar una tarea, aunque a costa de una mayor complejidad en el diseño del hardware y, en algunos casos, menor eficiencia en la ejecución. El ejemplo más común de este tipo de arquitectura es x86, empleado de forma extensiva por Intel y AMD.

- Arquitecturas RISC (del inglés Reduced Instruction Set Computing). Se basan en un conjunto reducido y optimizado de instrucciones simples y de ejecución rápida. Cada instrucción realiza una operación muy básica, lo que permite una ejecución más predecible y eficiente, facilitando la implementación de canalizaciones (pipelines) en el procesador. Esto suele traducirse en una mayor velocidad de ejecución por instrucción y en una arquitectura más sencilla y escalable. Sin embargo, puede requerir un mayor número de instrucciones para completar tareas complejas. Los ejemplos más comúnes actualmente de este tipo de arquitectura son ARM y RISC-V, que además destacan especialmente en el entorno de los dispositivos embebidos.

Como consecuencia de lo expuesto, especialmente en lo referente a eficiencia, la gran mayoría de los microcontroladores emplean arquitecturas de tipo RISC. En los siguientes apartados se detallan las características de las dos implementaciones principales: ARM y RISC-V.

2.2.1. ARM

ARM es una arquitectura RISC propietaria. Pertenece a ARM Holdings, quien comercializa la propiedad intelectual de su diseño a grandes empresas. La licencia de la arquitectura ARM permite desarrollar procesadores basados en su núcleo, simplificando el diseño y reduciendo los costes de desarrollo del hardware. Actualmente es la arquitectura más extendida en el mundo en cuanto a microcontroladores se refiere, estando presente en la inmensa mayoría de teléfonos móviles y dispositivos embebidos.

De cara a su uso en aplicaciones basadas en IA, cobra especial relevancia el equilibrio entre rendimiento y consumo, aspecto en el que ARM destaca. También es importante la compatibilidad con aceleradores y una amplia gama de microcontroladores, que facilitan la escalabilidad en caso de ser necesaria. Para poder aprovechar estas características de la mejor manera posible, existen librerías especializadas en aplicaciones de IA, tal que CMSIS-NN (especializada en redes neuronales). Como último aspecto a destacar, se tiene

el bajo coste que tienen este tipo de microcontroladores. Esto facilita la adquisición de hardware físico con el que experimentar, como el que se ha empleado en este proyecto.

2.2.2. RISC-V

RISC-V es una arquitectura de microcontroladores basada en RISC. Su principal diferencia con respecto a sus competidores es que es de uso libre y diseño abierto, no cobrándose ningún tipo de licencia ni requiriéndose confidencialidad en la descripción de su funcionamiento. Estas características tienen una importancia crucial en el ámbito comercial, dado que reducen la dependencia de proveedores externos y eliminan barreras en el desarrollo de nuevas funcionalidades.

Además, RISC-V ofrece una gran flexibilidad para personalizar y extender el conjunto de instrucciones, lo que permite adaptar el hardware a necesidades específicas, optimizando el rendimiento y el consumo energético para aplicaciones concretas. Esto es especialmente valioso en la inteligencia artificial embebida, donde los requerimientos varían mucho según el caso de uso.

En cuanto a la comunidad y el ecosistema, RISC-V está creciendo rápidamente, con un número creciente de herramientas de desarrollo, compiladores, sistemas operativos y librerías compatibles. Sin embargo, aún está en proceso de maduración comparado con ARM, especialmente en cuanto a soporte comercial y disponibilidad de hardware de bajo coste, aunque esto está cambiando con la aparición de nuevas placas y microcontroladores basados en RISC-V.

Por último, la naturaleza abierta de RISC-V fomenta la innovación y la colaboración, facilitando el acceso al diseño para investigadores y desarrolladores que buscan experimentar o desarrollar soluciones personalizadas sin los costes asociados a arquitecturas propietarias. Esto puede resultar en un impulso importante para proyectos educativos, prototipado y desarrollo de tecnología avanzada.

CAPÍTULO 3

DESCRIPCIÓN DE LA INFRAESTRUCTURA Y SUS COMPONENTES

La infraestructura que permite generar, compilar y ejecutar modelos de IA sobre los microcontroladores se compone de varios elementos independientes. Estos elementos necesitan integrarse y configurarse en función de la arquitectura del microcontrolador objetivo.

A continuación se describen en profundidad cada uno de los elementos que se han empleado para compilar, ejecutar y monitorizar los modelos de IA.

3.1. Apache TVM

Apache TVM (Tensor Virtual Machine) o simplemente TVM (Chen et al., 2018) es un framework de compilación de modelos de aprendizaje automático. Es también un proyecto de código abierto mantenido por una amplia comunidad y apoyado por empresas como OctoML, Microsoft o Nvidia (TVM, 2025d). Su repositorio en GitHub, desde el que se puede acceder al código fuente, es el siguiente: https://github.com/apache/tvm/.

El objetivo principal de esta herramienta es facilitar a los ingenieros de aprendizaje automático la optimización y ejecución de procesos computacionales sobre cualquier arquitectura hardware. Esta flexibilidad en cuanto a la arquitectura de destino es esencial, dado que es la que nos permite comparar el rendimiento de un mismo modelo sobre distintos dispositivos. En esta línea, TVM cuenta con las siguientes características:

- Compilación de modelos de aprendizaje automático: Esta es la funcionalidad principal de la herramienta, permitiendo obtener código máquina adaptado a una arquitectura específica a partir de descripciones de alto nivel del modelo.
- Optimización automática: Permite adaptar el código a la plataforma objetivo sin necesidad de aplicar optimizaciones de bajo nivel de forma manual. Es uno de los puntos más destacados de la herramienta, dado que ahorra mucho tiempo en el proceso.
- Soporte multiplataforma: Permite adaptarse a cambios en la plataforma objetivo sin apenas fricción, al soportar las más populares bajo un front-end común. Esto resulta especialmente útil para poder compilar los modelos en arquitecturas de microprocesadores, como es el propósito del presente trabajo.
- Extensibilidad: El proyecto está diseñado para poder incorporar nuevos módulos de forma sencilla. Esto permite añadir soporte para casos específicos tales que arquitecturas no implementadas, compiladores no soportados u operaciones nuevas.
- Soporte de los principales frameworks de ML: Permite trabajar de forma independiente a la plataforma en la que se ha diseñado y entrenado el modelo de Machine Learning. El único requisito es que la traducción a la representación soportada por TVM (Relay) esté implementada. De forma predeterminada, TVM soporta TensorFlow, PyTorch o Keras entre otros.

Las características expuestas se implementan sobre una serie de transformaciones estandarizadas del modelo, las cuales comienzan a partir de la descripción de alto nivel de las operaciones que lo componen y van transformándose en implementaciones más específicas hasta llegar al código máquina ejecutable. En cada una de las representaciones intermedias se aplican las optimizaciones posibles que vayan siendo detectadas, que en las representaciones de más alto nivel serán independientes de la arquitectura, pero según se aproximen a la implementación de bajo nivel, podrán emplear características específicas del hardware. Un ejemplo de optimización más específica es el uso de instrucciones de vectorización, que se comportan y especifican de forma distinta en función de la arquitectura de destino. El uso de estas instrucciones puede ser un factor diferencial cuando se compara el rendimiento de un mismo modelo sobre distintos dispositivos, lo que le otorga una alta relevancia en este trabajo.

3.1.1. Pasos para la compilación de un modelo en TVM

La Figura 3.1 muestra las distintas fases por las que pasa la transformación de un modelo de aprendizaje automático desde su descripción de alto nivel hasta su compilación en código máquina. A continuación se describen en mayor profundidad cada uno de los pasos intermedios:

1. TensorFlow/PyTorch/ONNX: Esta es la representación proporcionada por el framework de aprendizaje automático con el que se ha diseñado el modelo. Los formatos más comunes, que son soportados por defecto, son lo proporcionados por TensorFlow, PyTorch, MXNet y ONNX. Debido a la extensibilidad de TVM es posible hacer uso de otros frameworks no soportados siempre y cuando se implemente la transformación inicial de ellos al formato soportado por TVM (Relay).

2. Relay (representación intermedia de alto nivel): Consiste en una forma estándar de definir los modelos de los distintos frameworks de entrada. Los modelos se transforman a este lenguaje de representación común para poder estandarizar la aplicación de optimizaciones y transformaciones de alto nivel.

3. Expresión Tensorial (definición computacional): Presenta el modelo de forma más granular, permitiendo aplicar optimizaciones de más bajo nivel. Para ello, el modelo se representa como un grafo de tensores. Algunas de las optimizaciones aplicables en este punto consisten en la identificación de código vectorizable, paralelizable o en el que se pueden desenrollar o fusionar bucles.

4. AutoTVM o AutoScheduler (refinado automático): Manteniendo la misma representación intermedia del modelo, TVM emplea uno de los dos algoritmos de refinado automático (auto-tuning) disponibles para encontrar la mejor planificación del algoritmo. Para conseguir este objetivo se emplean modelos de coste y mediciones reales de rendimiento en el dispositivo de destino. Este paso es opcional y permite obtener una compilación mejor adaptada para nuestro caso de uso concreto.

5. Especificación de la optimización: Haciendo uso de los datos obtenidos en el paso anterior, se aplican las optimizaciones en la planificación a cada subgrafo del modelo.

6. Representación intermedia de tensores (representación intermedia de bajo nivel): En este paso se trabaja con una representación de los tensores que es ya específica de la arquitectura de destino. En ella se aplican las optimizaciones de más bajo nivel, justo antes de pasar a código máquina. En este punto TVM soporta de forma nativa varios compiladores tales que LLVM o NVCC (NVIDIA, 2025). También soporta el framework BYOC (Bring Your Own Codegen) (Chen et al., 2021) que permite compilar para arquitecturas especializadas.

7. Código máquina: Finalmente se obtiene el código máquina final, que puede ser cargado y ejecutado en el procesador objetivo.

Para hacer uso de las características que ofrece la herramienta, podemos hacerlo a través de la consola o de la API de Python. Aunque la consola requiere de una menor configuración inicial, es más común emplear scripts en Python debido a la versatilidad, escalabilidad y facilidad de uso que ofrecen.

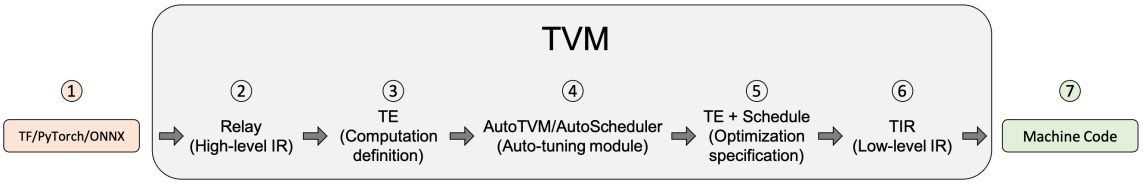

Figura 3.1: Fases de la compilación de TVM (Shiue et al., 2024)

3.2. MicroTVM

MicroTVM (TVM, 2024a) es una extensión de TVM que permite compilar y ejecutar modelos de aprendizaje automático sin necesidad de un sistema operativo. Esto permite extender las capacidades de TVM a dispositivos IoT (Internet of Things) que hacen uso de microcontroladores y, en algunos casos, sistemas operativos en tiempo real (RTOS por sus siglas en inglés). Las capacidades que añade MicroTVM sobre TVM son:

- Ausencia de necesidad de un sistema operativo: Gracias a esto los modelos se pueden desplegar en microcontroladores con recursos computacionales limitados, haciendo uso de un RTOS o implementaciones *bare-metal*.
- Ausencia de dependencias de entornos de ejecución: Permite generar un ejecutable final únicamente con las implementaciones necesarias incluidas, al no incluir entornos de ejecución completos. Esto facilita la optimización del uso de la memoria, ideal para casos en los que es un recurso escaso.
- Optimizaciones específicas de las arquitecturas de los microcontroladores: Las arquitecturas comunes en el entorno de los microcontroladores tienen diferencias significativas, lo que hace necesario adaptar las optimizaciones a estas.
- Soporte para compilación cruzada: Capacidad de compilar el modelo en una máquina independiente y distinta del microcontrolador objetivo.
- Soporte para RPC (Remote Procedure Call): El proceso de RPC permite la ejecución remota del código desde un host (modo de ejecución "Host-Driven", explicado en detalle más adelante), de modo que no haya que cargar el modelo de nuevo cada vez que se hace una modificación. Para ello, el dispositivo cuenta con una implementación de las operaciones que puede requerir el modelo y el dispositivo local llama remotamente a la ejecución de esas operaciones. Esto permite realizar pruebas con modificaciones de forma mucho más ágil.

