UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Diseño de algoritmos eficientes para aprendizaje automático sobre MCUs

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Maciá Lillo, Antonio

Tutor/a: Alonso Jordá, Pedro

Cotutor/a: Quintana Ortí, Enrique Salvador

Cotutor/a: Castelló Gimeno, Adrián

CURSO ACADÉMICO: 2021/2022

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de sistemas informáticos y computación
Universitat Politècnica de València

Diseño de algoritmos eficientes para aprendizaje automático sobre MCUs

TRABAJO FIN DE MASTER

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Autor: Antonio Maciá Lillo

Tutor: Pedro Alonso Jordá
Enrique Salvador Quintana Ortí

Curso 2021-2022

Resum

Vivim en un món interconnectat, amb microcontroladors i processadors de consum ultra-reduït (MCUs), integrats dins de rellotges i electrodomèstics intel·ligents, assistents de veu, telèfons mòbils, i tot tipus de dispositius, capaços de recollir una quantitat de dades enorme i de gran valor. En aquest context, les empreses es troben embarcades avui dia en un procés d'evolució, des de la era del "big data" a la de l'anàlisi de la informació, emprant a molts casos tècniques d'aprenentatge automàtic (machine learning). Els dispositius de tipus MCU es caracteritzen per unes fortes restriccions quant a capacitat de la memòria, consum d'energia i suport del sistema operatiu, així com limitacions en el tipus d'aritmètica suportada i, especialment, per una enorme heterogeneïtat de dispositius. En aquest context, la utilització d'entorns generals de aprenentatge automàtic, com TensorFlow, PyTorch, CNTK, DSSTNE o Keras, és directament inviable o resulta enormement ineficient. En resposta a aquest escenari, l'objectiu general d'aquest projecte és el desenvolupament d'algoritmes, conscients de l'arquitectura, per accelerar les tècniques d'aprenentatge automàtic sobre MCUs. Com a objectius addicionals, es pretén contribuir a crear interfícies de programació estandarditzades per a eines i biblioteques d'aprenentatge automàtic per a MCUs, així com estudiar els beneficis dels MCUs en inferència usant aplicacions de rellevància.

Paraules clau: microcontroladors de consum ultra-reduit, procesadors de consum ultra-reduit, MCUs, aprenentatge automàtic

Resumen

Vivimos en un mundo interconectado, con microcontroladores y procesadores de consumo ultra-reducido (MCUs), integrados dentro de relojes y electrodomésticos inteligentes, asistentes de voz, teléfonos móviles, y todo tipo de dispositivos, capaces de recoger una cantidad de datos enorme y de gran valor. En este contexto, las empresas se encuentran embarcadas hoy en día en un proceso de evolución, desde la era del "big data" a la del análisis de la información, empleando en muchos casos técnicas de aprendizaje automático (machine learning). Los dispositivos de tipo MCU se caracterizan por unas fuertes restricciones en cuanto a capacidad de la memoria, consumo de energía y soporte del sistema operativo, así como limitaciones en el tipo de aritmética soportada y, especialmente, por una enorme heterogeneidad de dispositivos. En este contexto, la utilización de entornos generales de aprendizaje automático, como TensorFlow, PyTorch, CNTK, DSSTNE o Keras, es directamente inviable o resulta enormemente ineficiente. En respuesta a este escenario, el objetivo general de este proyecto es el desarrollo de algoritmos, conscientes de la arquitectura, para acelerar las técnicas de aprendizaje automático sobre MCUs. Como objetivos adicionales, se pretende contribuir a crear interfaces de programación estandarizadas para herramientas y bibliotecas de aprendizaje automático para MCUs, así como estudiar los beneficios de los MCUs en inferencia usando aplicaciones de relevancia.

Palabras clave: microcontroladores de consumo ultra-reducido, procesadores de consumo ultra-reducido, MCUs, aprendizaje automático

Abstract

We live in an interconnected world, with microcontrollers and ultra-low power processors (MCUs), embedded within smart watches and appliances, voice assistants, phones, and all kinds of devices, capable of collecting a number of huge and valuable data. In this context, companies today are embarked on a process of evolution, from the era of "big

data" to that of information analysis, using in many cases machine learning techniques. The MCU-type devices are characterized by strong restrictions in terms of memory capacity, power consumption and operating system support, as well as limitations on the type of arithmetic supported and, especially, by an enormous heterogeneity of devices. In this context, the use of general environments of machine learning, such as TensorFlow, PyTorch, CNTK, DSSTNE, or Keras, is directly unfeasible or extremely inefficient. In response, the purpose of this project is the development of architecture-aware algorithms to speed up machine learning techniques on MCUs. Furthermore, it is intended to contribute to creating standardized programming interfaces for tools and libraries of machine learning for MCUs, as well as studying the benefits of MCUs in inference using relevant applications.

Key words: ultra-low power microcontrollers, ultra-low power processors, MCUs, machine learning

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

En este trabajo se estudiará la ejecución y el rendimiento de algoritmos de redes neuronales en dispositivos embebidos. En concreto, se pretende optimizar uno de los kernels más costosos computacionalmente en este tipo de algoritmos para una arquitectura concreta, y se realizarán pruebas para mostrar las bondades de las optimizaciones realizadas.

1.1 Motivación

La Inteligencia Artificial (IA) se puede definir como la ciencia e ingeniería de construir sistemas o máquinas que imitan la inteligencia humana para realizar tareas complejas [1].

El desarrollo del campo de la inteligencia artificial ha estado ligado al desarrollo de la computación desde los inicios de este último. En la década de los 50, científicos de diversos campos fueron definiendo lo que se conoce hoy como inteligencia artificial, con artículos como el que publicó Alan Turing en la revista Mind, en el que se describe lo que hoy se conoce como el experimento filosófico de la prueba de Turing [2]. Los primeros programas que se desarrollaron en este campo, estaban limitados por el tamaño y velocidad de la memoria, y la capacidad de cómputo de los ordenadores de la época. El evento que sentó las bases de la IA fue la conferencia de Dartmouth [3], en 1956. La tendencia después de esta conferencia fué desarrollar sistemas de IA basados en cómo funciona el pensamiento humano para resolver el problema. Estos sistemas utilizaban información simbólica como entrada, que es manipulada de acuerdo con un conjunto de reglas, y resuelve problemas de este modo [4].

En este contexto, se dieron los primeros pasos de lo que se conoce como aprendizaje automático. El origen del término se atribuye a Arthur Samuel, de IBM, que desarrolló un algoritmo capaz de aprender a jugar a las damas en 1959. No obstante, Frank Rosenblatt, en 1958, estudiando el funcionamiento del sistema nervioso, desarrolló un sistema llamado Perceptrón, que es capaz de reconocer las letras del alfabeto. Este prototipo sentó las bases del tipo de algoritmos de aprendizaje automático conocido como redes neuronales. Por otro lado, este primer modelo tiene el problema de que sólo funciona con problemas que son linealmente separables [5].

En 1975 Werbos desarrolla el primer perceptrón multicapa [5]. Después, en 1980, Kunihiko Fukushima propondrá la primera red neuronal convolucional, conocida como *"Neocognitron Fukushima"*. A esto le siguió la invención del algoritmo *"Backpropagation"* por Rumelhart en 1986. No obstante también se avanzó en otros campos del aprendizaje automático, como el desarrollo de los árboles de decisión por Quinlan, en 1986. En la década de los 90 destaca el desarrolo de las máquinas de vectores de soporte en 1997 por Cortes y Vapnik [6].

Desde comienzos del siglo XXI ha habido un creciente interés por algoritmos de aprendizaje automático. Una de las razones es la aparición del *Big Data*. El número de datos disponibles incrementó de manera considerable, por lo que se necesitaron nuevas maneras de procesar los datos. La segunda razón es la reducción del coste de la computación paralela y la ampliación de la cantidad disponible de memoria RAM. Debido a esto surgieron nuevos tipos de bases de datos, como las *NoSQL* [7], y frameworks dedicados al procesamiento distribuido de datos como *Apache Spark* [8]. La tercera y última razón es el desarrollo de nuevos algoritmos de aprendizaje automático, sobre todo los relacionados con Aprendizaje Profundo (*"Deep Learning"*) [9], que se compone de algoritmos de redes neuronales multicapa.

El aprendizaje automático es un conjunto de algoritmos computacionales de inteligencia artificial que aprenden del entorno que les rodea. Su área de aplicación incluye una gran variedad de campos como *Big Data*, reconocimiento de patrones, visión por computador, o finanzas.

El aprendizaje automático utiliza una colección de datos de entrada para conseguir la tarea deseada sin necesidad de ser programados explícitamente. Funcionan adaptando su arquitectura mediante la recepción de datos, de manera que consiguen mejores resultados paulatinamente. Este proceso de adaptación se denomina entrenamiento. En función del tipo de entrenamiento, hay tres tipos de aprendizaje automático, el supervisado, el no supervisado y el semi-supervisado.

En el entrenamiento de algoritmos de aprendizaje automático supervisado, muestras de el conjunto de datos de entrada se proporcionan junto con la salida esperada. El algoritmo entonces se configura a si mismo de manera que a parte de conseguir producir la salida deseada de los datos de entrenamiento, es capaz de generalizar y producir resultados correctos para el resto del conjunto de datos de entrada.

El aprendizaje automático no supervisado se diferencia del supervisado en que no existe una muestra del conjunto de datos previamente etiquetada, sino que el conocimiento se extrae directamente del conjunto de datos. En lugar de asociar una entrada concreta con una salida particular, este tipo de algoritmos se configuran directamente en función de los datos de entrada. El aprendizaje automático semi-supervisado es una combinación del supervisado y del no supervisado.

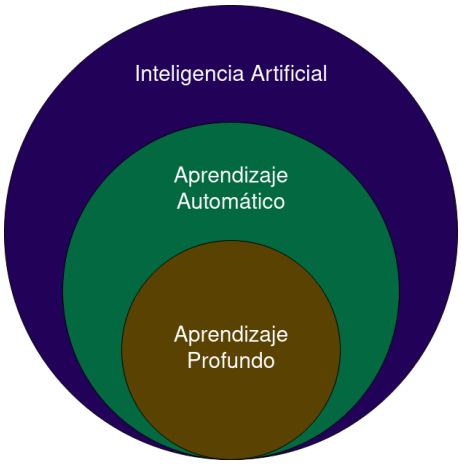


Figura 1.1: Jerarquía de los conceptos inteligencia artificial, aprendizaje automático y aprendizaje profundo.

Aunque hoy en día el aprendizaje profundo es el tipo de algoritmos de aprendizaje automático predominante, el aprendizaje automático es un campo que engloba a más

tipos de algoritmos como (a parte de los ya mencionados) Regresión Linear [10], Bosque Aleatorio [11], o *k-means* [12]. La Figura 1.1 muestra la jerarquía entre los conceptos de inteligencia artificial, aprendizaje automático y aprendizaje profundo.

El término Internet de las Cosas (IoT) hace referencia a escenarios donde la conectividad y capacidad de procesamiento se extiende a objetos, sensores, y, en general, dispositivos de uso diario que normalmente no se considerarían computadores. Estos dispositivos tienen la capacidad de generar, intercambiar, y utilizar datos con una mínima intervención humana [13].

Este término fué acuñado en 1999 por Kevin Ashton, para describir un sistema en el que objetos del mundo físico pueden estar conectados a internet mediante sensores. No obstante, aunque el concepto IoT es relativamente nuevo, el concepto de combinar computadores y redes para monitorizar y controlar dispositivos es conocido desde hace décadas.

La razón de que los dispositivos IoT hayan ganado popularidad recientemente se debe a un conjunto de tendencias tecnológicas y de mercado que han hecho posible interconectar dispositivos cada vez más baratos y accesibles. Entre estos avances destacan:

- Tecnologías para la conexión inalámbrica de alta velocidad y bajo coste.
- Amplia adopción del protocolo IP como el estándar global para la conexión de redes.
- Dispositivos de cada vez mayor capacidad de cómputo a precio más bajo.
- Avances en la fabricación han llevado a concebir dispositivos de tamaño pequeño que incorporan tecnologías innovadoras de computación y comunicaciones.
- Desarrollo de la Computación en la Nube (*"Cloud Computing"*), que permite disponer de recursos de computación de manera remota y bajo demanda.
- Avances en el campo del Análisis de Datos, que junto al incremento de potencia computacional, almacenamiento de datos, y servicios en la Nube, permite la agregación, correlación y análisis de grandes cantidades de datos.

Coincidiendo con el último punto, y debido al auge de los dispositivos IoT, y a su amplia adopción tanto en la industria como en el hogar, ha surgido una nueva corriente, llamada *"TinyML"*, que pretende ejecutar algoritmos de aprendizaje automático directamente sobre estos dispositivos, teniendo en cuenta las restricciones en términos de capacidad de cómputo y memoria disponible que presentan.

1.2 Objetivos

Como objetivo principal de este proyecto, se quiere optimizar la ejecución de algoritmos de aprendizaje automático de redes neuronales sobre un dispositivo concreto, el Arduino Nano 33 BLE. Este objetivo se enmarca dentro de la filosofía de *TinyML* de ejecutar algoritmos de aprendizaje automático sobre dispositivos embebidos.

En los objetivos específicos necesarios para alcanzar esta meta, se incluye un estudio previo sobre los algoritmos de aprendizaje profundo, y los kernels que intervienen, y cuales de estos son críticos para el rendimiento del algoritmo. Además se debe hacer un estudio de los algoritmos y librerías que provean la funcionalidad del kernel, para elegir la opción apropiada para la implementación.

También se realizará la implementación de rutinas basadas en la arquitectura concreta, para obtener el mayor rendimiento utilizando las capacidades disponibles en el procesador y las características de la estructura de la memoria.

Por último, se harán pruebas que confirmen el rendimiento del algoritmo desarrollado, así como de las optimizaciones realizadas. También se deben probar diferentes tipos de problema para comprobar el comportamiento del algoritmo en diferentes casos.

1.3 Estructura de la memoria

La memoria está estructurada en cuatro capítulos, estado del arte, implementación, experimentos y resultados, y conclusiones.

En el primer capítulo se presentarán las tecnologías involucradas en este trabajo. Se expondrán las tecnologías de *TinyML*, redes neuronales y la arquitectura del Arduino Nano 33 BLE. También serán estudiadas las investigaciones previas en las que se basa este trabajo, sobre la multiplicación de matrices general.

En el segundo capítulo será expuesta la implementación del algoritmo utilizado en este trabajo. Se procederá a estudiar el código que se ha implementado, y se mostrarán en detalle las diferentes optimizaciones que se han realizado.

El tercer capítulo trata de los diferentes experimentos que se han realizado y sus análisis, para mostrar las características del algoritmo desarrollado, y que parámetros y configuraciones son los más óptimos.

El cuarto y último capítulo está dedicado a explicar las conclusiones obtenidas de este trabajo.

CAPÍTULO 2
Estado del arte

La finalidad de este trabajo es la optimización de los kernels internos que utilizan las redes neuronales de Tensorflow Lite, sobre una arquitectura Arduino concreta. En este capítulo se explican brevemente las tecnologías involucradas en la ejecución de algoritmos de inteligencia artificial sobre dispositivos embebidos. También se exponen los detalles sobre la arquitectura utilizada, que serán útiles para entender las optimizaciones realizadas.

2.1 Redes Neuronales

Las redes neuronales son algoritmos de inteligencia artificial, originalmente inspirados en el funcionamiento de las redes neuronales biológicas que forman el cerebro humano. Pertenecen al campo de estudio del aprendizaje automático [5].

Las redes neuronales con más uso hoy en día son las Redes Neuronales Profundas (*"Deep Neurnal Networks"* o DNN). Estas redes se componen de capas de nodos, conectados entre sí. A la primera capa se le conoce como capa de entrada (*"input"*). A la última capa de la red se le conoce como capa de salida (*"output"*). A las capas intermedias se les conoce como capas ocultas (*"hidden"*). Cada uno de los nodos, también conocidos como neuronas, tienen un peso asociado y un sesgo. Utilizando las conexiones de entrada de la neurona, el peso y el sesgo, se calcula el valor resultado, utilizando la función de activación de la neurona, y se propaga el resultado a la siguiente capa. La Figura 2.1 muestra gráficamente esta estructura.

El uso de estos algoritmos de aprendizaje automático con redes neuronales se basa en dos fases. La primera fase es la de entrenamiento, donde se configura la DNN para que funcione como se desea. La segunda fase es la fase de inferencia, donde se utiliza realmente la DNN ya entrenada para su propósito.

Para entrenar la DNN se define una colección de vectores de entrada $x_1, x_2, ..., x_s \in \mathbb{R}^n$, con vectores de salida asociados $y_1, y_2, ..., y_s \in \mathbb{R}^n$. Matemáticamente, una DNN define una función no lineal $\mathcal{F} : \mathbb{R}^m \to \mathbb{R}^n$ de manera que $\mathcal{F}(x_r) = \tilde{y}_r$, donde se espera que $\tilde{y}_r \approx y_r$ [15].

Cuando se utiliza entrenamiento supervisado, la DNN se entrena utilizando datos previamente etiquetados, para reducir la diferencia (error) $\|y_r - \tilde{y}_r\|$ (para todas las salidas).

La primera parte del entrenamiento se conoce como *"forward pass"* (FP). En esta fase, la DNN recibe una entrada y produce una salida, y se calcula el error obtenido. En la segunda parte, se modifican los parámetros de la red para intentar reducir el error. Uno

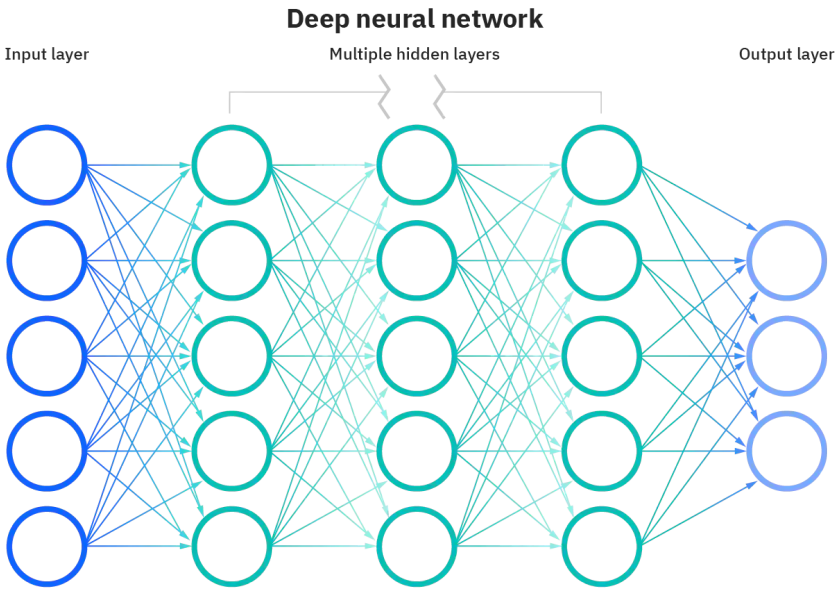


Figura 2.1: Diagrama de una red neuronal [14].

de los métodos utilizados para esto es Gradiente Estocástico Descendiente (SGD), que es un método iterativo de tipo *"back-propagation"* (BP). Este método calcula los gradientes de la DNN que minimizan el error, que a continuación se utilizan para actualizar los parámetros de la red.

El entrenamiento se realiza en ciclos de FP-BP, reduciendo el error en cada paso, hasta que idealmente se alcanza un error aceptable.

Una vez que se ha terminado la fase de entrenamiento, se puede utilizar la DNN para la aplicación deseada, realizando inferencias sobre esta. Estas inferencias consisten en la fase FP explicada anteriormente, donde se le pasa a la red una entrada y produce una salida.

La fase de inferencia es menos costosa computacionalmente que la fase de entrenamiento. No obstante, puede estar sometida a restricciones de tiempo real, de capacidad de cómputo o de consumo energético muy fuertes, tal como veremos a continuación.

2.2 Tiny ML

Tradicionalmente, en aprendizaje automático se ha trabajado con redes neuronales de gran tamaño. Para procesar estas redes es necesario el uso de computadores de Altas prestaciones y gran capacidad de almacenamiento.

TinyML cambia esta tendencia, y consiste en aplicar aprendizaje automático en dispositivos con características de computación, almacenamiento y consumo energético limitados. Según el libro de referencia sobre TinyML [16], hay consenso a nivel académico y en la industria en general para considerar como TinyML aquellos modelos de redes neuronales que pueden funcionar con un coste energético por debajo de 1 mW.

El tipo de hardware al que va destinado TinyML es, sobre todo, dispositivos embebidos. Estos suelen tener una memoria RAM del tamaño de kilobytes, y una velocidad de reloj del orden de megahercios. Con estas restricciones, estos dispositivos no suelen disponer de un sistema operativo. Además, en muchas ocasiones, no tienen memoria

dinámica, para evitar los problemas de estabilidad generados por la fragmentación del *heap* que ocurren a largo plazo. Otro de los retos que viene asociado al uso de estos dispositivos es que muchas veces no tienen la posibilidad de utilizar un *debugger*, ya que las interfaces que se utilizan para acceder al chip están muy especializadas.

Un dispositivo embebido es un sistema de computación que combina hardware y software, así como partes mecánicas o de otro tipo, diseñado para realizar una función específica. Están basados en microprocesadores o microcontroladores, donde la mayoría de componentes se encuentran incluidos en la placa base. Estos dispositivos generalmente deben ser de bajo coste y poseer un tamaño reducido, con suficiente rendimiento para el procesado de datos en tiempo real, y un consumo mínimo de energía [17]. Estas características hacen que se diseñen con fuertes restricciones de capacidad de cómputo y memoria.

Históricamente, estos dispositivos se han usado en sistemas de control y robótica. No obstante se están empezando a usar en dispositivos *"Internet of Things"* (IoT). Estos son dispositivos de cómputo no estándar, que se conectan a internet de manera inalámbrica, y pueden transmitir datos y comunicarse entre ellos.

La ventaja de utilizar algoritmos de inteligencia artificial sobre dispositivos IoT consiste en que tienen una gran cantidad de información disponible, ya que estos dispositivos se sitúan directamente sobre el medio en el que están trabajando. Transmitir todos estos datos a una puerta de enlace (*"edge"*) o a un servidor remoto para su procesado puede tener un coste en las comunicaciones prohibitivo.

La posibilidad de utilizar algoritmos de redes neuronales en dispositivos embebidos es bastante reciente, por lo que este hardware especializado así como la tecnología están en una rápida expansión en la actualidad.

TensorFlow es una biblioteca open source de aprendizaje automático desarrollada y mantenida por Google desde 2015. Esta biblioteca está destinada a entrenar y ejecutar modelos en los principales entornos de escritorio (Windows, MacOS, Linux), así como en servidores cloud que disponen de gigabytes de RAM, así como de terabytes de espacio de almacenamiento. La principal interfaz para el uso de esta biblioteca es Python, un lenguaje de programación y entorno de scripting disponible en la mayoría de los entornos de escritorio, y ampliamente usado en los servidores.

Todas estas características provocan que el tamaño del ejecutable resultante esté en el rango de los MB. Esto limitaba su uso en otro tipo de plataformas. No obstante, los primeros avances se hicieron cuando se empezó a utilizar TensorFlow en dispositivos móviles. El tamaño de una aplicación Android puede tener un impacto negativo en la cantidad de descargas y la satisfacción del cliente. Además, los teléfonos móviles tienen un procesador bastante menos potente que un ordenador convencional. Por esta razón se hicieron esfuerzos por reducir el tamaño y la carga computacional en lo que se acabó llamando TensorFlow Lite.

TensorFlow Lite es un proyecto hermano de TensorFlow que está orientado a la ejecución de modelos preentrenados de redes neuronales de manera eficiente en dispositivos móviles.

Debido a su uso específico en aplicaciones móviles, y aunque es posible realizar tareas de entrenamiento sobre el modelo, generalmente la biblioteca TensorFlow Lite se utiliza para modelos que han sido preentrenados en computadores mucho más potentes, con una gran cantidad de datos.

Más adelante, los ingenieros de Google se propusieron dar el paso a dispositivos embebidos, en lo que acabó siendo la biblioteca TensorFlow Lite Micro. Conociendo los retos

de trabajar con este tipo de dispositivos, esta biblioteca fue desarrollada con los siguientes requisitos en mente:

- Sin dependencias de sistema operativo.
- Las bibliotecas estándar de C o C++ no están disponibles.
- No se puede esperar soporte hardware para aritmética de coma flotante.
- No se utiliza memoria dinámica.

Debido a las restricciones de los dispositivos embebidos, la biblioteca TensorFlow Micro solo permite realizar inferencias sobre el modelo, y no permite realizar tareas de entrenamiento.

De esta manera se creó una biblioteca de TensorFlow de tamaño mínimo, con una implementación genérica para poder ejecutarse en los principales dispositivos embebidos. Además, esta biblioteca permite sustituir con facilidad los kernels genéricos, tales como la convolución, suma, multiplicación, entre otros, por kernels optimizados para la arquitectura específica donde se va a ejecutar.