3.2.1. Pasos para la compilación de un modelo en MicroTVM

La Figura 3.2 presenta los pasos por los que pasa un modelo en microTVM a alto nivel. Aunque comparten gran similitud con los realizados en el flujo general de TVM (especialmente en lo referente a transformaciones del modelo) contienen diferencias relevantes que

hacen que merezca la pena analizarlos por separado. Estas diferencias se deben a que el despliegue del modelo en el dispositivo final es más complejo. Los pasos se describen en mayor profundidad tal como sigue:

1. Importación del modelo: Se importa el modelo de la misma forma en que se hace para TVM. El modelo se convierte a una representación en formato Relay.
2. Transformaciones sobre el modelo: Se aplican transformaciones sobre el modelo. Al ser para una arquitectura con recursos limitados, en este punto puede interesarnos incluir transformaciones que reduzcan la carga del modelo. Las transformaciones se hacen sobre la representación en Relay.
3. Compilación: El modelo se transforma en su representación intermedia basada en tensores para después generar código en C o directamente compilado.
4. Integración: El código generado se integra en un entorno de ejecución completo como pueden ser una librería de ejecución de C o un RTOS.
5. Despliegue: Se construye y despliega el proyecto sobre el dispositivo. En este punto se puede usar el procedimiento de RPC (Remote Procedure Calls) o ejecutar el modelo completo en el propio dispositivo.

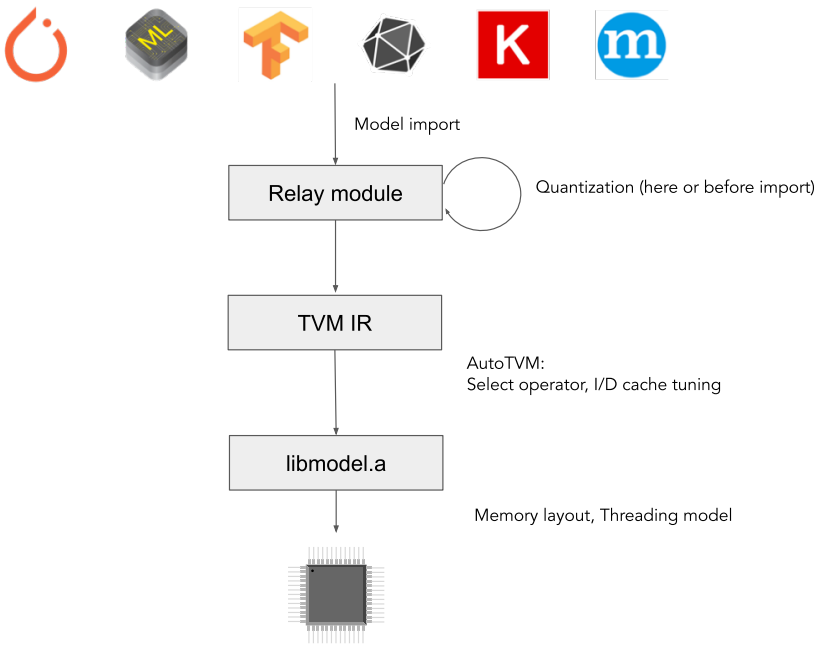

Figura 3.2: Fases de la compilación de MicroTVM (TVM, 2024a)

3.2.2. Configuración del entorno de ejecución

Para la implementación de la ejecución del modelo, existen dos opciones principales: *Ahead of Time* (AoT) y *Graph*. Ambas opciones se basan en el entorno de ejecución de C,

pero difieren en el mapeo e interpretación del modelo y las librerías del sistema.

- *Ahead of Time*: Convierte el grafo que define el modelo en una función ejecutable. Esta función se puede trasladar directamente al entorno de ejecución final. Es la opción más eficiente, dado que la interpretación del código ejecutable del modelo está mapeada de forma explícita.

- *Graph*: Inyecta el grafo del modelo en el proyecto a través de un archivo JSON. Para la ejecución, se lee el grafo y se van llamando a las funciones correspondientes según lo definido en el grafo. Esta implementación requiere de la inclusión de la librería del entorno de ejecución de C. Además, el hecho de tener que interpretar el grafo del modelo en tiempo de ejecución añade una sobrecarga en términos de rendimiento.

3.2.3. Modos de ejecución disponibles

Existen dos modos de ejecución disponibles cuando se despliega un modelo compilado con MicroTVM. Esta flexibilidad está diseñada con el objetivo de facilitar y agilizar la realización de pruebas sobre los microcontroladores de destino. Las opciones son las siguientes:

- *Standalone*: Es la opción más clásica, necesaria para los despliegues definitivos. Consiste en mapear todo el código ejecutable necesario en el microcontrolador, de forma que este pueda operar de forma independiente. La estructura del firmware final se presenta en la Figura 3.3.

- *Host-Driven*: Esta opción está diseñada para permitir la ejecución de pruebas y modificaciones de forma más ágil. Cuando se usa, el dispositivo de destino sólo tiene cargadas las funciones de la librería de C, las cuáles serán invocadas de forma remota según requiera un segundo dispositivo al que denominaremos *host*. El *host* contendrá el grafo del modelo y será habitualmente el dispositivo sobre el que se configura y flashea el microcontrolador.

  Para invocar métodos de forma externa a través de una conexión UART, se hace uso de un mecanismo de RPC (Remote Procedure Calls). Este modo permite liberar al microcontrolador de contener el grafo completo del modelo, evitando nuevos mapeos de memoria cuando este se modifica.

  Para poder ejecutar el modelo en esta modalidad, es necesario que el entorno de ejecución se haya configurado en su opción de Graph. Esto es porque el grafo del modelo debe estar representado de manera independiente al código concreto de las funciones. Se puede observar en mayor detalle en la Figura 3.4.

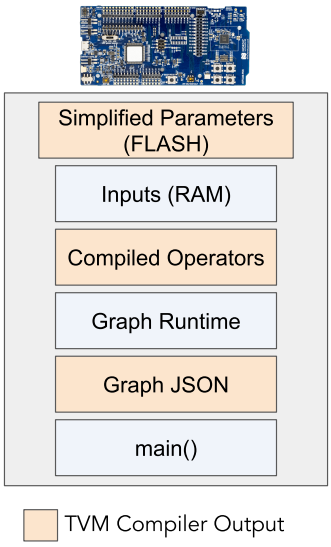

Figura 3.3: Firmware final en modo *Standalone* (TVM, 2024a)

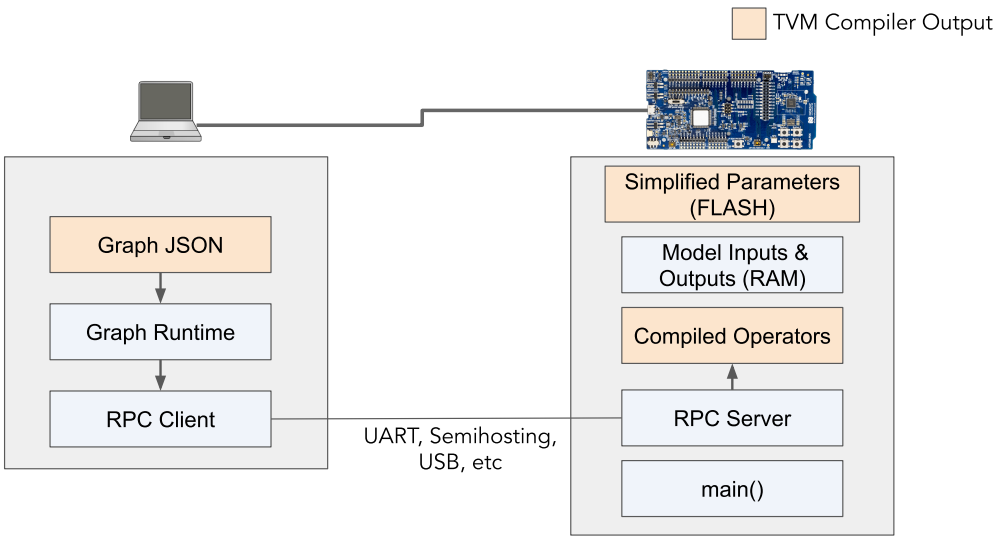

Figura 3.4: Firmware final en modo *Host-Driven* (TVM, 2024a)

3.3. LLVM

LLVM (Lattner y Adve, 2004) es el nombre de un proyecto que agrupa varias herramientas tecnológicas de compilación y optimización de código. Todas las herramientas son de código abierto, estando bajo la licencia de Apache 2.0 (LLVM, 2025a). La idea que soporta el diseño de estas herramientas es la posibilidad de tener un compilador modular, en

el cual se puedan separar los distintos pasos de compilación. De esta manera, es mucho más flexible ante la inclusión de nuevos lenguajes de programación y arquitecturas hardware de destino.

El diseño modular de LLVM se puede resumir en lo mostrado en la Figura 3.5. Este diseño es relevante para el trabajo con TVM, dado que se puede integrar con él para la optimización final y compilación de los modelos. A continuación se explican cada uno de los puntos.

- Frontend (GHC, llvm-gcc, Clang C, ...): Esta capa es la encargada de tomar el programa en el leguaje de origen y convertirlo a la representación intermedia (IR, por sus siglas en inglés) de LLVM. Los módulos pertenecientes a esta capa son los únicos que dependen del lenguaje de programación que se quiere compilar. Esto es gracias a la capacidad de representarlos a todos mediante un mismo lenguaje de representación intermedia.

- Optimización: Este paso trabaja con la IR para obtener versiones funcionalmente equivalentes pero mejor optimizadas. Para ello se encarga de encontrar patrones de casos que puedan mejorar en rendimiento si se cambia la forma de representarse. Es conceptualmente similar a la fase de optimización de TVM.

- Backend (X86, PowerPC, ARM, ...): La última fase consiste en la generación de código ejecutable a partir de la IR. Esta fase es dependiente de la arquitectura hardware de destino, y puede aplicar optimizaciones derivadas del uso de características específicas del hardware (como pueden ser la vectorización o el uso de tipos de registro especializados).

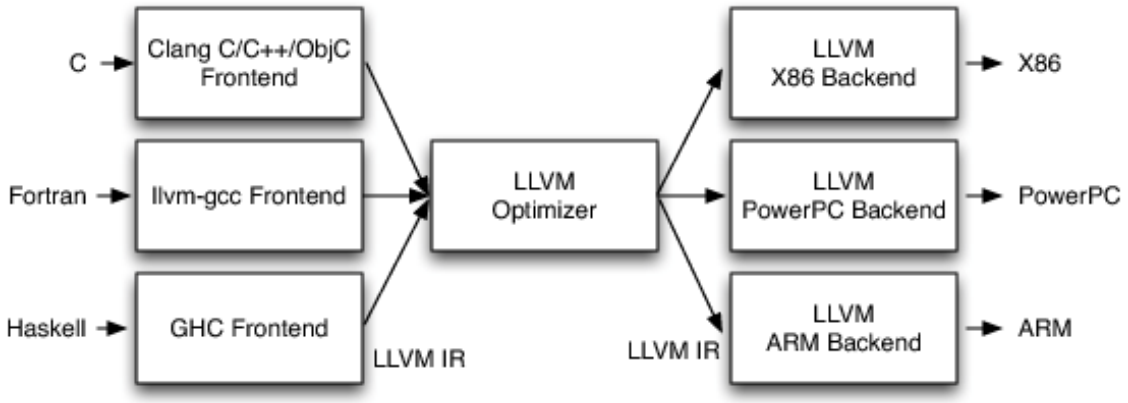

Figura 3.5: Esquema del diseño modular de LLVM (Lattner, 2025)

Las características anteriormente expuestas son esenciales para completar las funcionalidades de TVM. El diseño de TVM comparte la idea de lograr un diseño modular que permita reutilizar componentes e integrarlos más fácilmente con distintos casos de uso.