2.3 Redes Neuronales Convolucionales

Dentro del paradigma del aprendizaje automático destacan el campo de las Redes Neuronales Convolucionales (CNN). Estas añaden capas de convolución para extraer características de las imágenes, que serán analizadas por capas tradicionales posteriores.

La utilidad del uso de filtros de convolución en las redes neuronales proviene de su habilidad de extraer características de los contornos de la imagen. Dependiendo del filtro utilizado, se resaltan ciertas características de la imagen. En la Figura 2.2 se muestran los efectos de utilizar diferentes filtros al realizar la operación de convolución sobre la misma imagen.

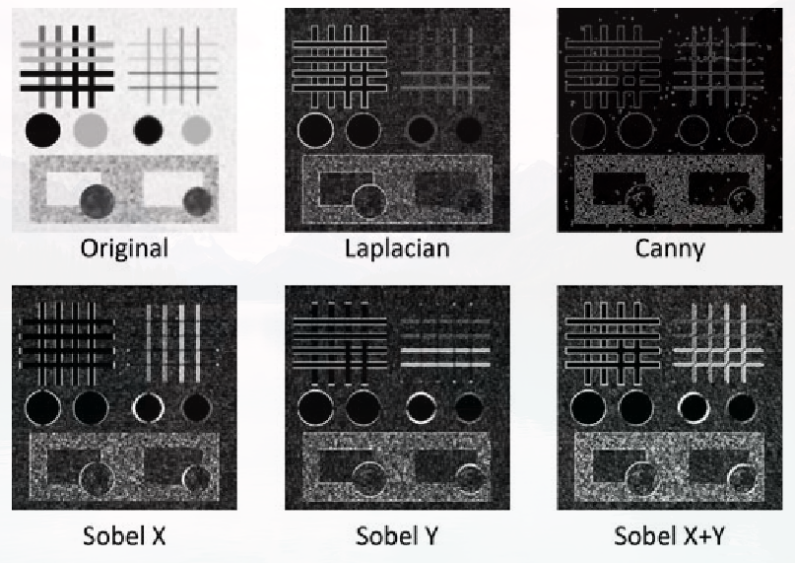


Figura 2.2: Efectos de diferentes filtros de convolución sobre la misma imagen [18].

Las redes neuronales convolucionales buscan los valores óptimos para un conjunto de filtros durante el entrenamiento, que resalten las características de la imagen, y que hagan

más sencilla la interpretación de la información de estas por las sucesivas capas de la red. En las CNN una capa de convolución se compone de un operador de convolución que aplica una colección de filtros F a una entrada I para producir la salida O, $O = conv(F, I)$.

F está compuesto por k_n filtros de dimensiones $k_h \times k_w \times c_i$, donde $k_h \times k_w$ especifican las dimensiones del filtro (2D). c_i hace referencia al número de canales de entrada. Además, la capa recibe una entrada I compuesta de t imágenes de dimensiones $h_i \times w_i \times c_i$, y produce una salida O con t salidas de tamaño $h_o \times w_o \times k_n$. Cada uno de los k_n filtros en esta capa combina una subentrada con la misma dimensión que el filtro, para producir un valor escalar en una de las k_n salidas. Al aplicar de manera repetida el filtro a toda la entrada, en forma de ventana deslizante (y con un cierto paso o *stride* vertical y horizontal, s_v y s_h), el operador de convolución produce la totalidad de los elementos de la salida [15].

En arquitecturas modernas, el rendimiento obtenido de una implementación directa de la operación de convolución se ve constreñido por el ancho de banda de la memoria. Por esta razón, mediante una implementación de este tipo, no se puede alcanzar el rendimiento óptimo del procesador. Se pueden obtener mejores resultados utilizando una aproximación basada en la Multiplicación de Matrices General (GEMM) [15]. En concreto, la transformación llamada *"IM2COL"*, aplicada a los operadores de convolución, transforma la entrada de la capa I en una matriz aumentada B, de manera que la salida de aplicar la convolución $O = CONV(F, I)$ se transforma en una multiplicación de matrices general, $C = A \cdot B = A \cdot IM2COL(I)$ donde $C \equiv O \rightarrow k_n \times (h_o \cdot w_o \cdot t)$ es la salida, vista como una matriz $m \times n$, con $m = k_n$ y $n = h_o \cdot w_o \cdot t$. La matriz $A \equiv F \rightarrow k_n \times (k_h \cdot k_w \cdot c_i) = m \times k$ contiene los filtros, y $B \rightarrow (k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t) = k \times n$ se obtiene de realizar la transformación IM2COl sobre la entrada I, de acuerdo con el tamaño de los filtros de entrada y *strides*. La Figura 2.3 muestra gráficamente esta transformación de la operación de convolución.

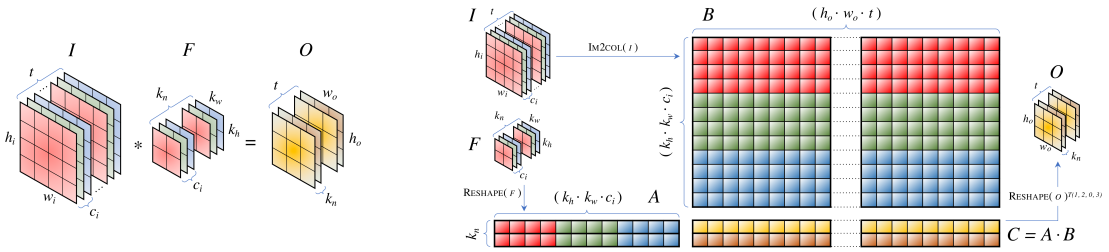


Figura 2.3: Operación de convolución con la transformación IM2COL [19].

2.4 Multiplicación de matrices

Dentro de las operaciones de las redes neuronales convolucionales, la operación que más tiempo ocupa es la de la convolución [20]. Como ya se ha visto anteriormente, esta operación se puede transformar en una multiplicación de matrices general. Por esta razón, en este trabajo, se han centrado los esfuerzos en optimizar esta última operación.

La multiplicación de matrices es un campo extensamente estudiado, y se incluye dentro de la funcionalidad de *"Basic Lineal Algebra Subprograms"* (BLAS). BLAS es una especificación que define un conjunto de subprogramas para realizar las operaciones más comunes de Álgebra Lineal. Su objetivo es definir un conjunto estándar de nombres de rutinas e interfaces [21].

```
L1   for ( jc = 0; jc < n; jc += nc )
L2     for ( pc = 0; pc < k; pc += kc ) {
         Bc ← B(pc : pc + kc − 1, jc : jc + nc − 1); // Pack
L3       for ( ic = 0; ic < m; ic += nc ) {
           Ac ← A(ic : ic + mc − 1, pc : pc + kc − 1); // Pack
L4         for ( jr = 0; jr < nc; jr += nr ) // macro-kernel
L5           for ( ir = 0; ir < mc; ir += mr )
L6             for ( pr = 0; pr < kc; pr += 1 ) // micro-kernel
                 Cc(ir : ir + mr − 1, jr : jr + nr − 1)
                   += Ac(ir : ir + mr − 1, pr)
                     · Bc(pr, jr : jr + nr − 1);
     }}
```

Figura 2.4: Pseudocódigo del algoritmo BLIS [28]

Se definen tres niveles dentro de BLAS. El primer nivel se compone de operaciones vectoriales [22]. El segundo nivel, define operaciones matriz-vector [23]. El tercer nivel está compuesto por operaciones matriz-matriz [24]. En este nivel se incluye la operación GEMM.

En la actualidad, hay una gran número de bibliotecas que ofrecen implementaciones de BLAS como Intel MKL [25], IBM ESSL [26], NVIDIA CuBLAS [27], etc. No obstante, las implementaciones de GEMM en estas bibliotecas generalmente están optimizadas para matrices cuadradas de grandes dimensiones, y al incluir toda la funcionalidad de BLAS, suelen tener un gran tamaño [28].

En contraste a estas bibliotecas, *"BLAS-like Library Instantiation Software"* (BLIS) es un framework para obtener de manera sencilla la funcionalidad de BLAS [29]. BLIS expresa las computaciones BLAS de nivel 2 y nivel 3 en forma de kernels sencillos. Este framework tiene una división entre el código de alto nivel o el macro-kernel y el código de bajo nivel o micro-kernel. El macro-kernel está pensado para que sea general y fácilmente portable a diferentes arquitecturas, mientras que el micro-kernel debe ser implementado y optimizado de manera específica para cada arquitectura.

La implementación de BLIS (Figura 2.4) es un algoritmo de multiplicación de matrices por bloques. Tiene dos bucles externos (L1 y L2), que contienen la rutina de empaquetamiento de B, y un tercer bucle (L3), que contiene la rutina de empaquetamiento de A. Estos tres bucles realizan la subdivisión del problema $C = A \times B$, donde $A \rightarrow m \times k$, $B \rightarrow k \times n$ y $C \rightarrow m \times n$, en subproblemas $C_c = A_c \times B_c$, donde $A_c \rightarrow m_c \times k_c$, $B_c \rightarrow k_c \times n_c$ y $C_c \rightarrow m_c \times n_c$. De esta manera se crean *buffers* de las matrices A y B, que si tienen los tamaños correctos, se pueden almacenar en las diferentes memorias caché.

El siguiente paso es el macro-kernel (L4 y L5), que se encarga de calcular cada subproblema $C_c = A_c \times B_c$. Se vuelve a subdividir el problema por segunda vez, dando lugar a un micro-kernel (L6) que resuelve el nuevo subproblema $C_r = A_r \times B_r$, donde $A_r \rightarrow m_r \times k_c$, $B_r \rightarrow k_c \times n_r$, y $C_r \rightarrow m_r \times n_r$.

El orden de los bucles en el algoritmo BLIS favorece que B_c se almacene en la caché de nivel L2, mientras que A_c se almacena en el nivel $L3$. El bloque (micro-tile) C_r se carga directamente de memoria a los registros del procesador. No obstante, en [28] se describen 5 variantes del algoritmo BLIS original, que reorganizan los bucles de diferentes maneras.

De manera general, este algoritmo está pensado para que dos de las matrices tengan los bloques con subdivisión c en memoria caché, mientras que la tercera matriz quedará almacenada en memoria RAM. El micro-kernel debe tener unas dimensiones que permitan al bloque de subíndice r de la tercera matriz residir en los registros del procesador.

Por otra parte, durante la ejecución del micro-kernel se han de ir cargando los "bloques r" desde la caché a los registros.

Ajustando las dimensiones de los bloques c y r se puede adaptar el algoritmo para que los diferentes bloques de las matrices permanezcan en nivel de memoria caché deseado. Además, ajustando las dimensiones apropiadas a r, es posible ajustar el algoritmo para que aproveche las instrucciones vectoriales presentes en la arquitectura. Además de esto, las rutinas para empaquetar (y desempaquetar) las matrices pueden aprovecharse para reorganizar el alineamiento de memoria de los buffers de manera eficiente, para que esté alineada con el orden requerido por el micro-kernel.

Las variaciones de BLIS se basan en reordenaciones de los bucles del algoritmo, para conseguir diferentes posiciones de las matrices en las diferentes memorias caché. Se pueden diferenciar en dos categorías principales. En la primera categoría se deja la matriz C_r residente en registros. En la segunda categoría están las variaciones que dejan la matriz A_r o B_r residente.

Estas variaciones son las que le permiten a este algoritmo ser adaptado a diferentes arquitecturas. Por ejemplo, en [19] se realiza una implementación eficiente del algoritmo BLIS sobre el procesador GreenWaves GAP8, demostrando la importancia de la realización del producto de matrices por bloques para adaptar el algoritmo a la arquitectura de caches y registros de este procesador. La versión que utilizan del algoritmo BLIS es la de A residente, ya que el micro-kernel resultante se adapta mejor a las instrucciones vectoriales del procesador.

2.5 Arduino

El microcontrolador que se utiliza para este trabajo es el Arduino Nano 33 BLE. Este chip tiene un procesador ARM Cortex-M4 con las siguientes características:

- Instrucciones de multiplicación y acumular de un solo ciclo.
- División por hardware.
- Instrucciones vectoriales *"Single Instruction Multiple Data" (SIMD)*.
- Unidad de aritmética de coma flotante de simple precisión.

Tiene un conjunto de instrucciones de 32 bits. No obstante, también trabaja con un subconjunto de instrucciones de 16 bits con las instrucciones más comunes, llamado *"Thumb"* [30].

Las instrucciones de suma y multiplicación, tanto con números enteros como con coma flotante, tardan un ciclo de reloj. Las instrucciones aritméticas más críticas son las de división y raíz cuadrada, que pueden llegar a tardar hasta 12 ciclos con aritmética entera, y hasta 14 ciclos con aritmética de coma flotante.

Las instrucciones de carga y escritura (*"Load"* y *"Store"*) tardan un número de ciclos que depende de la instrucción concreta. Generalmente un Load que cargue a un registro, con la dirección de memoria contenida por otro registro, tarda dos ciclos en completarse. Si la dirección base de memoria es el contador de programa, puede tardar un ciclo adicional. Si el contador de programa es el operando destino la carga tarda un total de 5 ciclos de reloj, ya que esta operación es bloqueante, y, además de los dos ciclos necesarios para la operación de carga, se necesitan tres ciclos extra para que el cauce de ejecución del procesador se recargue.

Las instrucciones de escritura tienen un coste de un solo ciclo de reloj. Esto se debe a la existencia de un buffer de escritura, que guarda el valor temporalmente mientras se completa la escritura. No obstante, el buffer de escritura se puede desactivar. En ese caso, la siguiente instrucción debe esperar a que se complete la escritura.

Las instrucciones de carga y escritura pueden ser agrupadas para ahorrar ciclos de reloj. El ahorro consiste en un ciclo de reloj por cada instrucción agrupada excepto la primera. Las cargas se pueden situar en cualquier posición, pero las instrucciones de escritura sólo se pueden agrupar siendo estas la última instrucción.

Además, si la carga o escritura no están alineadas, se añaden más ciclos. Esto ocurre porque, en tal caso, se debe leer o escribir de dos o más posiciones de memoria diferentes. Una lectura o escritura de un byte o un halfword (16 bits) desalineado añade un ciclo extra a la operación. Una operación con un word (32 bits) desalineado incurre en dos ciclos extra.

El procesador tiene trece registros de propósito general de 32 bits, que van desde $r0$ hasta $r12$. Se dividen en dos grupos. El primer grupo va del registro 0 al registro 7, y son accesibles por todas las instrucciones que utilizan un registro. El segundo grupo va de los registros 8 al 12 y son accesibles por todas las instrucciones de 32 bits que utilizan un registro, pero no las de 16 bits. Además, hay tres registros especiales, que son el Stack pointer, el Link register y el contador de programa.

Este procesador también dispone de instrucciones vectoriales SIMD integradas en la Unidad de Procesamiento Digital, o DSP. Estas instrucciones especiales solo pueden usarse con números enteros.

El juego completo de instrucciones SIMD del procesador se puede encontrar en la siguiente documentación [31]. Entre las instrucciones disponibles hay instrucciones aritméticas, como la instrucción de suma con signo __SADD8, que tiene como parámetros de entrada dos números enteros de 32 bits, que contienen cada uno cuatro números de 8 bits empaquetados. El resultado es un número de 32 bits que contiene 4 números de 8 bits que son el resultado de la suma con signo de cada par de números. También hay instrucciones de multiplicación como __SMLAD, e instrucciones de empaquetado como __PKHBT y __PKHTB.

La memoria se divide en SRAM (*"Static Random Access Memory"*) y Flash 2.5. Estas memorias tienen una capacidad de $256KB$ para la SRAM y $1MB$ para la Flash. La memoria Flash es de solo lectura, y solo se puede modificar mediante el cargador Micro-USB. Esta memoria tiene la función de almacenar el binario completo del programa. Durante la ejecución del programa almacena el código del programa y cualquier dato constante. Los datos guardados en esta memoria se mantienen aunque el sistema Arduino esté apagado. La memoria SRAM es de lectura y escritura. Su función es albergar todas las variables modificables del programa. Al inicio de la ejecución del programa las variables globales modificables son copiadas a esta memoria. Existe una diferencia de velocidad entre estas memorias, siendo la memoria SRAM más rápida que la Flash. Más adelante se presenta un experimento realizado para comparar la velocidad de estas dos memorias.

Un aspecto a destacar del mapa de memoria es la ausencia de memorias caché, a excepción de una caché de instrucciones, conectada a la memoria Flash. Esto significa que el procesador lee y escribe a los registros directamente desde la memoria.

Tal como se observa en la Figura 2.6, la memoria SRAM se divide entre las regiones *Static Data*, *Heap* y *Stack*. La zona de Static Data y Heap van contiguas en un extremo de la memoria, mientras que el Stack se sitúa en el otro extremo. El Heap y el Stack van creciendo el uno hacia el otro. Si se agota el espacio libre entre ellos se produce comportamiento no definido.

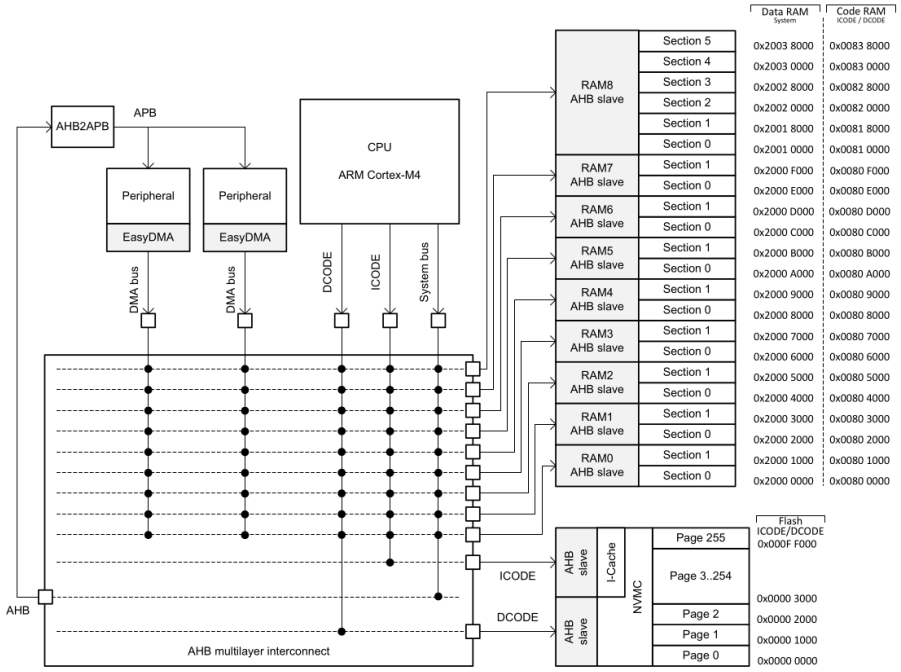


Figura 2.5: Mapa de memoria del Arduino Nano 33 BLE [32].

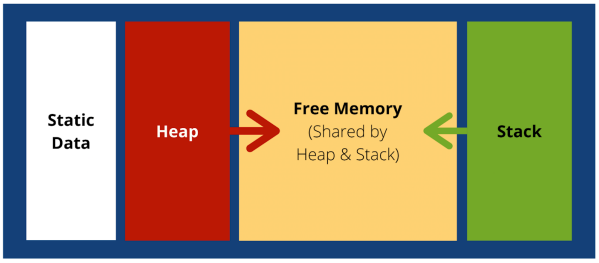


Figura 2.6: Distribución de las zonas de la memoria SRAM [33].

La zona de Static Data alberga las variables globales y las variables estáticas que no son constantes. En el Heap se localiza la memoria dinámica del programa. El Stack lo componen las variables locales y las direcciones de retorno de las funciones. El Heap puede acabar fragmentándose si se reserva y libera periódicamente memoria de diferentes tamaños. Esta fragmentación provoca se queden trozos de memoria sin aprovechar, y puede provocar el agotamiento de la memoria. Por esta razón, se desaconseja el uso de memoria dinámica cuando sea posible.

CAPÍTULO 3
Desarrollo e implementación

En este apartado se procederá a explicar el código desarrollado para la optimización del kernel de multiplicación de matrices como parte de este proyecto. Se comentan las diferentes características y razones de la implementación, así como las diferentes optimizaciones realizadas.

3.1 Multiplicación de Matrices General

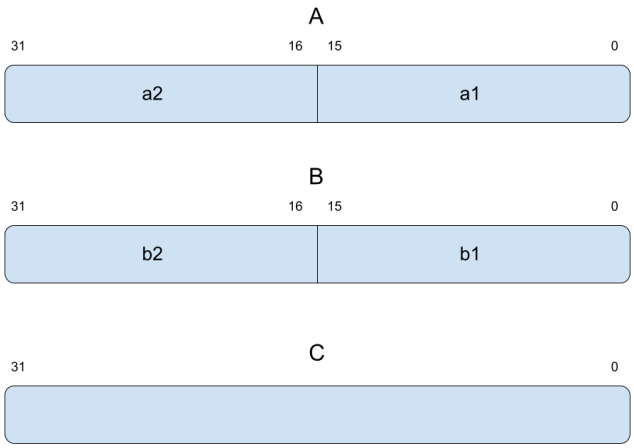
El primer paso ha sido realizar la operación de multiplicación de matrices general, para tener una base sobre la que comparar. El algoritmo clásico para la multiplicación de matrices consiste en tres bucles que van desde 0 a $n-1$, desde 0 a $m-1$, y desde 0 a $k-1$ respectivamente. Este algoritmo, por lo tanto, tiene una complejidad temporal cúbica en $O(n)$. La Figura 3.1 muestra el código de este algoritmo.

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p++) {
            C(i,j) += A(i,p) * B(p,j);
        }
    }
}
```

Figura 3.1: Código de la multiplicación de matrices básica.

Dada la limitada cantidad de memoria disponible en el Arduino, las matrices A y B están compuestas por números de 8 bits, para ahorrar espacio. La matriz C está compuesta por números de 32 bits, para reducir la posibilidad de que ocurra un desbordamiento cuando se trabaja con números grandes.

La primera optimización ha consistido en la utilización de instrucciones vectoriales SIMD del juego de instrucciones ARMv7-M. En concreto, la instrucción que se ha utilizado ha sido la instrucción __SMLAD. Esta instrucción realiza dos operaciones de multiplicación y acumula su resultado, sumándolo al valor inicial del acumulador. Los operandos son números de 16 bits agrupados en registros de 32 bits. El acumulador es un número de 32 bits, al que a su valor inicial se le suma el resultado de la suma de las dos multiplicaciones. Por lo tanto, los parámetros de entrada para esta función son dos números de 32 bits que contienen, cada uno, dos números de 16 bits empaquetados, y un número de 32 bits con el valor inicial del acumulador. La Figura 3.2 muestra el empaquetado y la operación que hace la instrucción __SMLAD.



$$C = a1 \times b1 + a2 \times b2$$

Figura 3.2: Empaquetado y funcionamiento de la instrucción __SMLAD.

La Figura 3.3 muestra el código de la primera optimización. Se empaquetan 4 números enteros de 8 bits (dos de la matriz A y dos de la matriz B) en dos registros de 32 bits.

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p+=2) {
            // Guardar los dos numeros de cada operando en un array de dos
                numeros de 16 bits
            int16_t v1[2] = {A(i,p), A(i,p+1)};
            int16_t v2[2] = {B(p,j), B(p+1,j)};
            int32_t *r1 = (int32_t*)v1;
            int32_t *r2 = (int32_t*)v2;

            // Ejecutar instruccion SIMD
            C(i,j) = __SMLAD(*r1, *r2, C(i,j));
        }
    }
}
```

Figura 3.3: Código de la primera optimización.

Con esta primera optimización, se ejecutan la mitad de ciclos del bucle interno (el que va de 0 a $k-1$), ya que se recorre de dos en dos. No obstante, no se observa una gran mejora con respecto al algoritmo base. Esto se debe a que, al utilizarse un array local auxiliar para empaquetar los valores, se están realizando cuatro escrituras y dos lecturas adicionales a las operaciones del bucle interno del algoritmo original. Las lecturas y escrituras son operaciones más costosas que las operaciones aritméticas, siendo esta la razón de que esta primera optimización no mejore el rendimiento.

Para la segunda optimización se decidió utilizar aritmética de bits para empaquetar los números. El primer número se desplaza 16 bits a la izquierda, y después se realiza una operación de bits OR con el segundo número. El objetivo es evitar las lecturas y escrituras que provocaba el uso de un array auxiliar para empaquetar los valores. La Figura 3.4 muestra el código de esta segunda optimización.