Como consecuencia, LLVM es sólo una de las opciones disponibles cuando queremos generar el código ejecutable final. En cualquier caso, esta es la opción más común y la usada en nuestro caso. Por tanto, las herramientas proporcionadas por TVM se usan para los siguientes dos propósitos:

1. Integración nativa con la representación intermedia de LLVM: Además de usarse la representación intermedia propia basada en tensores de TVM, se puede realizar una conversión directa a la IR de LLVM por medio de su correspondiente API. Esto permite mantener las optimizaciones detectadas por TVM al tiempo que se aprovechan las capacidades de LLVM, como pueden ser sus optimizaciones o capacidad de generación de código ejecutable.

2. Compilación de proyectos en Zephyr: Tras generarse el código ejecutable de un modelo con MicroTVM, el procedimiento habitual consiste en integrarlo en un proyecto de Zephyr. Este proyecto debe compilarse para después mapearse en el microcontrolador de destino. Convenientemente, LLVM proporciona las herramientas necesarias para realizar la compilación final del proyecto en Zephyr.

Para compilar código fuente cuyo destino es una arquitectura ARM haciendo uso de sus más avanzadas instrucciones vectoriales, existe una versión adaptada del conjunto de herramientas proporcionadas por LLVM. Esta cadena de herramientas, denominada oficialmente LLVM Embedded Toolchain for Arm (LLVM, 2025b), se ha intentado integrar en el flujo de compilación de MicroTVM sin éxito. De haberse conseguido, habría sido capaz de aprovechar la existencia de las instrucciones vectoriales Helium (ARM, 2025a) sobre el procesador Cortex-M85 (ARM, 2025b), obteniéndose una mejora notable en el rendimiento de aquellos microcontroladores que contengan este procesador.

3.4. Zephyr

Zephyr (Project, 2025a) es un sistema operativo en tiempo real (RTOS por sus siglas en inglés) de código abierto. Al ser un RTOS, más ligero que otros sistemas operativos como Linux, es ideal para desplegarse en microcontroladores. Además, es compatible con una gran variedad de arquitecturas hardware. Para lograr sus objetivos, es mantenido a través del Zephyr Project (Project, 2025b), que es un proyecto colaborativo amparado por la Linux Foundation.

Una de las características más importantes de Zephyr para este proyecto es su modularidad, que permite una gran flexibilidad en el uso de sus componentes. Gracias a ello, MicroTVM es capaz de generar proyectos en Zephyr en los que poder incorporar los modelos compilados. Esta integración dota a MicroTVM de la capacidad de desplegar sus modelos compilados sobre todos los microcontroladores que soporta Zephyr.

3.4.1. West

West (Project, 2025c) es una herramienta de línea de comandos proporcionada y mantenida por el Zephyr Project. Su propósito es soportar la gestión de repositorios, proporcionando funcionalidades de construcción, mapeo y depuración del código. Se utiliza para construir y mapear los proyectos de Zephyr.

3.5. Dispositivos hardware

Como hardware físico sobre el que mapear los resultados de compilación de los modelos de aprendizaje automático, se han empleado distintas opciones para poder comparar el rendimiento en cada una de ellas. Además, se explora la adaptabilidad de MicroTVM a cada plataforma de destino. A continuación, se describen en detalle los dispositivos hardware empleados.

3.5.1. STM32 Nucleo-144 para el microcontrolador STM32H743

La STM32 Nucleo-144 (STMicroelectronics, 2025a) es un modelo de placa diseñado para poder hacer uso de los microcontroladores diseñados por STMicroelectronics. En este caso, se ha hecho uso del microcontrolador STM32H743ZI (STMicroelectronics, 2025b), que está basado en un núcleo Arm Cortex-M7. Este núcleo tiene un uso muy extendido en el mundo de los microcontroladores, por lo que la evaluación de su rendimiento resulta especialmente relevante.

Como especificaciones técnicas destacadas del dispositivo, tenemos:

- Microcontrolador: STM32H743ZI.
- Núcleo: Arm Cortex-M7 (frecuencia máxima de procesamiento de 480 MHz).
- Memoria Flash: 2 MB.
- Memoria SRAM: 1 MB.

3.5.2. Renesas EK-RA8M1 Evaluation Kit

La Renesas EK-RA8M1 Evaluation Kit (Renesas, 2025a) es una placa diseñada para probar y evaluar las características del microcontrolador RA8M1 (Renesas, 2025b). Este microcontrolador es el primero de la serie RA de Renesas que está basado en el núcleo del ARM Cortex-M85 (ARM, 2025b). Como se ha mencionado anteriormente (en el capítulo 3.3), este tipo de procesadores posee una serie de características innovadoras respecto a sus predecesores que, de aprovecharse, le permitirían obtener una mejora notable en el rendimiento. La más destacable de estas características es la incorporación de las instrucciones vectoriales Helium (ARM, 2025a), que permiten hacer uso de instrucciones de tipo

SIMD (Single Instruction, Multiple Data) con una mayor agrupación de datos. De forma adicional a su relevancia en términos de computación general, es destacable su potencial en el contexto del aprendizaje automático, que acostumbra a tener numerosas operaciones vectorizables.

Sus especificaciones técnicas más relevantes son:

- Microcontrolador: R7FA8M1AHECBD.
- Núcleo: ARM Cortex-M85 (frecuencia máxima de procesamiento de 480 MHz).
- Memoria Flash: 2 MB.
- Memoria SRAM: 1 MB.

3.6. STM32CubeProgrammer

STM32CubeProgrammer (STMicroelectronics, 2025c) es la herramienta oficial de STM para mapear secciones de memoria en sus microcontroladores. Para ello, hace uso de una distribución propia de OpenOCD (Rath, 2025). Se usa de forma integrada en el flujo de construcción y mapeado de los proyectos en Zephyr sobre la placa STM32 Nucleo-144.

3.7. J-Link

J-Link (Renesas, 2025c) es la herramienta soportada para mapear y depurar código sobre una serie de placas del fabricante Renesas, entre las que se incluye la EK-RA8M1.

3.8. EEMBC

EEMBCs EnergyRunner® (EEMBC) es un conjunto de herramientas de benchmarking proporcionado por la organización sin ánimo de lucro EEMBC. Estas herramientas buscan crear una forma estandarizada de medir el rendimiento de distintos microcontroladores. MLPerf™ Tiny (Banbury et al., 2021) hace uso de estas herramientas para, mediante una serie de modelos preseleccionados, crear unas líneas de referencia en cuanto a rendimiento se refiere. Este conjunto de herramientas y modelos es el usado para comparar el rendimiento en los microcontroladores empleados en este proyecto.

3.9. QEMU

QEMU (Quick Emulator) (QEMU, 2025) es una herramienta de emulación y virtualización de código abierto que se utiliza ampliamente en el desarrollo y prueba de sistemas

embebidos. En el contexto de este proyecto, QEMU se emplea como una solución para emular plataformas de hardware sobre las cuales no se dispone de acceso físico, lo que permite realizar pruebas y depuración sin la necesidad de contar con los dispositivos reales.

QEMU soporta una gran variedad de arquitecturas, incluidas las basadas en ARM, como los microcontroladores que se están utilizando en este proyecto (STM32 y Renesas). Gracias a su capacidad de emulación, QEMU puede simular el comportamiento de estos microcontroladores, permitiendo validar los modelos de aprendizaje automático compilados por MicroTVM en un entorno controlado.

Para este proyecto, destacan las siguientes características de QEMU:

- Emulación de microcontroladores ARM: QEMU puede emular microcontroladores de arquitectura ARM, lo que incluye los núcleos Cortex-M7 y Cortex-M85 de los dispositivos STM32H743ZI y Renesas RA8M1, respectivamente. Esto facilita las pruebas sin necesidad de tener el hardware específico a mano.
- Desarrollo sin hardware físico: QEMU permite que el flujo de trabajo de desarrollo y prueba continúe sin interrupciones incluso cuando no se dispone de dispositivos físicos para todas las plataformas de destino. Esto resulta especialmente útil en la fase inicial del desarrollo.
- Compatibilidad con Zephyr y MicroTVM: QEMU se integra bien con Zephyr y su ecosistema de desarrollo. Al ser una herramienta de emulación de código abierto, puede configurarse para funcionar con Zephyr en un entorno de desarrollo sin hardware real, lo cual es perfecto para la construcción y prueba de modelos de aprendizaje automático sobre microcontroladores.

Aunque QEMU ofrece grandes ventajas en términos de emulación y flexibilidad, también presenta algunas limitaciones que deben tenerse en cuenta. Entre ellas se cuentan la falta de precisión en la emulación, las diferencias en rendimiento respecto al hardware real o la falta de soporte para características específicas del hardware simulado.

CAPÍTULO 4

MÉTODOS DE MAPEADO DE MODELOS DE ALTO NIVEL A MICROCONTROLADORES

En este capítulo se exponen de forma práctica las funcionalidades más relevantes de MicroTVM. Para ello se explican primero los pasos a dar para instalar las herramientas necesarias, haciendo especial hincapié en los puntos que varían según el hardware utilizado. Después se exponen distintos casos de uso, explorando herramientas (interfaz de línea de comandos y scripts de Python) y funcionalidades (modos de compilación, autotuning, medición del rendimiento, etc). Los conocimientos incluidos en estas explicaciones son los necesarios para obtener los resultados de rendimiento finales que se analizan comparativamente en el capítulo 5.

4.1. Instalación

Para la instalación de MicroTVM es necesario seguir los pasos de instalación de TVM a partir de su código fuente (TVM, 2025a). Para el proyecto se ha empleado la versión 0.17.0, descargada desde su correspondiente repositorio de GitHub (TVM, 2025b). Se debe prestar especial atención a las funcionalidades que queremos habilitar a través del fichero *config.cmake*, dado que es necesario especificar ahí la opción correspondiente a MicroTVM. Además, para lograr que la construcción del código fuente fuera correcta, fue necesario habilitar también la opción de *libbacktrace*.

De forma adicional, se necesita contar con una instalación adecuada del entorno de

herramientas proporcionadas por Zephyr. Para este propósito, se hace uso de las instrucciones proporcionadas en los tutoriales de MicroTVM. La versión de Zephyr utilizada es la 3.7.99, que da soporte, entre muchos otros, a los dispositivos empleados posteriormente.

La instalación descrita anteriormente es funcional para aquellos dispositivos soportados de forma predeterminada por MicroTVM. Para añadir nuevos dispositivos, es necesario configurar detalles específicos, como pueden ser el software de mapeo o la arquitectura del hardware. A continuación, se exponen los cambios necesarios para poder hacer uso de los dispositivos objetivo de este trabajo.

4.1.1. Adaptación de la instalación a los dispositivos hardware empleados

Para poder utilizar dispositivos no soportados de forma predeterminada por MicroTVM, como lo son los empleados en este proyecto, es necesario modificar su instalación. Estas modificaciones deben indicar a MicroTVM las características específicas de los dispositivos incluidos. Las modificaciones realizadas se exponen a continuación.