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p+=2) {
            // Juntar los dos valores en 32 bits
            int32_t r1 = (A(i,p) << 16) | A(i,p+1);
            int32_t r2 = (B(p,j) << 16) | B(p+1,j);

            // Ejecutar instruccion SIMD
            C(i,j) = __SMLAD(r1, r2, C(i,j));
        }
    }
}
```

Figura 3.4: Código de la segunda optimización.

En vista de los problemas con las lecturas y escrituras de la segunda optimización, el objetivo de la tercera optimización es reducir el número de cargas necesarias para obtener los números y empaquetarlos. Para conseguirlo, primero se obtiene un puntero a entero de 8 bits a la posición de memoria del elemento *ij* de la matriz. Después se convierte este puntero a uno de enteros de 16 bits. Por último, se desreferencia este puntero, realizando así una carga de un número de 16 bits, y se guarda el valor en una variable temporal. De esta manera, se consigue cargar dos valores con una sola instrucción de carga.

En cuanto a la aritmética de bits, es necesaria una ligera modificación, ya que se tienen que separar los dos valores de 8 bits que están contiguos. A los dos valores se les aplica una máscara de unos en su posición final con una operación AND, para eliminar los bits del otro valor, y mantener los bits del número. El valor de la izquierda solo tiene que moverse 8 bits a la izquierda, porque inicialmente ya está situado 8 bits a la izquierda en el valor original. El valor de la derecha no hay que desplazarlo porque ya está en su posición final. Este empaquetamiento se explica de manera gráfica en la Figura 3.5. La Figura 3.6 muestra el código de la tercera optimización.

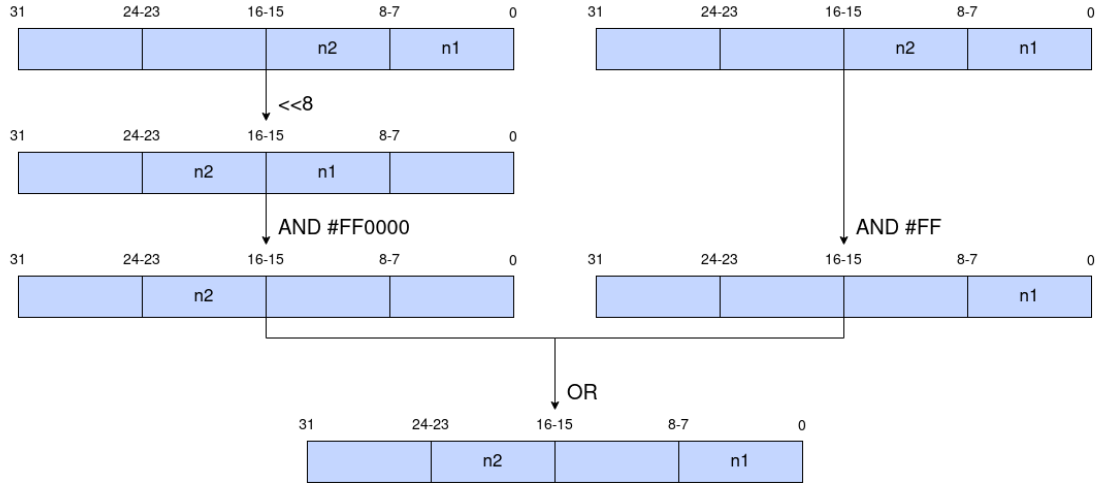


Figura 3.5: Empaquetado de los valores con aritmética de bits de la tercera optimización.

La cuarta optimización que se desarrolló consiste en realizar la multiplicación de 4 elementos por iteración del bucle interno, en lugar de 2. El contador del bucle ahora aumenta de 4 en 4. Para cargar 4 números de cada matriz, se convierte el puntero a número entero de 8 bits a un puntero a número entero de 32 bits (antes se convertía a puntero de 16 bits). De esta manera, al desreferenciar el puntero, se realiza la carga de los 4 valores al cargar un número entero de 32 bits. Con este procedimiento, se reduce todavía más el

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p+=2) {
                int16_t v1c = *(int16_t*)&A(i,p);
                int32_t r1 = (v1c << 8 & 0x00FF0000) | (v1c & 0x000000FF);

                int16_t v2c = *(int16_t*)&B(p,j);
                int32_t r2 = (v2c << 8 & 0x00FF0000) | (v2c & 0x000000FF);

            // Ejecutar instruccion SIMD
            C(i,j) = __SMLAD(r1, r2, C(i,j));
        }
    }
}
```

Figura 3.6: Código de la tercera optimización.

número de cargas necesarias para ejecutar el algoritmo. La Figura 3.7 muestra el código de la cuarta optimización.

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p+=4) {
            // Cargar 4 numeros de las matrices A y B
            int32_t va = *(int32_t*)&A(i,p);
            int32_t vb = *(int32_t*)&B(p,j);

            // Empaquetar los valores
            int32_t a12 = (va >> 8 & 0x00FF0000) | (va >> 16 & 0x000000FF);
            int32_t a34 = (va << 8 & 0x00FF0000) | (va & 0x000000FF);

            int32_t b12 = (vb >> 8 & 0x00FF0000) | (vb >> 16 & 0x000000FF);
            int32_t b34 = (vb << 8 & 0x00FF0000) | (vb & 0x000000FF);

            // Ejecutar instrucciones SIMD
            C(i,j) = __SMLAD(a12, b12, C(i,j));
            C(i,j) = __SMLAD(a34, b34, C(i,j));
        }
    }
}
```

Figura 3.7: Código de la cuarta optimización.

Con esta optimización las operaciones de bits vuelven a cambiar. Ahora hay que tratar los 4 números de A y B, que están contiguos en memoria, y empaquetarlos de dos en dos en 4 enteros de 32 bits (dos para A y dos para B). La máscara aplicada sobre los números de la izquierda y de la derecha es la misma que ya se ha explicado anteriormente. Para empaquetar el par de números situados en la parte alta (a la izquierda), el desplazamiento que hace el número de la izquierda es de 8 bits a la derecha, para situarse en el inicio del entero de 16 bits de la izquierda, del registro de 32 bits destino. El desplazamiento que hace el número de la derecha es de 16 bits a la derecha, para situarse el inicio del entero de 16 bits de la derecha, del registro destino. El par de números de la parte baja se empaquetan de la misma manera que se ha explicado en la optimización anterior. La Figura 3.8 muestra de manera gráfica el empaquetamiento del primer par de números.

La quinta optimización de este algoritmo consistió en utilizar una variable temporal con el valor de $C(i,j)$ para reducir el número de lecturas sobre este valor. La Figura 3.9

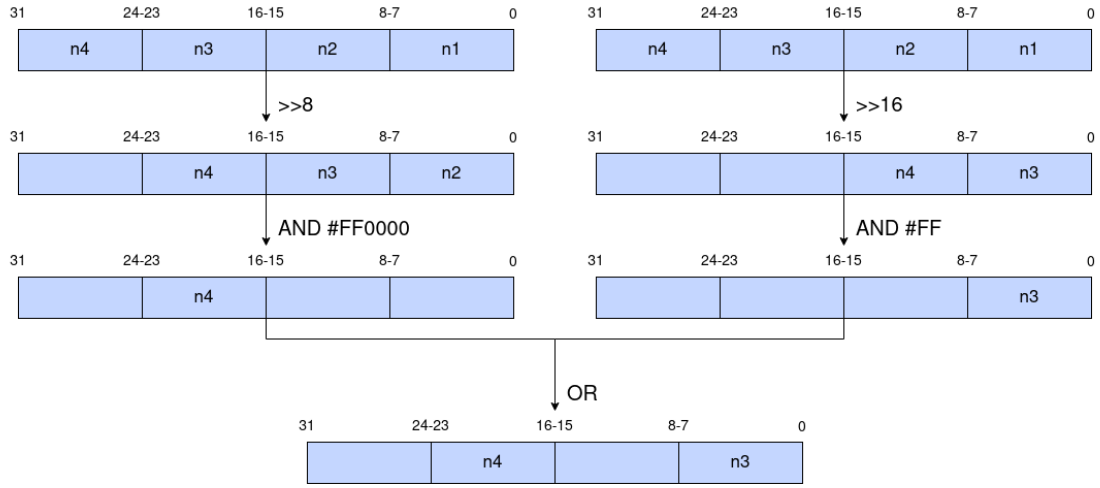


Figura 3.8: Empaquetado de los valores con aritmética de bits para el par de números de la parte alta de la cuarta optimización.

muestra el código de esta optimización. En ella se observa la variable "tmp", que es la variable temporal con el valor de $C(i, j)$.

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        int32_t tmp = C(i,j); // Guardar C(i,j) en una variable temporal
        for (int p = 0; p < k; p+=4) {
            // Cargar 4 numeros de las matrices A y B
            int32_t va = *(int32_t*)&A(i,p);
            int32_t vb = *(int32_t*)&B(p,j);

            // Empaquetar los valores
            int32_t a12 = (va >> 8 & 0x00FF0000) | (va >> 16 & 0x000000FF);
            int32_t a34 = (va << 8 & 0x00FF0000) | (va & 0x000000FF);

            int32_t b12 = (vb >> 8 & 0x00FF0000) | (vb >> 16 & 0x000000FF);
            int32_t b34 = (vb << 8 & 0x00FF0000) | (vb & 0x000000FF);

            // Ejecutar instrucciones SIMD
            tmp = __SMLAD(a12, b12, tmp);
            tmp = __SMLAD(a34, b34, tmp);
        }
        C(i,j) = tmp;
    }
}
```

Figura 3.9: Código de la quinta optimización.

Por último, la sexta optimización hace uso de otras dos instrucciones SIMD para empaquetar los números. En la Figura 3.12 se muestra el código de la optimización, donde se observan las instrucciones intrínsecas utilizadas, __PKHBT y __PKHTB. La primera instrucción recibe tres parámetros. Combina los bits [15:0] del primer parámetro con los bits [31:16] del segundo parámetro, y aplica un desplazamiento de bits hacia la izquierda al segundo registro. El número de bits de este desplazamiento está indicado por el tercer parámetro. La Figura 3.11 muestra el funcionamiento de la instrucción __PKBT. La instrucción __PKTB es igual que la primera, solo que combina los bits [31:16] del primer registro y los bits [15:0] del segundo registro. Además el desplazamiento se realiza hacia la izquierda, también sobre el segundo registro.

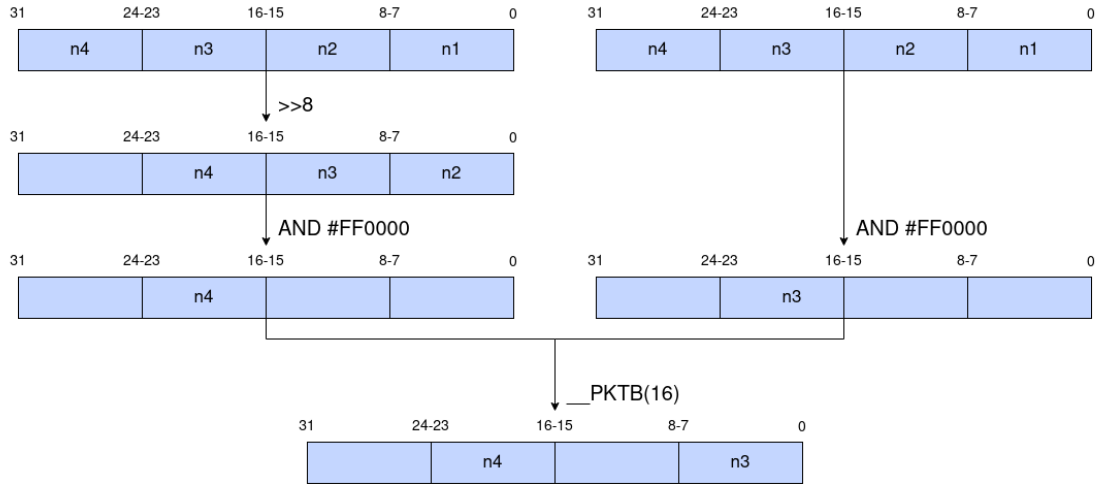


Figura 3.10: Empaquetado de valores con la instrucción __PKTB. Este empaquetamiento forma parte de la sexta optimización.

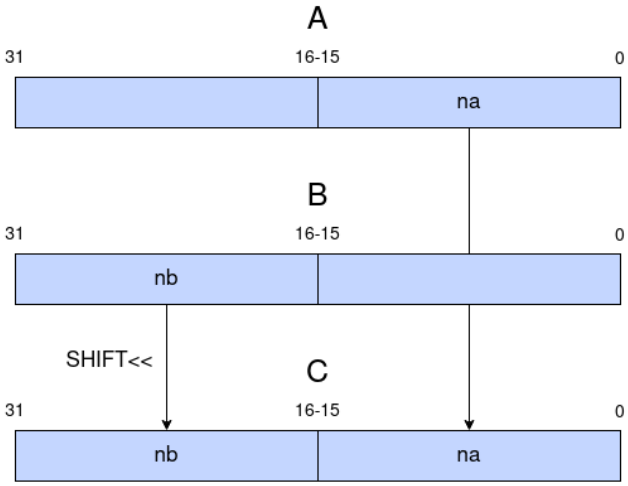


Figura 3.11: Empaquetado y funcionamiento de la instrucción __PKBT.

La manera de aprovechar estas instrucciones ha sido para realizar las operaciones de desplazamiento y AND, de las instrucciones de bits usadas para empaquetar los números, con una sola instrucción. La Figura 3.10 muestra gráficamente el empaquetado del par de números situados en la parte alta, utilizando la instrucción __PKTB. Para el par de números de la parte baja se utiliza la instrucción __PKBT.

3.2 Adaptación del algoritmo BLIS a la arquitectura del Arduino

Se decidió adaptar la multiplicación de matrices por bloques mediante el algoritmo BLIS a la arquitectura específica del Arduino Nano 33 BLE. Para ello, se utiliza una configuración específica basada en las características de esta arquitectura.

La variación del algoritmo escogida para esta arquitectura ha sido la de *A* residente. Se ha tomado esta decisión basándose en el hecho de que en el conjunto de instrucciones vectoriales SIMD no hay ninguna que haga la multiplicación de dos registros vectoriales elemento a elemento y devuelva un registro con los resultados de estas multiplicaciones. Esta instrucción sería la necesaria para la variante del algoritmo BLIS de *C* residente. En

```
void matmul(const DTYPE_AB *A, int ldA, const DTYPE_AB *B, int ldB, DTYPE_C *C,
     int ldC, int m, int n, int k) {

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            int32_t tmp = C(i,j); // Guardar C(i,j) en una variable temporal
            for (int p = 0; p < k; p+=4) {
                // Cargar 4 numeros de las matrices A y B
                int32_t va = *(int32_t*)&A(i,p);
                int32_t vb = *(int32_t*)&B(p,j);

                // Empaquetar los valores
                int32_t a12 = (va >> 8 & 0x00FF0000);
                a12 = __PKHTB(a12, (va & 0x00FF0000), 16); //Correr a la
                    derecha y empaquetar

                int32_t a34 = (va & 0x000000FF);
                a34 = __PKHBT(a34, (va & 0x0000FF00), 8); //Correr a la
                    izquierda y empaquetar

                int32_t b12 = (vb >> 8 & 0x00FF0000);
                b12 = __PKHTB(b12, (vb & 0x00FF0000), 16);

                int32_t b34 = (vb & 0x000000FF);
                b34 = __PKHBT(b34, (vb & 0x0000FF00), 8);

                // Ejecutar instrucciones SIMD
                tmp = __SMLAD(a12, b12, tmp);
                tmp = __SMLAD(a34, b34, tmp);
            }
            C(i,j) = tmp;
        }
    }
}
```

Figura 3.12: Código de la sexta optimización.

lugar de esta instrucción, hay instrucciones de multiplicar y acumular, que permiten realizar el producto escalar entre dos vectores. Estas instrucciones tienen como parámetros de entrada dos registros con dos números empaquetados en cada uno (a y b). También recibe como parámetro el valor inicial del acumulador (c). La instrucción realiza las multiplicaciones de los números de la izquierda y de la derecha, y suma los resultados y el valor inicial del acumulador ($c \mathrel{+}= a1 \times b1 + a2 \times b2$). La versión A residente se puede optimizar de manera eficiente utilizando esta instrucción.

Tal como se ha explicado antes, para adaptar el código a esta arquitectura, sólo ha habido que implementar el microkernel de BLIS [28], ya que tanto las rutinas de empaquetamiento como el macrokernel son genéricos y se pueden adaptar para las diferentes arquitecturas, buscando los parámetros adecuados para la arquitectura concreta. En el caso concreto de la arquitectura del Arduino Nano 33 BLE, conviene limitar el número de rutinas de empaquetado, ya que no hay memoria caché.

Se han probado microkernels de tipo A residente de tamaños 2×2 y 4×4. Se han hecho pruebas para comprobar que no estaba ocurriendo el fenómeno de *"register spilling"*, donde el rendimiento se reduce drásticamente debido a que los datos del algoritmo no caben en los registros del procesador, y hay que guardarlos temporalmente en memoria. Finalmente se ha decidido utilizar el microkernel 4×4, donde $A \rightarrow 4 \times 4$, $B \rightarrow 4 \times n_c$ y $C \rightarrow 4 \times n_c$, porque las últimas optimizaciones estudiadas funcionan con este tamaño.

Este microkernel, siendo de tipo A residente, guarda los 16 valores de la matriz A en registros, ahorrando el tiempo de cargar de memoria estos valores para cada una de las n_c iteraciones del bucle del microkernel. El bucle del algoritmo recorre la dimensión n_c, y en cada iteración se realiza la operación $C_j = A \times B_j + C_j, j = 1, 2, ..., n_c$.

A este microkernel también se le han aplicado las optimizaciones que se han explicado para el apartado de multiplicación de matrices general. Se han utilizado 8 números enteros de 32 bits para almacenar los 16 números de 8 bits de la matriz A. En cada iteración del bucle se almacenan en dos números de 32 bits los 4 números de la matriz B que se multiplican en esa iteración. Es importante tener en cuenta que, para que esta implementación funcione, A tiene que tener los datos almacenados por filas, y las matrices B y C tienen que estar almacenadas por columnas.

Si las matrices o los tamaños del macrokernel no son múltiplos de 4, hay llamadas al microkernel con tamaños para m o k menores de 4. La primera solución que se ha desarrollado consiste en realizar la multipicación clásica para esos casos, y utilizar el microkernel optimizado para el resto.

Una segunda solución consiste en utilizar un buffer de tamaño 4×4, y antes de ejecutar el microkernel, copiar todos los valores de A en el buffer, dejando a 0 aquellos valores que no se escriben. Entonces se ejecuta el microkernel como si fuera de 4×4, ya que los ceros de la matriz A harán que las multiplicaciones que no tienen que hacerse no influyan sobre el resultado.

No obstante, ha sido necesario realizar una modificación sobre esta solución para que sea realmente eficiente. Esta modificación ha consistido en solamente utilizar el buffer cuando el tamaño es menor que 4. Esto ahorra la operación de copia al buffer en la mayoría de llamadas al microkernel.

CAPÍTULO 4
Experimentos y resultados

Una vez se han explicado el estudio y las implementaciones realizadas para este trabajo, se va a proceder a exponer y analizar los resultados obtenidos. Los resultados de los experimentos se han expresado tanto en unidades de tiempo como en Millones de Operaciones por Segundo (MOPS).

Para calcular el resultado de un experimento en MOPS se ha tenido en cuenta el tamaño del problema a resolver, en este caso expresado por el tamaño de las matrices, controlado por los parámetros m, n y k. Para obtener el número de operaciones que realiza una determinada multiplicación de matrices se ha utilizado como base la versión básica 3.1 de esta, que tiene tres bucles anidados, y en el interno contiene una multiplicación y una suma. Además, debido a que en la mayoría de pruebas, el tiempo que tarda la prueba en terminar es demasiado pequeño como para medirlo con precisión, se ha repetido la prueba un número determinado de veces para obtener las medidas de tiempo dentro de un rango razonable. Estas repeticiones también se han tenido en cuenta para calcular los MOPS de la siguiente forma

$$MOPS = 2 \times m \times n \times k \times \text{Repeticiones}/\text{Tiempo de ejecución (en segundos)}$$

Todos los experimentos se han ejecutado sobre el dispositivo embebido Arduino Nano 33 BLE. Las gráficas utilizadas en los siguientes experimentos se han generado utilizando Octave.

4.1 Análisis del efecto en el rendimiento de BLIS con diferentes tamaños de buffer

El primer experimento tiene el objetivo de estudiar cómo afecta al rendimiento del algoritmo BLIS el tamaño de los buffers escogido. Para esta prueba, se ha elegido un tamaño arbitrario para las matrices, pero con el tamaño más grande posible. El valor elegido para m, n y k es de 150. Para el tamaño de los buffers se ha escogido variar m_c, n_c y k_c desde 1 hasta 150, en incrementos de uno. En este experimento se han almacenado las matrices A y B en memoria Flash, para tener mas capacidad de almacenamiento. Las características del almacenamiento de estas matrices en memoria SRAM o Flash se explicarán más adelante.

En este experimento, que muestra la Figura 4.1, se observa un patrón de "dientes de sierra". Cuando el tamaño de los buffers no es múltiplo de 4, se producen llamadas al micro-kernel con tamaños para m_r y k_r menores que 4, lo que provoca una bajada del rendimiento. También se observa en el experimento que cuanto mayor es el tamaño de

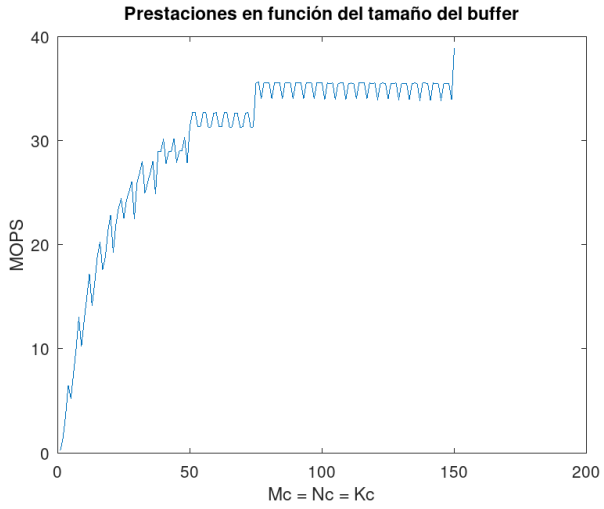


Figura 4.1: Rendimiento obtenido para $m = n = k = 180$, variando el tamaño de los buffers.

los buffers mayor es el rendimiento obtenido. Esto ocurre porque a mayor tamaño, menor es la cantidad de llamadas a las rutinas de empaquetamiento. Debido a que en el Arduino Nano 33 BLE no hay memorias caché, interesa reducir al mínimo el número de llamadas a estas rutinas.

4.2 Velocidad de las memorias del Arduino

Tal como se ha comentado anteriormente, el Arduino Nano 33 BLE dispone de dos tipos de memoria principal, memoria SRAM y Flash. Se ha procedido a realizar un experimento para comparar la velocidad de las dos memorias y cómo afecta al rendimiento del algoritmo. Las matrices que se van a multiplicar son cuadradas. El tamaño de estas matrices se ha ido variando desde un tamaño de m, n y k de 10 hasta un tamaño de 180, realizando incrementos de 10. Se ha utilizado el valor de 52 para los parámetros m_c, n_c y k_c, ya que, observando la gráfica de la Figura 4.1, ofrece buen rendimiento sin que los buffers ocupen demasiado espacio. La diferencia entre las versiones evaluadas en este experimento es que las matrices A y B están almacenadas en una u otra memoria. La matriz resultado C no se puede almacenar en la memoria Flash, ya que esta no se puede modificar.

En la prueba se ha calculado el rendimiento obtenido tanto ejecutando el algoritmo BLIS en memoria SRAM como en Flash, medido en MOPS. La gráfica de la figura 4.2 muestra el ratio de rendimiento obtenido de almacenar las matrices A y B en la memoria SRAM frente a Flash.

El pico máximo de diferencia de rendimiento se encuentra con los tamaños entre 30 y 40. Este máximo, que es de 0,64 MOPS, representa un incremento del rendimiento sobre utilizar la memoria Flash de un 0,03 %. En vista de estos resultados, compensará utilizar la memoria Flash en lugar de la SRAM, ya que de esta manera se dispone de una mayor capacidad de almacenamiento. La diferencia de rendimiento entre las dos memorias no es lo suficientemente significativa como para compensar la restricciones de capacidad resultantes de almacenar todas las matrices en la memoria SRAM. Por esta razón en el resto de experimentos se almacenan las matrices A y B en memoria Flash.

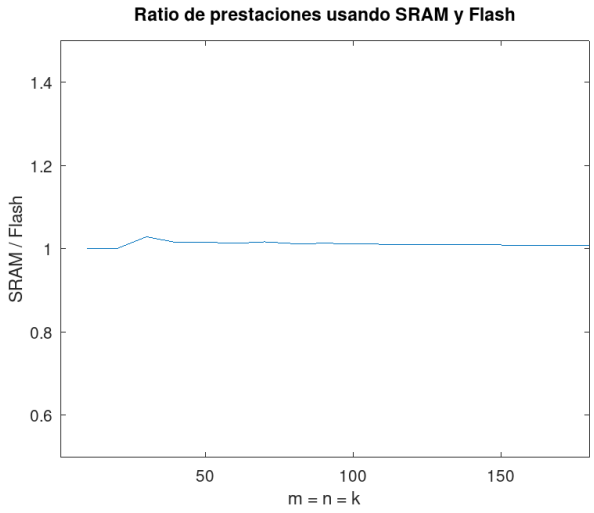


Figura 4.2: Ratio de rendimiento entre el uso de las memorias SRAM y Flash.