4.1.1.1. Modificación del fichero *boards.json*

MicroTVM posee un fichero en el que se almacenan los datos de Zephyr asociados a cada placa. Este fichero se encuentra en la ruta *<path a tvm>/apps/microtvm/zephyr/template_project/boards.json*. Si queremos que el proceso de compilación e integración en un proyecto de Zephyr funcione correctamente en una placa no soportada de forma predeterminada, debemos modificar este fichero. Este fichero sigue el formato json. En la Figura 4.1 se muestra el contenido añadido para hacer funcionar los dispositivos empleados en este trabajo.

4.1.1.2. Modificación del fichero *MicroTVM_api_server.py*

Este fichero afecta al funcionamiento de MicroTVM en su comunicación con los dispositivos de destino. Resulta que para que funcione correctamente con los dispositivos objetivo, ha sido necesario reparar ciertos errores contenidos en él. El fichero se encuentra en la siguiente ruta: *tvm/apps/microtvm/zephyr/template_project/MicroTVM_api_server.py*. Las líneas modificadas son las mostradas en la Figura 4.2. ±

4.1.1.3. Modificación del fichero *target.py*

En este fichero se especifica el parámetro de compilación asociado a la arquitectura de la placa en uso. Las placas de STMicroelectronics ya tienen especificado este parámetro para la mayoría de sus modelos, pero no es así con los dispositivos de Renesas. Por ello,

```
{
    ...
    "nucleo_h743zi": {
        "board": "nucleo_h743zi",
        "model": "stm32h7xx",
        "is_qemu": false,
        "fpu": true,
        "vid_hex": "0483",
        "pid_hex": "374e"
    },
    "ek_ra8m1": {
        "board": "ek_ra8m1",
        "model": "ek_ra8xx",
        "is_qemu": false,
        "fpu": true,
        "vid_hex": "1366",
        "pid_hex": "1024"
    }
}
```

Figura 4.1: Contenido del fichero *boards.json*

```
±  ZEPHYR_VERSION = 3.7

   ...

±  set_platform = re.match(r"set\(EMU_PLATFORM (.*)\)", line)

   ...

   "# For RPC server C++ bindings.\n"
-  "CONFIG_CPLUSPLUS=y\n"
-  "CONFIG_LIB_CPLUSPLUS=y\n"
+  "CONFIG_CPP=y\n"
+  "CONFIG_REQUIRES_FULL_LIBCPP=y\n"
```

Figura 4.2: Líneas de código Python modificadas en el archivo *MicroTVM_api_server.py*

debemos añadir el contenido mostrado en la Figura 4.3 al fichero de la siguiente ruta: *<path a tvm>/python/tvm/target/target.py*.

4.2. Instrucciones de uso

Esta sección se dedica a explicar los pasos que se dan cuando se compila un modelo de aprendizaje automático con MicroTVM. Entre medias, se aprovecha para introducir las distintas funcionalidades y variantes de configuración que se pueden aplicar.

```
MICRO_SUPPORTED_MODELS = {
    ...
    "ek_ra8xx": ["-mcpu=cortex-m85"],
    ...
}
```

Figura 4.3: Líneas de código Python modificadas en el archivo *target.py*.

4.2.1. Descarga e importación del modelo (frontend)

En primer lugar, debemos tener descargado el modelo que queremos compilar mediante MicroTVM. El modelo debe provenir de un framework soportado por MicroTVM; de lo contrario, sería necesario añadir un nuevo frontend. Algunos de los modelos soportados por MicroTVM son, como ya se mencionó en capítulos anteriores, TensorFlow, PyTorch u ONNX. Podemos realizar este procedimiento de diversas maneras, como pueden ser a través de un navegador web, un script de Python o la propia línea de comandos. También podemos emplear un modelo generado de forma propia, en cuyo caso no sería necesario descargar el modelo de una fuente externa. La Figura 4.4 muestra un ejemplo de cómo se haría esto en Python.

```
from tvm.contrib.download import download_testdata

MODEL_URL = (
    "https://github.com/mlcommons/tiny/raw/
    bceb91c5ad2e2deb295547d81505721d3a87d578/benchmark/training/
    keyword_spotting/trained_models/kws_ref_model.tflite"
)
MODEL_FILE_NAME = "kws_ref_model.tflite"
MODEL_PATH = download_testdata(MODEL_URL, MODEL_FILE_NAME, module="model")
```

Figura 4.4: Código Python para descargar un modelo de aprendizaje automático

Tras tener el modelo a compilar ubicado en nuestro sistema local de archivos, podemos importar el modelo en MicroTVM. Para ello podemos hacer uso de la utilidad de línea de comandos de TVM, llamada *tvmc*, o del paquete de Python correspondiente de MicroTVM. Cualquiera que sea la opción que elijamos, deberá mantenerse en el resto de los pasos de compilación. En la Figura 4.5 se muestra el código necesario para importar el modelo desde Python. Como se puede ver, aparte del modelo propio, incluimos información sobre el tamaño de los datos de entrada. Para el caso de *tvmc* esto se hace automáticamente en el momento de compilación, por lo que se muestra un ejemplo de ello ya en la siguiente sección.

```
import os
import tensorflow as tf
import tflite
from tvm import relay

tflite_model_buf = open(MODEL_PATH, "rb").read()
tflite_model = tflite.Model.GetRootAsModel(tflite_model_buf, 0)

input_name = input_details[0]["name"]
input_shape = tuple(input_details[0]["shape"])
input_dtype = np.dtype(input_details[0]["dtype"]).name

relay_mod, params = relay.frontend.from_tflite(
    tflite_model, shape_dict={input_name: input_shape}, dtype_dict={
    input_name: input_dtype}
)
```

Figura 4.5: Código Python para importar un modelo.

4.2.2. Compilación del modelo (backend)

Para compilar el modelo usando MicroTVM se precisa de cuatro elementos: la definición de las operaciones del modelo (importada en el paso anterior), la definición de los parámetros de entrada del modelo (también importada en el paso anterior), la descripción de la arquitectura de destino y la configuración de las librerías de ejecución. Además, de forma opcional, podemos especificar el tipo de implementación que se quiere tener para la ejecución del modelo, distinguiendo entre *AoT* y *Graph* (descritas en la sección 3.2.2). Por defecto, la opción escogida es *Graph*, aunque comúnmente se recomienda el uso de *AoT* debido a su mejor rendimiento y uso eficiente de la memoria.

En el caso de que no se disponga del hardware necesario para ejecutar el modelo, en este punto podemos usar una definición de la arquitectura hardware general. Esta definición general será suficiente para poder simular el proyecto en QEMU.

La Figura 4.6 muestra el proceso de compilación del modelo, incluyendo la configuración previa necesaria. Como resultado, se obtiene un objeto Python que se puede emplear para exportar el modelo o incluir directamente la exportación del modelo en un proyecto Zephyr (como se muestra en la siguiente sección). En caso de querer exportar el modelo, deberemos emplear la línea de código mostrada en la Figura 4.7.

En el caso de estar usando la utilidad de comandos tvmc, no hay más remedio que generar el archivo que contiene la exportación del modelo, dado que no se puede guardar el resultado de otra manera. Para ello se hace uso del comando mostrado en la Figura 4.8. La configuración es similar a la realizada en Python, siendo en este caso toda la información especificada en un solo paso.

El resultado final obtenido tras exportar la compilación del modelo es un archivo en

```
RUNTIME = Runtime("crt", {"system-lib": True})
TARGET = tvm.micro.testing.get_target("crt")
EXECUTOR = Executor("aot")

BOARD = os.getenv("TVM_MICRO_BOARD", default="nucleo_h743zi")
SERIAL = os.getenv("TVM_MICRO_SERIAL", default=None)
TARGET = tvm.micro.testing.get_target("zephyr", BOARD)
with tvm.transform.PassContext(opt_level=3, config={"tir.disable_vectorize"
    : True}):
    module = tvm.relay.build(
        relay_mod, target=TARGET, params=params, runtime=RUNTIME, executor=
    EXECUTOR
    )
```

Figura 4.6: Código Python para compilar un modelo mediante MicroTVM.

```
temp_dir = tvm.contrib.utils.tempdir()
model_tar_path = temp_dir / "model.tar"
export_model_library_format(module, model_tar_path)
```

Figura 4.7: Código Python para exportar el resultado de compilación de un modelo.

```
tvmc compile model.tflite \
    --target='c -keys=cpu -model=host' \
    --runtime=crt \
    --runtime-crt-system-lib 1 \
    --executor='aot' \
    --output model.tar \
    --output-format mlf \
    --pass-config tir.disable_vectorize=1
```

Figura 4.8: Comando para compilar un modelo mediante tvmc.

formato MLF (Model Library Format) (TVM, 2025c). Este archivo contiene toda la información necesaria para poder ejecutar el modelo, siendo en realidad un conjunto de archivos comprimidos que contienen el resultado de compilación de MicroTVM. Este comprimido se puede integrar en un proyecto Zephyr mediante las utilidades proporcionadas por MicroTVM y expuestas en la siguiente sección. También se puede exportar a cualquier otra plataforma que soporte el entorno de ejecución de C. Esta característica aporta una mayor portabilidad y capacidad de integración a MicroTVM.

4.2.3. Integración en Zephyr

Una vez compilado el modelo, es necesario integrarlo en un proyecto que pueda ser ejecutado sobre la plataforma de destino objetivo. Para este propósito, lo más habitual es

emplear el sistema operativo de tiempo real Zephyr. La integración con Zephyr, además de proporcionar la infraestructura software necesaria sin apenas esfuerzo, ofrece soporte para la mayoría de los microcontroladores disponibles en el mercado. Es por ello que el uso de MicroTVM se puede extender con facilidad a nuevas arquitecturas, ya que la mayor parte de la integración requerida es proporcionada por Zephyr.

El proceso de creación del proyecto en Zephyr y la integración del modelo compilado en dicho proyecto es un proceso sencillo. Esto es porque MicroTVM proporciona las herramientas necesarias para realizarlo de forma integrada con el resto de pasos.

En el caso de Python, se muestra un código con las instrucciones a aplicar en la Figura 4.9. Como se puede ver, se hace uso del objeto Python del modelo generado en el paso anterior. Además, se hace uso de configuraciones que en su mayoría son heredadas del paso anterior, tales como la especificación de la arquitectura de destino.

```
template_project_path = pathlib.Path(tvm.micro.
    get_microtvm_template_projects("crt"))
project_options = {}

if use_physical_hw:
    template_project_path = pathlib.Path(tvm.micro.
    get_microtvm_template_projects("zephyr"))
    project_options = {
        "project_type": "host_driven",
        "board": BOARD,
        "serial_number": SERIAL,
        "config_main_stack_size": 4096,
        "zephyr_base": os.getenv("ZEPHYR_BASE", default="/content/
    zephyrproject/zephyr"),
    }

temp_dir = tvm.contrib.utils.tempdir()
generated_project_dir = temp_dir / "project"
project = tvm.micro.generate_project(
    template_project_path, module, generated_project_dir, project_options
)
```

Figura 4.9: Código Python para integrar el modelo compilado en un proyecto Zephyr.

Si se emplea la utilidad de línea de comandos, el comando necesario es el empleado en la Figura 4.10. El resultado final es equivalente: un proyecto modelo de Zephyr con el modelo integrado en él de modo que pueda ser ejecutado.

Se puede observar en ambos casos que se ha especificado la opción *host_ driven* en el tipo de proyecto. Esta opción ya ha sido explicada en la sección 3.2.3 y permite ejecutar el proyecto con control desde un dispositivo externo.

```
tvmc micro create \
    project \
    model.tar \
    zephyr \
    --project-option project_type=host_driven board=nucleo_h743zi
```

Figura 4.10: Comando de tvmc para integrar el modelo compilado en un proyecto Zephyr.