4.3 Tiempos de empaquetado del algoritmo BLIS

En este experimento se va a comprobar el coste de las rutinas de empaquetado del algoritmo BLIS. Para este experimento se han empleado matrices cuadradas cuyos tamaños de m, n y k varían desde 40 hasta 180, con incrementos de 10. Los parámetros m_c, n_c y k_c se han mantenido a 52. Las matrices A y B se han almacenado en memoria Flash.

La siguiente gráfica 4.3 muestra el porcentaje del tiempo total del algoritmo que representan las rutinas de empaquetado de las matrices B y C. Estas medidas se han conseguido midiendo el tiempo que tarda el algoritmo BLIS sin ejecutar las rutinas de empaquetado de matrices, para restar este tiempo al tiempo total del algoritmo, obteniendo así el tiempo que representan las rutinas de empaquetado.

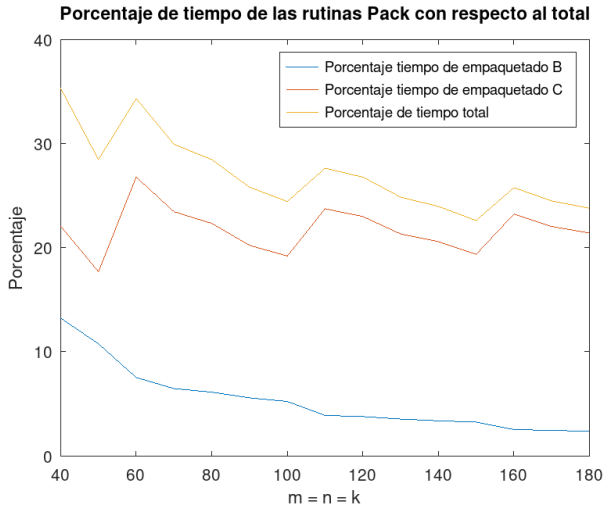


Figura 4.3: Porcentaje de tiempo que representan las rutinas de empaquetado del algoritmo BLIS.

Este experimento muestra que el tiempo de las rutinas de empaquetado del algoritmo BLIS representan entre el 23 % y el 35 % del tiempo total. Para tamaños superiores a 80 el porcentaje de tiempo que aportan las rutinas de empaquetado se estabiliza por debajo del 30 %. Se puede observar que el empaquetado de C es el que más tiempo lleva. Esto se debe principalmente a que la matriz C debe ser desempaquetada también.

4.4 Análisis de la multiplicación de diferentes tipos de matrices con el algoritmo BLIS

Tras estudiar los factores de la arquitectura del Arduino, como del diseño del algoritmo que influyen en el rendimiento, se va a proceder a estudiar el comportamiento del algoritmo BLIS al multiplicar matrices de diferentes tamaños y formas. El primer tipo de matrices que se va a estudiar son cuadradas. En este experimento se ha medido el rendimiento obtenido, tanto en MOPS como en tiempo de ejecución, de la multiplicación de matrices cuadradas que van desde tamaños de m, n y k de 10 a tamaños de 180, en incrementos de 10. m_c, n_c y k_c mantienen el valor de 52.

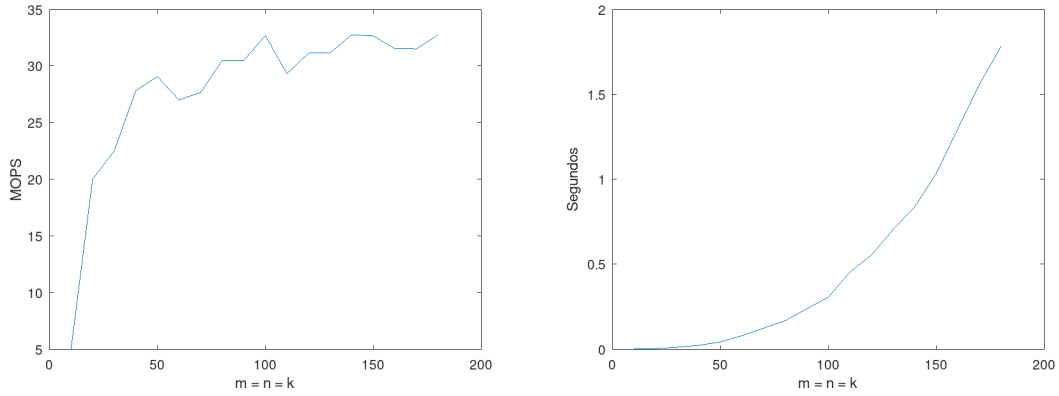


Figura 4.4: Prestaciones del algoritmo BLIS con matrices cuadradas.

La Figura 4.4 muestra los resultados de este experimento. Se puede observar que, tal como se espera de un problema con una complejidad cúbica ($m \times n \times k$), el tiempo crece de manera exponencial. No obstante, si observamos la gráfica que muestra el rendimiento en MOPS se observa que, tras un rápido aumento inicial del rendimiento, este tiende a mantenerse por encima de los 30 MOPS.

Una vez analizado el rendimiento para matrices cuadradas, se va a estudiar las matrices de tamaño $m = 100$ y $n = 100$, donde k se varía desde 50 hasta 2000 en incrementos de 50. Los valores de m_c, n_c y k_c se han mantenido a 52. Es posible alcanzar estos tamaños de matriz, porque el parámetro k solo afecta a las matrices A y B, que pueden ser almacenadas en Flash. De esta manera, se pueden probar las prestaciones con el máximo tamaño de problema posible. Como en el experimento anterior, el rendimiento obtenido se ha medido tanto en MOPS como en tiempo de ejecución. En la Figura 4.5 se pueden observar los resultados obtenidos.

En este caso se puede observar que el tiempo de ejecución aumenta linealmente. Esto es de esperar ya que en esta prueba solo se está incrementando una de las tres variables que definen el tamaño del problema. La gráfica que muestra el rendimiento en MOPS del algoritmo BLIS muestra que el rendimiento se estabiliza aproximadamente en 30 MOPS, siendo ligeramente inferior al de las matrices cuadradas.

Habiendo estudiado ya la implementación concreta del microkernel en el capítulo anterior, se podría asumir que cabría obtener el mejor rendimiento con este algoritmo para tamaños donde A fuese una matriz 4×4. La razón es que, con matrices con este tamaño se reduce el número de llamadas al microkernel. No obstante el experimento (Figura 4.6) muestra lo contrario.

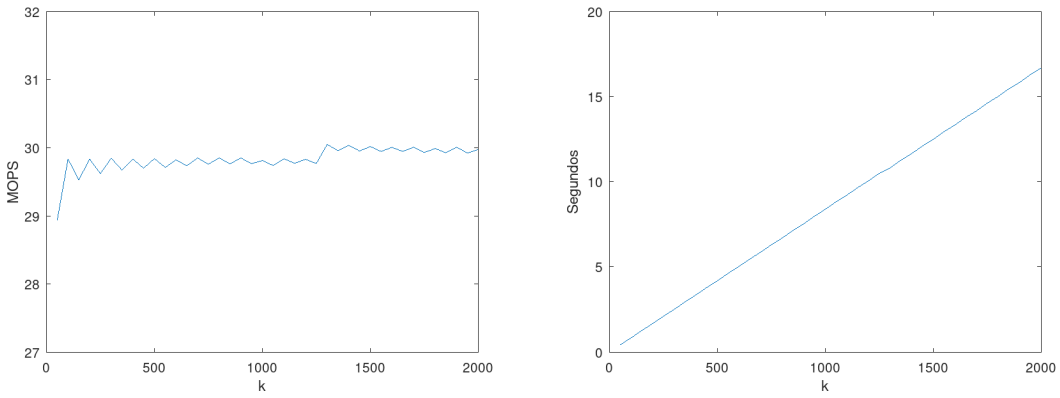


Figura 4.5: Prestaciones del algoritmo BLIS con m y n a 100, variando el parámetro k.

Para este experimento se han fijado los tamaños de m y k a 4, y se ha variado el tamaño de n desde 10 hasta 500, con incrementos de 10. También se ha fijado el valor n_c a 500, de manera que las matrices se pueden empaquetar enteras. m_c y k_c tienen el valor de 52.

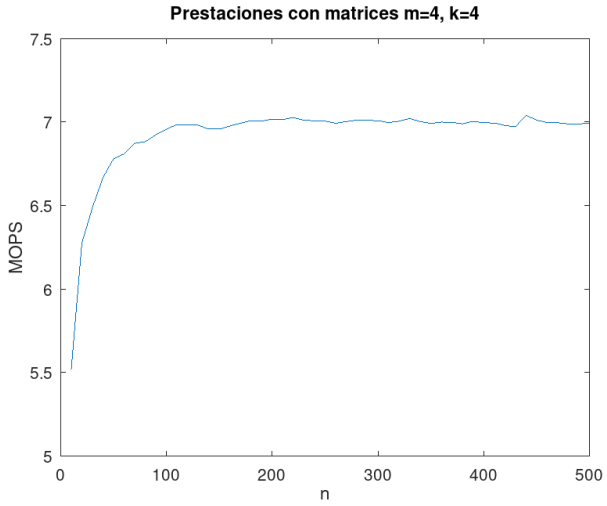


Figura 4.6: Prestaciones del algoritmo BLIS con m y k a 4, variando el parámetro n.

Los resultados de este experimento muestran que las matrices que tienen esta forma obtienen peores prestaciones que el resto. Por ejemplo, la gráfica de la Figura 4.7 muestra el rendimiento obtenido por matrices cuadradas con un número similar de elementos. En esta segunda prueba se han utilizado matrices cuadradas, donde m, n y k se van variando desde 1 hasta 40 con incrementos de 1. Los valores de m_c, n_c, y k_c se han igualado a 40 para que las matrices se puedan empaquetar enteras.

Un aspecto que destaca sobre la gráfica de la Figura 4.7 es el patrón de "dientes de sierra" que presenta. De manera similar al experimento donde se comparan diferentes tamaños para los buffers, esta pauta se debe a que el rendimiento es mejor cuando la matriz tiene un tamaño que es múltiplo de 4, debido a la implementación específica del microkernel.

Para entender por qué aparece esta diferencia de rendimiento entre estos dos tipos de matrices, se ha realizado un tercer experimento (Figura 4.8), que compara los tiempos de empaquetado del algoritmo BLIS para matrices cuadradas y matrices donde $m = 4$ y $k = 4$, teniendo en cuenta que el número de elementos total entre A y B sea el mismo.

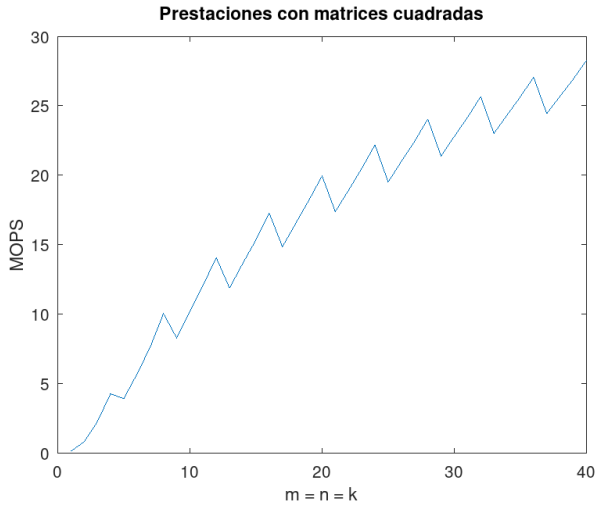


Figura 4.7: Prestaciones con matrices cuadradas de tamaños desde 1 hasta 40.

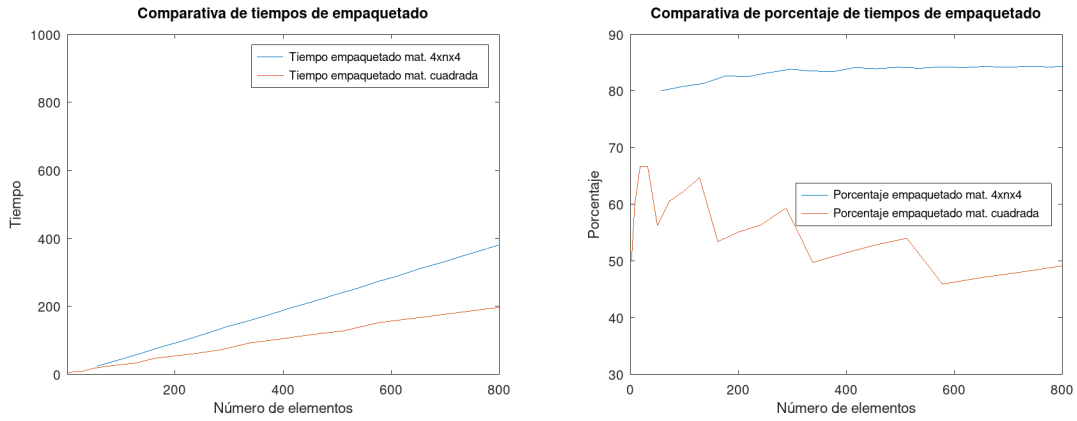


Figura 4.8: Comparativa de tiempos de empaquetado con matrices m y k igual a 4, y matrices cuadradas.