4.2.4. Despliegue sobre la arquitectura de destino

La construcción y el mapeado del modelo en el dispositivo son muy sencillos, ya que toda la configuración necesaria ya se ha realizado en los pasos anteriores. La construcción del proyecto generará los archivos finales que deben mapearse en la memoria del hardware de destino. Tras realizar el mapeo, se obtiene como resultado la ejecución adecuada del modelo sobre el microcontrolador del dispositivo.

Para que el mapeado en memoria funcione en los dispositivos que se han usado para este proyecto (la STM32 Nucleo-144 y la Renesas EK-RA8M1) es necesario añadir un paso intermedio entre la construcción del proyecto y el mapeo en memoria. Este paso se debe realizar porque MicroTVM no soporta estos dispositivos de forma predeterminada. En él, añadimos una referencia al software que debe gestionar la comunicación entre el dispositivo mapeado y el ordenador de trabajo. En el caso de la STM32 Nucleo-144, este software es OpenOCD, mientras que para el modelo de Renesas es J-Link. Esta modificación del proyecto construido se puede automatizar con código Python.

En el caso de utilizar Python, la Figura 4.11 muestra las instrucciones que se emplean para construir y mapear el proyecto de Zephyr en la memoria del microcontrolador, incluyendo las modificaciones necesarias en nuestro contexto. Como se puede observar, no es necesario incluir ningún parámetro nuevo.

```
project.build()

if(BOARD == "nucleo_h743zi"):
   with open(f'{generated_project_dir}/build/CMakeCache.txt', 'a') as file:
      file.write('ZEPHYR_BOARD_FLASH_RUNNER:STRING=openocd\n')
elif(BOARD == "ek_ra8m1"):
   with open(f'{generated_project_dir}/build/CMakeCache.txt', 'a') as file:
      file.write('ZEPHYR_BOARD_FLASH_RUNNER:STRING=jlink\n')

project.flash()
```

Figura 4.11: Código Python para construir y mapear el proyecto Zephyr con el modelo integrado. Incluye la modificación necesaria para funcionar con los dispositivos de este proyecto.

En el caso de emplear la línea de comandos, el proceso es muy similar, con la diferencia

de que se toma como referencia la carpeta del proyecto de Zephyr. La Figura 4.12 muestra los comandos concretos que se deben utilizar.

```
tvmc micro build project zephyr
tvmc micro flash project zephyr
```

Figura 4.12: Comandos en tvmc para construir y mapear el proyecto de Zephyr con el modelo integrado.

4.2.5. Ejecución en modo host-driven

En caso de haber configurado el proyecto en modo *host-driven*, es necesario conectar la ejecución del host con la del microcontrolador para obtener resultados del modelo. De esta manera, el host leerá el grafo de unidades computacionales que debe ejecutar el modelo y solicitará al dispositivo remoto que realice los cálculos correspondientes. Finalmente, se obtendrán los datos de salida del modelo.

Para realizar estos pasos en Python, debemos configurar la comunicación entre ambos dispositivos como una sesión, de modo que después podamos enviar y recibir los pertinentes datos. Los datos deben corresponderse con el formato esperado por el modelo. La Figura 4.13 muestra las instrucciones completas que cumplen con estas especificaciones para un caso de ejemplo.

```
with tvm.micro.Session(project.transport()) as session:
    aot_executor = tvm.runtime.executor.aot_executor.AotModule(session.
    create_aot_executor())
    sample = np.load(SAMPLE_PATH)
    aot_executor.get_input(INPUT_NAME).copyfrom(sample)
    aot_executor.run()
    result = aot_executor.get_output(0).numpy()
    print(f"Index `{np.argmax(result)}`")
```

Figura 4.13: Código Python para ejecutar el modelo en modo host-driven.

En el caso de tvmc, el proceso es más sencillo, pero también menos configurable. En Python disponemos de librerías que nos permiten un manejo sencillo de los datos de entrada de los modelos, pero esto no es así en la línea de comandos. La Figura 4.14 muestra un ejemplo de ejecución en el que la entrada es una repetición del mismo número. El resultado obtenido en la salida se mostrará en la consola.

```
tvmc run --device micro project --fill-mode ones --print-top 4
```

Figura 4.14: Comando en tvmc para ejecutar el modelo en modo host-driven.

4.3. Capacidades adicionales

De forma complementaria, existen una serie de características funcionales que se pueden incorporar al flujo de compilación habitual. Estas características añaden capacidades adicionales a MicroTVM y extienden su usabilidad en proyectos reales. En las siguientes secciones se describen las más relevantes.

4.3.1. Optimización del modelo mediante autotuning

Antes de compilar el modelo, existe la posibilidad de optimizar su representación intermedia de alto nivel para la arquitectura de destino objetivo. Para lograrlo, MicroTVM prueba las distintas implementaciones disponibles para cada operador y selecciona la que obtiene un mejor rendimiento. Esto implica que se debe tener conectado el microcontrolador sobre el que vamos a compilar a fin de poder ejecutar sobre él las pruebas de rendimiento.

Los pasos a dar para aplicar autotuning son los siguientes:

1. En primer lugar se extrae una lista de las tareas (tasks) que disponen de varias implementaciones y, por tanto, son susceptibles de aplicárseles el proceso.

2. Se definen las opciones de configuración relacionadas con el proceso de autotuning, como son el dispositivo de destino o el número de ejecuciones a realizar por cada implementación disponible.

3. Tarea a tarea, se ejecuta el proceso de autotuning según se ha configurado en el paso anterior. Esto generará unos archivos de registro que se podrán pasar como contexto al proceso de compilación.

Extraído de la documentación de MicroTVM, podemos ver el código que permite realizar autotuning en Python en la Figura 4.15.

Aplicando esta funcionalidad, que viene heredada de TVM, logramos un código mejor adaptado a la plataforma de destino. Sin embargo, es también relevante mencionar que el tiempo que conlleva realizar este proceso puede ser bastante largo, convirtiéndolo en un paso a evitar cuando se están haciendo pruebas preliminares. Es, por tanto, adecuado cuando la compilación tiene como propósito el despliegue final de un modelo.

4.3.2. Cuantización

Aunque MicroTVM no ofrece soporte directo para la cuantización de los modelos importados, esta es una optimización que podemos aplicar de forma previa. Resulta especialmente relevante en el contexto en el que se mueve MicroTVM, dado que los dispositivos para los que está diseñado para funcionar disponen de cantidades de memoria muy bajas, lo que limita la información relativa al modelo que se puede llegar a almacenar. Como solución a

```
pass_context = tvm.transform.PassContext(opt_level=3, config={"tir.disable_vectorize": True})

with pass_context:
    tasks = tvm.autotvm.task.extract_from_program(relay_mod["main"], {}, TARGET)

module_loader = tvm.micro.AutoTvmModuleLoader(
    template_project_dir=pathlib.Path(tvm.micro.get_microtvm_template_projects("zephyr")),
    project_options={
        "board": BOARD,
        "verbose": False,
        "project_type": "host_driven",
        "serial_number": SERIAL,
    },
)
builder = tvm.autotvm.LocalBuilder(
    n_parallel=1,
    build_kwargs={"build_option": {"tir.disable_vectorize": True}},
    do_fork=False,
    build_func=tvm.micro.autotvm_build_func,
    runtime=RUNTIME,
)
runner = tvm.autotvm.LocalRunner(number=1, repeat=1, timeout=100, module_loader=module_loader)

measure_option = tvm.autotvm.measure_option(builder=builder, runner=runner)

autotune_log_file = pathlib.Path("microtvm_autotune.log.txt")
if os.path.exists(autotune_log_file):
    os.remove(autotune_log_file)

num_trials = 10
for task in tasks:
    tuner = tvm.autotvm.tuner.GATuner(task)
    tuner.tune(
        n_trial=num_trials,
        measure_option=measure_option,
        callbacks=[
            tvm.autotvm.callback.log_to_file(str(autotune_log_file)),
            tvm.autotvm.callback.progress_bar(num_trials, si_prefix="M"),
        ],
        si_prefix="M",
    )
```

Figura 4.15: Código Python para realizar autotuning con MicroTVM.

este problema, se puede reducir el tamaño de los registros que almacenan los parámetros del modelo. Esta modificación resulta en una pérdida de precisión en los cálculos y, por tanto, en el rendimiento del modelo, pero es en muchos casos la única alternativa posible para poder mapear el modelo con éxito.

La cuantización de los modelos es una opción soportada por la gran mayoría de frameworks de aprendizaje automático disponibles. Un ejemplo de ello es TFLite, que permite la cuantización de sus modelos (y sus conjuntos de datos de entrada). Para aplicar esta cuantización y exportar el modelo, podríamos emplear un código Python como el mostrado en la Figura 4.16. Para comparar el modelo con su versión anterior, podemos utilizar la herramienta de visualización de modelos disponible en https://netron.app.

```python
def representative_dataset():
    for image_batch, label_batch in full_dataset.take(10):
        yield [image_batch]

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8

quantized_model = converter.convert()

QUANTIZED_MODEL_PATH = f"{FOLDER}/models/quantized.tflite"
with open(QUANTIZED_MODEL_PATH, "wb") as f:
    f.write(quantized_model)
```

Figura 4.16: Código Python para cuantizar un modelo previamente importado con TFLite.

4.3.3. Benchmarking

La medición del rendimiento es una parte muy relevante de MicroTVM, ya que permite comparar los resultados de compilación con los de otras herramientas similares. También es esencial para comparar el rendimiento sobre distintas arquitecturas, como se ha hecho en este trabajo. Esta funcionalidad es la empleada para analizar la adaptabilidad de MicroTVM a nuevas arquitecturas hardware.

MicroTVM no dispone de una implementación propia que permita monitorizar el rendimiento de los modelos sobre los dispositivos en que se ejecutan. Sin embargo, sí dispone de soporte para su integración con el software de EEMBC. Esta herramienta facilita la ejecución del benchmark MLPerf™ Tiny, que proporciona una serie de modelos preentrenados junto a un conjunto de datos de prueba. La información relativa al benchmark se puede encontrar en https://github.com/mlcommons/tiny.

Para realizar las pruebas de rendimiento, se debe crear un proyecto Zephyr con MicroTVM que contenga el resultado de compilación de uno de los modelos proporcionados por MLPerf™ Tiny. Además, se debe indicar a MicroTVM que este proyecto debe soportar las características necesarias para medir el rendimiento de forma remota. Esto se hace indicando el tipo de proyecto como *mlperftiny*, tal y como se puede ver en la Figura 4.17. El resto de pasos son similares a los ya mostrados en las anteriores secciones.

```
project_options = {
    "extra_files_tar": str(extra_tar_file),
    "project_type": "mlperftiny",
    "board": BOARD,
    "compile_definitions": [
        f"-DWORKSPACE_SIZE={workspace_size + 512}",
        f"-DTARGET_MODEL={MODEL_INDEX}",
        f"-DTH_MODEL_VERSION=EE_MODEL_VERSION_{MODEL_SHORT_NAME}01",
        f"-DMAX_DB_INPUT_SIZE={input_total_size}",
    ],
}
```

Figura 4.17: Código Python con la configuración necesaria para incluir soporte al benchmark de MLPerf™ Tiny.

Además, tras compilar y ejecutar el proyecto generado de Zephyr, es necesario preparar y configurar el conjunto de datos de entrada para que los pueda procesar el EEMBCs EnergyRunner®. Las instrucciones completas se encuentran en el repositorio de GitHub de la herramienta, localizado en https://github.com/eembc/energyrunner. Siguiendo estas instrucciones, podemos realizar las siguientes pruebas:

- *Setup*: Realización de una prueba de rendimiento con un único caso de entrada. En este modo podemos configurar el número de iteraciones de *calentamiento* y el número de repeticiones de la medición. Como resultado obtendremos el tiempo medio que tarda en devolver el modelo una salida para esa entrada.
- *Median Performance*: Ejecución completa de 5 casos aleatorios del conjunto de datos de entrada disponible. Se obtienen como resultado los datos de rendimiento de estos casos.
- *Accuracy*: Proporciona la tasa de éxito del modelo tras ejecutarlo sobre el conjunto de datos de entrada completo. El resultado debe ser superior al indicado por el benchmark de MLPerf™ Tiny para que las mediciones de las anteriores opciones sea considerado válido. La ejecución de esta opción toma una cantidad de tiempo sustancialmente superior a las otras opciones.

En el capítulo 5 se exponen los resultados de utilizar estas funcionalidades sobre el hardware seleccionado para este proyecto.

CAPÍTULO 5

RESULTADOS

En este capítulo se evalúa el rendimiento que ofrece MicroTVM sobre arquitecturas hardware novedosas. Con este análisis se pretende conocer la idoneidad de la herramienta para casos de uso en los que la explotación de características arquitectónicas innovadoras del hardware sea una necesidad. Para ello, se ha hecho uso del benchmark MLPerf™ Tiny y la herramienta de monitorización EEMBCs EnergyRunner®, ya expuestos anteriormente. En esta evaluación se emplearon la versión 14 de LLVM y la versión 17 de TVM, con la configuración por defecto de la herramienta de monitorización (1 inferencia de calentamiento y 10 inferencias medidas por cada caso de entrada) en el modo de *Median Performance*. De los modelos ofrecidos por el benchmark para evaluar el rendimiento, se ha hecho uso de los siguientes:

- Keyword Spotting (KWS): Modelo de detección de palabras en archivos de audio. Su principal aplicación es la de, bajo un bajo consumo de energía, activar un programa o sistema de mayor consumo cuando se detecta la palabra predefinida. El ejemplo más común es el de los asistentes de voz que se activan bajo un comando concreto predefinido.
- Visual Wake Word (VWW): Modelo capaz de detectar si existe alguna persona presente en una imagen o no. Permite, al igual que el modelo de Keyword Spotting, activar un sistema más complejo en caso de que se produzca la detección de una persona (o cualquier otro objeto con el que se haya entrenado el modelo). Un ejemplo de caso de uso de este modelo es la activación de una cámara de seguridad cuando se detecta una persona en la imagen.
- Image Classification (IC): Este modelo aborda uno de los problemas más populares en el mundo de la visión artificial, el de clasificar las imágenes en categorías según

el objeto que se encuentra en ellas. Su popularidad permite comparar los resultados con los obtenidos en otros contextos de uso, que no tienen por qué restringirse al de los microcontroladores.

Con el uso de modelos de aprendizaje automático sobre microcontroladores de bajo consumo se pretende dotar a los dispositivos de IoT de la capacidad de entender y reaccionar ante el entorno que les rodea. Para ello, es esencial que los modelos cuenten con las siguientes dos características: precisión y rapidez. En la prueba aquí realizada, la precisión está garantizada gracias al uso de modelos preentrenados, que solo se considera que se están ejecutando correctamente si superan una tasa de acierto mínima predefinida. Por tanto, el foco de estas pruebas se centra en la rapidez, medida a través del número de inferencias por segundo de computación.

En esta línea, MicroTVM cuenta con un parámetro denominado *opt_level* que permite definir el nivel de optimización que se quiere aplicar en el proceso de compilación del modelo. De esta manera, una optimización internamente definida en MicroTVM sólo se aplica si su *opt_level* es menor o igual al establecido en el contexto de compilación. Por tanto, un nivel mayor de optimización definido en el contexto de compilación implica la aplicación de un mayor número de optimizaciones sobre el modelo.

Esta evaluación se ha ejecutado sobre dos dispositivos hardware cuya principal diferencia es la presencia de un microcontrolador de diferente versión. Los dispositivos son el STM32 Nucleo-144 y el Renesas EKRA8M1, cuyos microcontroladores son el ARM Cortex-M7 y el ARM Cortex-M85 respectivamente. La evaluación de ambos dispositivos se ha automatizado mediante el script de Python ubicado en el siguiente repositorio de GitHub: https://github.com/Jaimepas77/TFM/blob/master/8_MicroTVM_benchmark/example.py. En este script se han variado las variables relativas al hardware, el modelo y el nivel de optimización (*opt_level*) según correspondiera.

5.1. STM32 Nucleo-144

Los resultados obtenidos sobre este dispositivo sirven como punto de referencia, dado que hace uso de una arquitectura hardware popular y ampliamente soportada. Como consecuencia, es de esperar que el código generado para ella aproveche todas sus características, especialmente en cuanto a instrucciones específicas se refiere. Además, aunque MicroTVM no soporta de forma predeterminada este modelo en concreto, sí que soporta otros similares de la misma marca, por lo que la integración ha sido más sencilla. A continuación, se muestran los resultados obtenidos tras ejecutar cada modelo con los distintos niveles de optimización disponibles.

5.1.1. Keyword Spotting

La Figura 5.1 muestra los resultados de ejecutar el modelo de Keyword Spotting sobre el STM32 Nucleo-144, demostrándose que este es el modelo más rápido de los tres evaluados. Como se puede observar, los resultados son muy similares entre los distintos niveles de optimización, llegando a disminuir el rendimiento conforme se aplican más transformaciones al modelo. Esto resulta sorprendente, aunque no supone un gran inconveniente debido a que el empeoramiento es mínimo. Sin embargo, es un detalle a tener en cuenta de cara al despliegue de un modelo con MicroTVM. En la Tabla 5.1 se muestran las variaciones concretas de cada nivel de optimización con respecto del anterior.

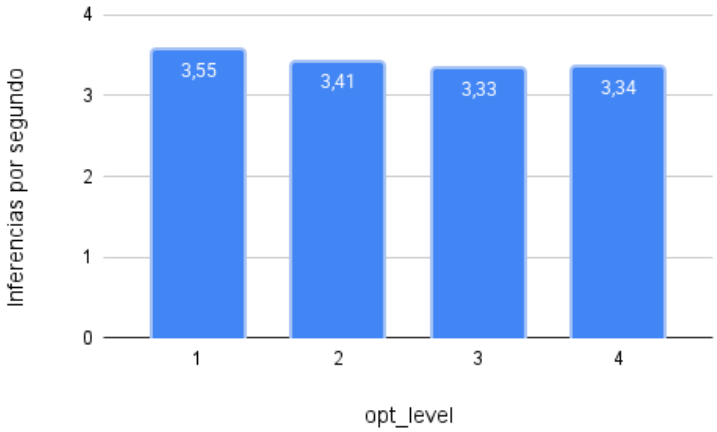

Figura 5.1: Resultados para el modelo KWS sobre el dispositivo STM32 Nucleo-144

	opt_level			
	1	**2**	**3**	**4**
Inf./segundo	3,551	3,411	3,330	3,340
Incremento del rendimiento	-	-3,94 %	-2,37 %	0,30 %

Tabla 5.1: Variación del rendimiento del modelo KWS sobre el dispositivo STM32 Nucleo-144

5.1.2. Visual Wake Word

El modelo de Visual Wake Word sobre el dispositivo de STMicroelectronics obtiene un rendimiento cercano a una inferencia por segundo, como se puede observar en la Figura 5.2. Al igual que en el caso anterior, su rendimiento es muy similar independientemente del nivel de optimización aplicado en MicroTVM, aunque en este caso el rendimiento mejora ligeramente conforme se aumenta el número de optimizaciones aplicadas. Esto se puede observar con mayor nivel de detalle en la Tabla 5.2.

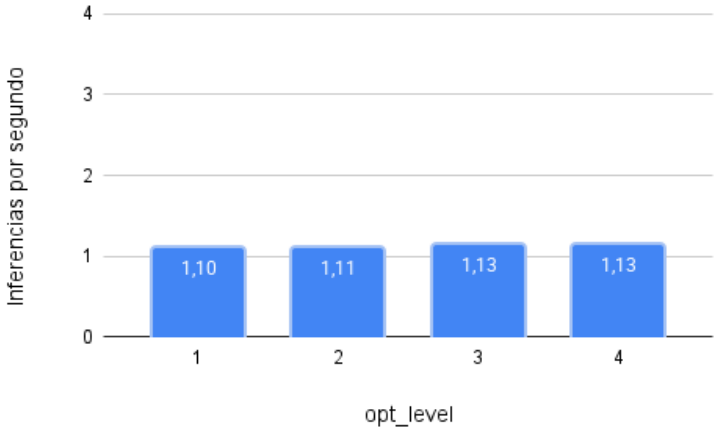

Figura 5.2: Resultados para el modelo VWW sobre el dispositivo STM32 Nucleo-144

	opt_level			
	1	**2**	**3**	**4**
Inf./segundo	1,098	1,105	1,134	1,134
Incremento de rendimiento	-	0,64 %	2,62 %	0,00 %

Tabla 5.2: Variación del rendimiento del modelo VWW sobre el dispositivo STM32 Nucleo-144

5.1.3. Image Classification

Al igual que en los anteriores casos, los datos de rendimiento, mostrados en la Figura 5.3, demuestran que el parámetro de optimización del modelo apenas varía el comportamiento del mismo. De hecho, se vuelve a observar que el mejor valor es el asociado con un menor número de optimizaciones aplicadas. Pueden observarse con mayor detalle las variaciones en la Tabla 5.3.

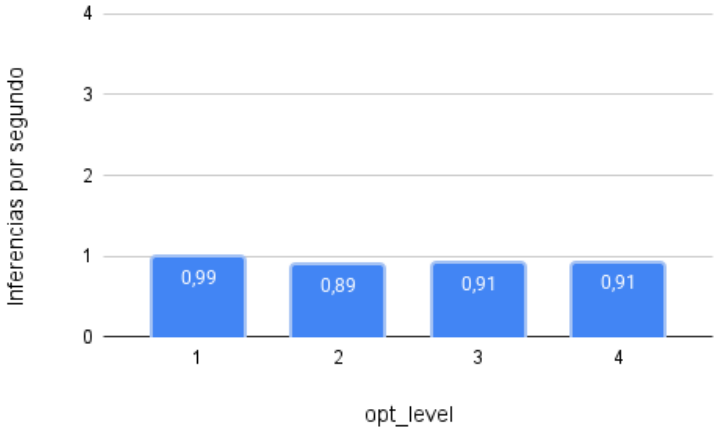

Figura 5.3: Resultados para el modelo IC sobre el dispositivo STM32 Nucleo-144

	opt_level			
	1	**2**	**3**	**4**
Inf./segundo	0,987	0,892	0,911	0,914
Incremento de rendimiento	-	-9,63 %	2,13 %	0,33 %

Tabla 5.3: Variación del rendimiento del modelo IC sobre el dispositivo STM32 Nucleo-144

5.2. Renesas EKRA8M1

En el caso del dispositivo de Renesas empleado, los resultados son peores a pesar de haber empleado una arquitectura más moderna con un conjunto de instrucciones más completo y eficiente. Este empeoramiento es, muy probablemente, la consecuencia de no haber podido emplear un compilador adaptado a la arquitectura ARM Cortex-M85. Como alternativa viable, aunque subóptima, se ha empleado el compilador de LLVM habitual, que no soporta las instrucciones vectoriales ARM Helium que incorpora el ARM Cortex-M85. De corregirse en un futuro el problema de integración de la variante de LLVM que sí soporta estas instrucciones, es de esperar una mejora significativa en los resultados de rendimiento respecto a los aquí mostrados.

5.2.1. Keyword Spotting

En la Figura 5.4 se muestra el resultado del modelo de Keyword Spotting con el dispositivo de Renesas. Como se ha mencionado ya, es peor que el del anterior dispositivo a pesar de usar una arquitectura hardware más moderna. Se observan unas fluctuaciones en el rendimiento mínimas, de forma similar al caso del otro dispositivo. Estas fluctuaciones se pueden analizar en detalle en la Tabla 5.4.