Se observa que el tiempo de empaquetado es mayor para las matrices donde $m = 4$ y $k = 4$, que para matrices cuadradas con el mismo número de elementos. Para este tipo de matrices, el tiempo de empaquetado llega a suponer más de un 80 % del tiempo total de la operación de multiplicación, comparado con el 67 % máximo obtenido con matrices cuadradas. Este tiempo adicional anula cualquier ventaja obtenida por la reducción del número de llamadas al microkernel de las matrices con m y k igual a 4.

La razón de este incremento en el tiempo de empaquetado para este tipo de matrices se debe a que la matriz A no se empaqueta, y la mayoría de los elementos están contenidos en B, que sí se empaqueta. El número de elementos total a empaquetar es mayor que con matrices cuadradas, donde el número de elementos se reparte equitativamente entre A y B, dejando menos elementos a empaquetar.

Con el propósito de comprobar estrictamente el rendimiento del microkernel, el experimento de la Figura 4.9 muestra el rendimiento de matrices con un número similar de elementos, tanto cuadradas como con $m = 4$ y $k = 4$, pero sin las rutinas de empaquetamiento. Para este experimento se han utilizado los mismos parámetros que en los casos anteriores con matrices cuadradas y con $m = 4$ y $k = 4$.

En este experimento se demuestra que, sin tener en cuenta las rutinas de empaquetamiento, las matrices con la forma m y k igual a 4 consiguen mejores prestaciones porque se adaptan mejor a la implementación concreta del microkernel que se ha hecho. Se ob-

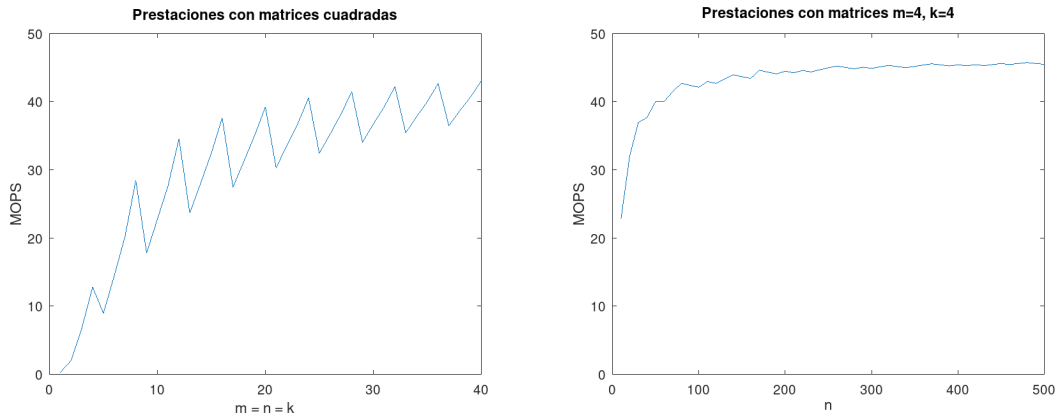


Figura 4.9: Comparativa de matrices cuadradas y m y k igual a 4 sin empaquetamiento.

serva que, para valores de n mayores que 100, el rendimiento obtenido se estabiliza sobre 45 MOPS.

De entre los tres tipos de matrices que se han estudiado, las matrices cuadradas son las que mejor rendimiento obtienen. No obstante, sin las rutinas de empaquetamiento, las matrices de tipo m y k igual a 4 obtienen más rendimiento.

4.5 Rendimiento de las diferentes optimizaciones realizadas

En el capítulo anterior se han explicado un conjunto de optimizaciones realizadas sobre el algoritmo de multiplicación de matrices. En el siguiente experimento se pone a prueba el rendimiento obtenido por cada una de estas implementaciones. Para esta prueba se ha medido el tiempo que tarda cada una de las versiones en completar una multiplicación de matrices de tamaños $m = 180$, $n = 180$ y $k = 180$. Se ha utilizado el valor de 52 para los parámetros m_c, n_c y k_c de BLIS. La Figura 4.10 muestra el resultado de este experimento en MOPS. Para todas las versiones las matrices están almacenadas en memoria SRAM, debido a la manera en la que se inicializan, excepto para BLIS, donde A y B están almacenadas en memoria Flash.

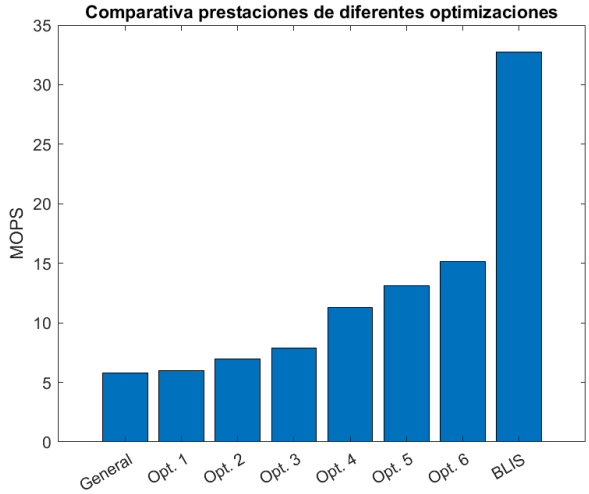


Figura 4.10: Rendimiento de las diferentes optimizaciones realizadas para una multiplicación de tamaño $m = n = k = 180$, fijando $m_c = n_c = k_c = 52$ para BLIS.

En este experimento se ha demostrado que las sucesivas optimizaciones sobre la multiplicación de matrices genérica han ido obteniendo paulatinamente mejores resultados en cuanto al rendimiento. Se puede concluir que la multiplicación mediante el algoritmo BLIS adaptado a las características de esta arquitectura es la opción óptima.

CAPÍTULO 5
Conclusiones

En este apartado se realiza una retrospectiva de los objetivos conseguidos por este trabajo, y qué conclusiones se sacan sobre ellos.

En primer lugar, se ha hecho un estudio previo de la tecnología en su estado actual. Se han analizado las características del Arduino Nano 33 BLE. También se ha reconocido que la optimización del algoritmo de multiplicación de matrices general es un algoritmo a priorizar si se quiere mejorar el rendimiento de las redes neuronales, sobretodo de las redes neuronales convolucionales. Además, se ha revisado el estado actual de GMM, y, de entre las librerías disponibles que proporcionan esta funcionalidad, se ha elegido el framework BLIS por su adaptabilidad a diferentes arquitecturas, y por ser ligero y apropiado para el tipo de dispositivo sobre el que se ha trabajado.

Se ha conseguido implementar rutinas que aprovechan las características de la arquitectura concreta del Arduino Nano 33 BLE, usando las instrucciones SIMD disponibles, y adaptándose a la estructura y tamaño de las memorias para conseguir un mejor rendimiento en el kernel de la multiplicación de matrices general. Como parte de esta implementación se ha adaptado el algoritmo GEMM BLIS a esta arquitectura, desarrollando un micro-kernel adaptado al Arduino que aprovecha las optimizaciones realizadas.

Los experimentos realizados muestran las características del algoritmo desarrollado, y también demuestran que se ha conseguido obtener un mejor rendimiento con las optimizaciones que se han realizado sobre el kernel de la multiplicación de matrices general. Estas pruebas han servido para estudiar los diferentes parámetros de BLIS para saber que configuración se adapta mejor a la arquitectura del Arduino. Las pruebas hechas sobre diferentes tipos de matrices han demostrado que las matrices cuadradas son las que más rendimiento obtienen con este algoritmo.

En general, vista la importancia que tiene este kernel para la ejecución de algoritmos de inteligencia artificial de redes neuronales, se puede afirmar que se ha conseguido avanzar en el campo del Aprendizaje Automático aplicado a dispositivos embebidos.

Bibliografía

[1] Oracle. (2022) ¿qué es la inteligencia artificial—ia? [Online]. Available: https://www.oracle.com/es/artificial-intelligence/what-is-ai

[2] A. M. Turing and J. Haugeland, "Computing machinery and intelligence," *The Turing Test: Verbal Behavior as the Hallmark of Intelligence*, pp. 29–56, 1950.

[3] J. Moor, "The dartmouth college artificial intelligence conference: The next fifty years," *Ai Magazine*, vol. 27, no. 4, pp. 87–87, 2006.

[4] S. Dick, "Artificial intelligence," 2019.

[5] I. El Naqa and M. J. Murphy, "What is machine learning?" in *machine learning in radiation oncology*. Springer, 2015, pp. 3–11.

[6] A. L. Fradkov, "Early history of machine learning," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 1385–1390, 2020, 21st IFAC World Congress. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896320325027

[7] C. Strauch, U.-L. S. Sites, and W. Kriha, "Nosql databases," *Lecture Notes, Stuttgart Media University*, vol. 20, no. 24, p. 79, 2011.

[8] Apache. (2022) Unified engine for large-scale data analytics. [Online]. Available: https://spark.apache.org

[9] N. Rusk, "Deep learning," *Nature Methods*, vol. 13, no. 1, pp. 35–35, 2016.

[10] D. Maulud and A. M. Abdulazeez, "A review on linear regression comprehensive in machine learning," *Journal of Applied Science and Technology Trends*, vol. 1, no. 4, pp. 140–147, 2020.

[11] Y. Liu, Y. Wang, and J. Zhang, "New machine learning algorithm: Random forest," in *International Conference on Information Computing and Applications*. Springer, 2012, pp. 246–252.

[12] K. P. Sinaga and M.-S. Yang, "Unsupervised k-means clustering algorithm," *IEEE access*, vol. 8, pp. 80 716–80 727, 2020.

[13] K. Rose, S. Eldridge, and L. Chapin, "The internet of things: An overview," *The internet society (ISOC)*, vol. 80, pp. 1–50, 2015.

[14] IBM. (2020) Neural networks. [Online]. Available: https://www.ibm.com/cloud/learn/neural-networks

[15] S. Barrachina, M. F. Dolz, P. San Juan, and E. S. Quintana-Ortí, "Efficient and portable gemm-based convolution operators for deep neural network training on multicore processors," *Journal of Parallel and Distributed Computing*, 2022.

[16] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly, 2020. [Online]. Available: https://books.google.es/books?id=sB3mxQEACAAJ

[17] D. Pérez, "Sistemas embebidos y sistemas operativos embedidos," *Lecturas en ciencias de la computación. Universidad Central de Venezuela, Vols. % i de % 2lSSN*, pp. 1316–6239, 2009.

[18] S. Uchida, "Image processing and recognition for biological images," *Development, growth & differentiation*, vol. 55, no. 4, pp. 523–549, 2013.

[19] C. Ramírez, A. Castelló, and E. S. Quintana-Ortí, "A blis-like matrix multiplication for machine learning in the risc-v isa-based gap8 processor," *The Journal of Supercomputing*, pp. 1–10, 2022.

[20] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[21] R. J. Hanson, F. T. Krogh, and C. Lawson, "A proposal for standard linear algebra subprograms," Tech. Rep., 1973.

[22] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[23] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms: Model implementation and test programs," Argonne National Lab., IL (USA), Tech. Rep., 1987.

[24] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[25] Intel. (2021) Get started guide. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html

[26] IBM. (2018) Essl guide and reference. [Online]. Available: https://www.ibm.com/docs/en/SSFHY8_6.1/reference/essl_reference_pdf.pdf

[27] Nvidia. (2022) cublas. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[28] A. Castelló, E. S. Quintana-Ortí, and F. D. Igual, "Anatomy of the blis family of algorithms for matrix multiplication," in *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2022, pp. 92–99.

[29] F. G. Van Zee and R. A. Van De Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, pp. 1–33, 2015.

[30] A. Developer. (2022) The thumb instruction set. [Online]. Available: https://developer.arm.com/documentation/ddi0333/h/introduction/arm1176jz-s-architecture-with-jazelle-technology/the-thumb-instruction-set

[31] CMSIS. (2022) Intrinsic functions for simd instructions. [Online]. Available: https://www.keil.com/pack/doc/CMSIS/Core/html/group__intrinsic__SIMD__gr.html

[32] N. Semiconductor. (2021) nrf52840, memory. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fps_nrf52840%2Fmemory.html

[33] SEEED. (2021) Managing arduino memory: Flash, sram, eeprom! [Online]. Available: https://www.seeedstudio.com/blog/2021/04/26/managing-arduino-memory-flash-sram-eeprom