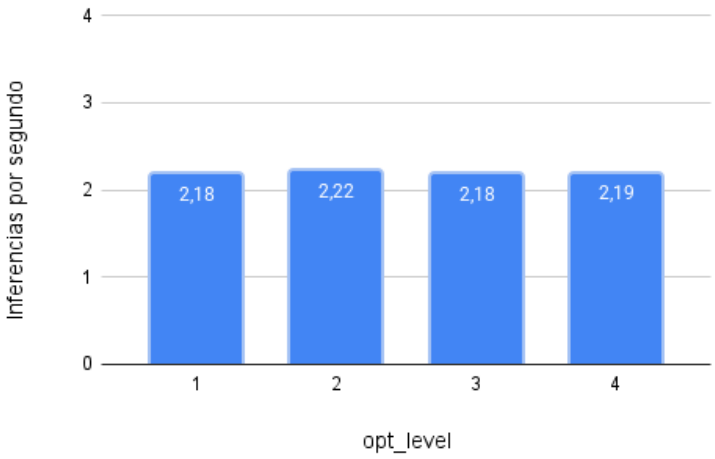

Figura 5.4: Resultados para el modelo KWS sobre el dispositivo Renesas EK-RA8M1

	opt_level			
	1	**2**	**3**	**4**
Inf./segundo	2,182	2,215	2,178	2,187
Incremento de rendimiento	-	1,51 %	-1,67 %	0,41 %

Tabla 5.4: Variación del rendimiento del modelo KWS sobre el dispositivo Renesas EK-RA8M1

5.2.2. Visual Wake Word

La Figura 5.5 muestra los resultados del modelo sobre el Renesas EKRA8M1. Su rendimiento es ligeramente inferior al del dispositivo de STMicroelectronics. Al igual que en anteriores casos, el parámetro *opt_level* apenas afecta al resultado final, como se aprecia en la Tabla 5.5.

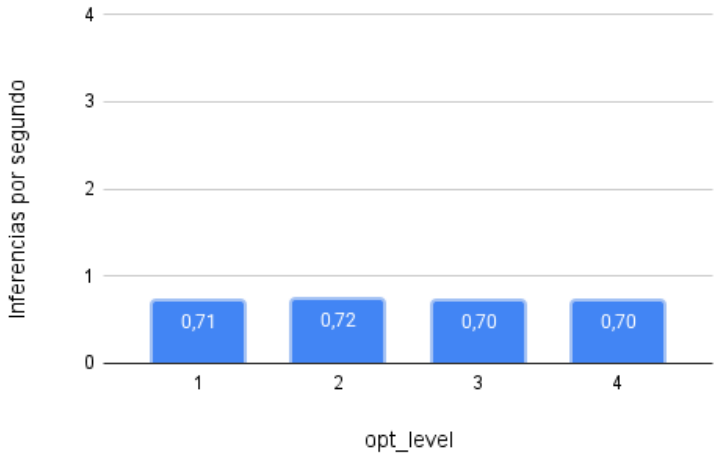

Figura 5.5: Resultados para el modelo VWW sobre el dispositivo Renesas EK-RA8M1

	opt_level			
	1	**2**	**3**	**4**
Inf./segundo	0,708	0,716	0,702	0,702
Incremento de rendimiento	-	1,13 %	-1,96 %	0,00 %

Tabla 5.5: Variación del rendimiento del modelo VWW sobre el dispositivo Renesas EK-RA8M1

5.2.3. Image Classification

Los resultados de este modelo se muestran en la Figura 5.6, que se encuentran dentro de lo esperable a la vista de los anteriores resultados vistos. De igual manera, se corrobora una

fluctuación mínima entre niveles de optimización en términos de rendimiento. Los datos se dan con mayor detalle en la Tabla 5.6.

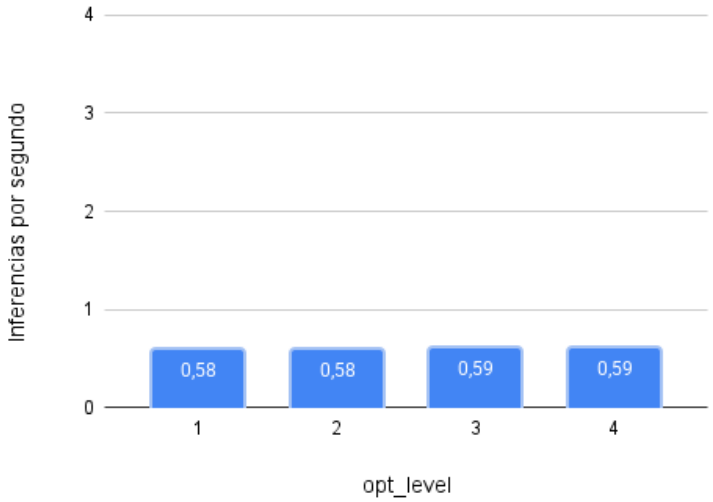

Figura 5.6: Resultados para el modelo IC sobre el dispositivo Renesas EK-RA8M1

	opt_level			
	1	**2**	**3**	**4**
Inf./segundo	0,578	0,579	0,594	0,593
Incremento de rendimiento	-	0,17 %	2,59 %	-0,17 %

Tabla 5.6: Variación del rendimiento del modelo IC sobre el dispositivo Renesas EK-RA8M1

5.3. Comparación entre los dispositivos

Como ya se ha podido observar en las secciones anteriores, el rendimiento ha sido mejor para el dispositivo de arquitectura hardware más antigua, lo cual resulta contraintuitivo. Esta diferencia de rendimiento que se mantiene en todos los modelos probados se puede apreciar con mayor detalle en la Figura 5.7, que muestra los mejores resultados obtenidos para cada modelo en cada dispositivo enfrentados sobre un gráfico de barras. Se ve claramente que el dispositivo de Renesas obtiene en estas condiciones un rendimiento menor, siendo su diferencia de rendimiento proporcionalmente similar en los tres casos. Esta proporcionalidad mantenida en las diferencias obtenidas se puede corroborar de forma más precisa con los datos proporcionados en la Tabla 5.7.

A pesar de las diferencias, esto no significa que el dispositivo de Renesas, basado en una arquitectura ARM Cortex-M85, tenga un peor rendimiento general. Es importante resaltar que la compilación que se ha realizado para este dispositivo ha estado limitada por el software empleado y es subóptima. Sin embargo, esto no asegura que, de haberse

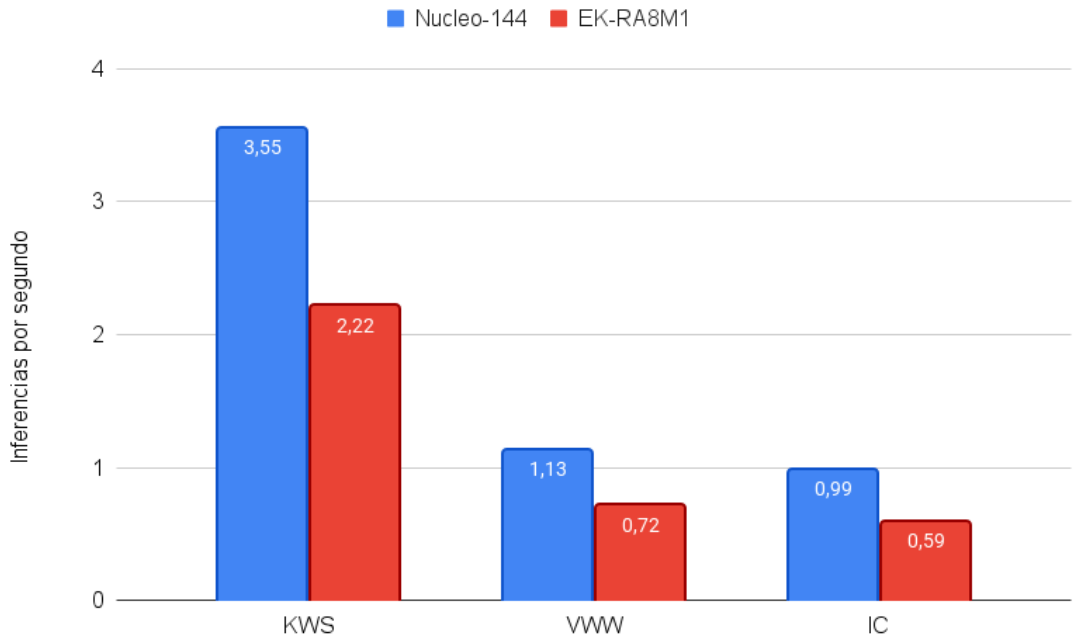

Figura 5.7: Comparativa de resultados de rendimiento por modelos entre dispositivos. Se toma la mejor medida obtenida en cada par dispositivo-modelo.

	Nucleo-144	**EK-RA8M1**	**Variación**
KWS	3,551	2,215	-37,62 %
VWW	1,134	0,716	-36,86 %
IC	0,987	0,594	-39,82 %

Tabla 5.7: Variación de los resultados de rendimiento entre dispositivos por modelos. Se toma la mejor medida de inferencias por segundo obtenida en cada par dispositivo-modelo.

empleado un compilador adaptado de forma óptima, se hubieran mejorado los resultados, dado que hay muchos factores que pueden afectar al rendimiento del microcontrolador. Un ejemplo de estos factores adicionales podría ser la latencia de acceso a memoria.

CAPÍTULO 6

CONCLUSIONES Y TRABAJO FUTURO

Durante el desarrollo de este proyecto se han visto los desafíos y oportunidades que plantea el uso de MicroTVM como compilador de modelos de IA. Las oportunidades son claras y están claramente expuestas en las fuentes ofrecidas por la comunidad entorno a TVM: se trata de un compilador de código abierto, altamente modular y flexible, que tiene el potencial de resolver los desafíos actuales en relación al mundo del aprendizaje automático en dispositivos de IoT. Sin embargo, aún no es un proyecto completamente maduro, con lo que enfrenta serias limitaciones entre las que se encuentran las acontecidas en el desarrollo de este trabajo. Además, la reciente pérdida de interés de la comunidad que mantiene TVM y MicroTVM plantea serias dudas sobre su utilidad como pieza clave en proyectos a largo plazo. Esto es porque, de mantenerse la falta de mantenimiento, los nuevos dispositivos que incorporen mejoras significativas en el hardware no serán compatibles con la herramienta, e incluso si lo fueran, los errores que pudieran detectarse en ella no serían solucionados.

Como se ha podido demostrar, MicroTVM es altamente flexible, permitiendo la integración de nuevos dispositivos y compiladores al tiempo que mantiene su funcionalidad completa. Esto se logra gracias al enfoque modular, en el cual se separa la preparación del entorno de ejecución (en este caso, Zephyr) de la compilación del propio modelo de IA. A pesar de ello, dispone de una curva de aprendizaje pronunciada, especialmente en el caso de que se requiera de una modificación en el flujo de compilación estándar. Esta limitación se ha reflejado en este caso en la complejidad de configuración de los dispositivos empleados y en la imposibilidad de integrar un compilador novedoso adaptado a la arquitectura de destino de uno de ellos.

En cuanto a las arquitecturas de los microcontroladores, se puede presuponer de los resultados de rendimiento comparativos que el uso de compiladores adaptados a la arquitectura de destino es crucial. Aunque queda como trabajo futuro la comparación del rendimiento con un compilador en tales condiciones, el hecho de que, a pesar de usar una arquitectura más moderna, se pierda eficiencia en el procesamiento computacional refuerza esta hipótesis sobremanera.

En lo referente a las diferencias de rendimiento entre modelos, es evidente que la arquitectura y el tipo de datos procesados afectan a su velocidad de inferencia. Se puede intuir que, en términos generales, el procesamiento de imágenes requiere de un tiempo de inferencia mayor que el del audio. Además, es relevante destacar que las diferencias de rendimiento entre los modelos se mantienen en las mismas proporciones en los dos dispositivos empleados. Por tanto, es muy probable que la diferencia de rendimiento entre modelos se deba a la complejidad del mismo.

Como trabajo futuro, se abren varias líneas de investigación posibles. En primer lugar, para evaluar el potencial de la herramienta, convendría realizar un análisis comparativo del rendimiento con respecto a otras herramientas de similar propósito (TensorFlow Lite, PyTorch Glow y similares). Aunque algunos estudios sugieren un rendimiento comparable al de TensorFlow Lite Micro (Hasanpour et al., 2025), o identifican cuellos de botella específicos como la latencia adicional por operaciones de *reshape* (Sponner et al., 2025), una evaluación exhaustiva es esencial para poder entender sus puntos fuertes. De esta manera, podrían evaluarse de mejor forma los potenciales beneficios de invertir recursos en su mantenimiento. También, es de especial interés conocer qué casos de uso de esta herramienta se están dando en el entorno empresarial, sobre todo al haber varias empresas implicadas en la comunidad que mantiene TVM. Por último, pero no menos importante, de tenerse los conocimientos y recursos adecuados, el estudio y resolución de los problemas actuales de MicroTVM, así como su modernización e integración con las nuevas características de TVM, resultarían esenciales para asegurar la relevancia del proyecto a largo plazo.

CHAPTER 7

INTRODUCTION

In this first chapter, the current context that justifies the realization of this work is presented. The specific objectives pursued are then outlined, along with the planning developed to achieve them. The aim is to provide a clear view of the adopted approach, highlighting both the relevance of the addressed problem and the intended contribution of this study.

7.1. Motivation

In recent years, we have witnessed a true explosion in the development and application of artificial intelligence (AI) techniques. These advances, especially in the field of machine learning and deep learning, have led to the emergence of intelligent applications in a wide variety of sectors, ranging from industry and medicine to transportation and entertainment.

At the same time, the proliferation of connected devices has consolidated the Internet of Things (IoT) paradigm, in which small devices with computational capabilities can interact with their environment, collect data, and make decisions in real time. These devices, often based on microcontrollers, are commonly found in environments such as smart traffic light cameras, motion detection sensors in alarm systems, or vehicles with autonomous driving features.

To meet the performance and energy efficiency demands of these applications, new hardware architectures optimized for embedded environments have emerged. However, this architectural diversity poses a significant challenge: adapting and optimizing software to fully exploit the capabilities of each platform. In this context, automatic generation of optimized code has become a necessity, especially in resource-constrained scenarios such as microcontrollers.

It is in this setting that MicroTVM (TVM, 2024a) emerges as an extension of the TVM compiler (Chen et al., 2018), aimed at resource-constrained devices. MicroTVM provides a framework for generating optimized code for various hardware architectures, with the goal of facilitating the efficient deployment of machine learning models on these devices. Nonetheless, given how recent these tools are and their close connection to constantly evolving hardware, it is necessary to evaluate their maturity, flexibility, and performance in real-world contexts.

This work is framed within this changing technological landscape and aims to contribute to the study of tools that enable better integration of artificial intelligence into IoT devices, with a particular focus on leveraging MicroTVM as a solution for the efficient compilation of models on microcontroller-based platforms.

7.2. Objectives

The aim of this work is to analyze the potential of MicroTVM as a code generation tool for hardware architectures based on microcontrollers. To achieve this, the following specific objectives have been established:

- Analysis of the existing capabilities in TVM, as well as those added by MicroTVM in the context of microcontrollers.
- Study of the available compilation methods, considering the possibility of integrating different compilers and configuration parameters that may affect the final result.
- Comparative evaluation of performance across different hardware architectures, obtaining real metrics using independent benchmarking tools.

7.3. Work Plan

To achieve the proposed objectives, work began with the expectation of completing the project within one year. Within this timeframe, the necessary tasks were carried out and completed as planned. The work has been structured over time as follows:

1. Training in the use of the project's tools. During the first few months, the available documentation was studied and all official tutorials on the main tool were completed.
2. Experimentation with the tool, studying and applying real and innovative use cases on the provided hardware. Over approximately four months, various configurations of the tool were tested on hardware not officially supported. This required modifying the tool's internal code and creating Python scripts to provide the necessary adaptability.

3. Comparative analysis of results and report writing. In the final three months, performance data was collected and analyzed for the tool across various devices relevant to the IoT context. The project report was also written in parallel.

To meet the schedule and overcome technical limitations, it was necessary to pivot in terms of the hardware used and the initial objectives. In any case, this contingency was managed without major issues, as it was among the expected scenarios when working with novel hardware that is not yet fully supported by existing tools.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

During the development of this project, the challenges and opportunities associated with the use of MicroTVM as a compiler for AI models have been explored. The opportunities are clear and well-documented in the resources provided by the TVM community: it is an open-source, highly modular and flexible compiler that has the potential to address current challenges related to machine learning on IoT devices. However, it is not yet a fully mature project and still faces serious limitations, some of which were encountered during this work. Moreover, the recent decline in interest from the community maintaining TVM and MicroTVM raises serious concerns about its viability as a key component in long-term projects. If the lack of maintenance persists, new devices incorporating significant hardware improvements may not be compatible with the tool, and even if they are, any detected bugs may not be resolved.

As demonstrated in this work, MicroTVM is highly flexible, allowing the integration of new devices and compilers while maintaining full functionality. This is achieved through its modular approach, which separates the preparation of the runtime environment (in this case, Zephyr) from the compilation of the AI model itself. Nonetheless, it presents a steep learning curve, particularly when modifications to the standard compilation flow are required. This limitation has been evident in this project in the complexity of configuring the target devices and the inability to integrate a new compiler tailored to the architecture of one of them.

Regarding microcontroller architectures, the comparative performance results suggest that using compilers tailored to the target architecture is crucial. Although comparing performance with such a compiler remains a task for future work, the fact that a more modern architecture led to reduced computational efficiency strongly reinforces this hypothesis.

As for performance differences between models, it is clear that both architecture and

the type of data being processed affect inference speed. It can generally be inferred that image processing requires more inference time than audio processing. Furthermore, it is noteworthy that the performance differences between models remain proportionally consistent across the two devices used. Therefore, it is highly likely that the performance gap between models is due to their inherent complexity.

Several possible lines of future research emerge from this work. First, to properly assess the tool's potential, it would be beneficial to conduct a comparative performance analysis against other tools with similar purposes (such as TensorFlow Lite, PyTorch Glow, and others). Although some studies suggest performance comparable to TensorFlow Lite Micro (Hasanpour et al., 2025), or identify specific bottlenecks such as additional latency from reshape operations (Sponner et al., 2025), a comprehensive evaluation is essential to understand its strengths. This would help determine the potential benefits of investing resources in its continued maintenance. Additionally, it would be especially interesting to investigate the current use cases of this tool in industry, particularly given that several companies are involved in the TVM community. Finally, and perhaps most importantly, if the necessary knowledge and resources are available, studying and solving MicroTVM's current issues—as well as modernizing it and integrating it with new TVM features—would be essential to ensure the long-term relevance of the project.

BIBLIOGRAFÍA

ARM. https://developer.arm.com/documentation/102102/0102/What-is-Helium-, 2025a.

ARM. https://developer.arm.com/Processors/Cortex-M85, 2025b.

Banbury, C., Reddi, V. J., Torelli, P., Holleman, J., Jeffries, N., Kiraly, C., Montino, P., Kanter, D., Ahmed, S., Pau, D. et al. Mlperf tiny benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C. y Krishnamurthy, A. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *arXiv e-prints*, página arXiv:1802.04799, 2018.

Chen, Z., Yu, C. H., Morris, T., Tuyls, J., Lai, Y.-H., Roesch, J., Delaye, E., Sharma, V. y Wang, Y. Bring your own codegen to deep learning compiler. 2021.

David, R., Duke, J., Jain, A., Janapa Reddi, V., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., Wang, T., Warden, P. y Rhodes, R. Tensorflow lite micro: Embedded machine learning for tinyml systems. En *Proceedings of Machine Learning and Systems* (editado por A. Smola, A. Dimakis y I. Stoica), vol. 3, páginas 800–811. 2021.

EEMBC. https://github.com/eembc/energyrunner/tree/main?tab=readme-ov-file, 2025.

Hasanpour, M. A., Kirkegaard, M. y Fafoutis, X. Edgemark: An automation and benchmarking system for embedded artificial intelligence tools. 2025.

Ikarashi, Y., Bernstein, G. L., Reinking, A., Genc, H. y Ragan-Kelley, J. Exocompilation for productive programming of hardware accelerators. En *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, página 703–718. Association for Computing Machinery, New York, NY, USA, 2022. ISBN 9781450392655.

Lattner, C. https://aosabook.org/en/v1/llvm.html, 2025.

Lattner, C. y Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. páginas 75–88. San Jose, CA, USA, 2004.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N. y Zinenko, O. MLIR: Scaling compiler infrastructure for domain specific computation. En *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, páginas 2–14. 2021.

LLVM. https://llvm.org/docs/DeveloperPolicy.html#new-llvm-project-license-framework, 2025a.

LLVM. https://github.com/ARM-software/LLVM-embedded-toolchain-for-Arm, 2025b.

NVIDIA. https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf, 2025.

OpenVino. https://github.com/openvinotoolkit/openvino, 2025.

Ortiz, L. A. P., Soliz, I. F. M. y Balarezo, V. K. G. Advancing tinyml in iot: A holistic system-level perspective for resource-constrained ai. *Preprints*, 2025.

Project, Z. https://www.zephyrproject.org/, 2025a.

Project, Z. https://www.zephyrproject.org/learn-about/, 2025b.

Project, Z. https://docs.zephyrproject.org/latest/develop/west/index.html, 2025c.

PyTorch. https://github.com/pytorch/glow, 2025.

QEMU. https://www.qemu.org/, 2025.

Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S. y Durand, F. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, vol. 61(1), página 106–115, 2017. ISSN 0001-0782.

Rath, D. https://openocd.org/files/thesis.pdf, 2025.

Renesas. https://www.renesas.com/en/products/microcontrollers-microprocessors/ra-cortex-m-mcus/ek-ra8m1-evaluation-kit-ra8m1-mcu-group, 2025a.

Renesas. https://www.renesas.com/en/products/microcontrollers-microprocessors/ra-cortex-m-mcus/ra8m1-480mhz-arm-cortex-m85-based-microcontroller-helium-and-trustzone, 2025b.

Renesas. https://www.segger.com/downloads/jlink/, 2025c.

Shiue, J., Hoge, C. y Zheng, L. https://tvm.apache.org/docs/v0.16.0/tutorial/introduction.html, 2024.

Sponner, M., Servadei, L., Waschneck, B., Wille, R. y Kumar, A. Peax - a model augmentation framework for adaptive techniques and embedded applications. En *Proceedings of the 2025 Workshop on Compilers, Deployment, and Tooling for Edge AI*, CODAI '25. ACM, 2025.

STMicroelectronics. https://www.st.com/en/evaluation-tools/stm32-nucleo-boards.html, 2025a.

STMicroelectronics. https://www.st.com/en/microcontrollers-microprocessors/stm32h743zi.html, 2025b.

STMicroelectronics. https://www.st.com/en/development-tools/stm32cubeprog.html, 2025c.

TensorFlow. https://ai.google.dev/edge/litert, 2025.

TVM. https://tvm.apache.org/docs/v0.16.0/install/from_source.html#install-from-source, 2025a.

TVM. https://github.com/apache/tvm/tree/v0.17.0, 2025b.

TVM. https://tvm.apache.org/docs/v0.16.0/arch/model_library_format.html#model-library-format, 2025c.

TVM, A. https://tvm.apache.org/docs/v0.16.0/topic/microtvm/index.html, 2024a.

TVM, A. https://discuss.tvm.apache.org/t/questions-regarding-the-future-development-of-mi 17926, 2024b.

TVM, A. https://tvm.apache.org/community, 2025d.