ACELERACIÓN DE AI EN DISPOSITIVOS DE BAJO CONSUMO

AI ACCELERATION ON LOW-POWER DEVICES

SALVADOR ALBARRÁN TIRADAS

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería de Computadores

Junio 2020

Director/es y/o colaborador:

Luis Piñuel Moreno
Francisco Igual Peña

Resumen

En la actualidad podemos observar un claro auge de la introducción de técnicas de aprendizaje automático aplicadas a múltiples campos como por ejemplo la robótica, biometría mediante verificación facial, reconocimiento de voz, clasificación de objetos, etc.

En respuesta a este auge, resulta cada vez más común la fabricación y distribución de *hardware* de altas prestaciones, pero con un consumo demasiado elevado y con un propósito generalista, no enfocado en ningún caso a procesos de aceleración de algoritmos clásicos de Inteligencia Artificial en general, o Aprendizaje Automático en particular.

Por ello, cada vez más, se encuentran en el mercado nuevos dispositivos especializados en IA (Inteligencia Artificial) que conjugan bajo consumo y alto rendimiento. Este tipo de dispositivos se centran en la aceleración de procesos de entrenamiento o inferencia sobre redes neuronales. En el presente trabajo realizaremos un estudio centrándonos en el proceso de inferencia, cuyos principales retos desde el punto de vista computacional son los estrictos requisitos a nivel de consumo energético y tiempo de respuesta.

Concretamente, en este trabajo nos centraremos en realizar el análisis de un nuevo *hardware* de propósito específico (concretamente el procesador Kendryte K210), incidiendo sobre su rendimiento y consumo energético sobre una aplicación de visión artificial empotrada para la clasificación automática de objetos. Para alcanzar este objetivo se han utilizado dos placas, Maix Go y Maix Bit del fabricante Seeed Studio Sipeed, un conjunto de datos del reto ImageNet ILSVRC 2012 y modelos MobileNet v1. Se han realizado los mismos experimentos en ambas placas para determinar la diferencia de consumo energético y rendimiento usando dos entornos de desarrollo, MicroPython y Standalone SDK C.

Palabras clave

- Inteligencia artificial (IA)
- Arquitecturas de propósito específico para IA
- *Hardware* de bajo consumo
- Inferencia sobre redes neuronales
- Clasificación de objetos
- Redes neuronales
- RISC-V
- Kendrite K210

Abstract

Nowdays we can observe a clear boom in the introduction of automatic learning techniques applied to multiple fields such as robotics, biometry through facial verification, voice recognition, object classification, etc.

In response to this trend, it is increasingly common to manufacture and distribute high performance hardware, but at the exchange of an unacceptable power consumption and with a generalist purpose, not focused in any case on the acceleration of algorithms in Artificial Intelligence (AI) in general, or Machine Learning (ML) in particular.

For this reason, more and more, new devices specialized in AI are available in the market, combining low power consumption and high performance. These types of devices are focused on accelerating training processes or inference on neuronal networks. In this work, we will carry out a study focusing on the inference process, whose main challenges from the computational point of view are the strict requirements in terms of energy consumption and response time.

Specifically, in this work we will focus on the analysis of a new domain-specific accelerator –DSA– (specifically the Kendryte K210 processor) with emphasis on its performance and power consumption on an embedded machine-vision application for automatic object classification. To achieve this goal, we have used two boards, Maix Go and Maix Bit manufactured by Seeed Studio Sipeed, a data set from the ImageNet ILSVRC 2012 challenge and MobileNet v1 models. The same experiments have been performed on both boards to determine the difference in consumption and performance using two development environments, MicroPython and Standalone SDK C.

Keywords

- Artificial Intelligence (AI)
- Domain Specific Architectures (DSAs) for AI
- Low-power hardware
- Neural network inference
- Object classification
- Neural networks
- RISC-V
- Kendryte K210

Índice general

Agradecimientos

En primer lugar, me gustaría referirme a mis directores Luis Piñuel Moreno y Francisco Igual Peña por hacer posible la realización de este TFG, por apoyarme y guiarme durante todo el trayecto.

Por supuesto dar las gracias a mis amigos y familiares que me han estado apoyando cualquier ayude que necesitase desde el principio del grado hasta el final siendo un pilar necesario para poder llegar a este punto.

Capítulo 1

Introducción

1.1. Estado del arte

La Inteligencia Artificial (IA) es el área científica que se encarga del diseño y creación de sistemas que puedan imitar las capacidades cognitivas del ser humano, percibiendo el entorno y realizando acciones. Actualmente se ha convertido en un campo de estudio necesario para facilitar y automatizar un elevado número de tareas de forma rápida, eficiente y autónoma. Una rama específica de la Inteligencia Artificial es el Aprendizaje Automático, cuyo objetivo es hacer que una máquina aprenda por si misma a partir de un conjunto de datos inicial, desempeñándose cada vez mejor según la experiencia obtenida progresivamente.

Una de las herramientas más extendidas en los últimos tiempos en el ámbito de la IA en general, y del Aprendizaje Automático en particular, son las *redes neuronales*. Las redes

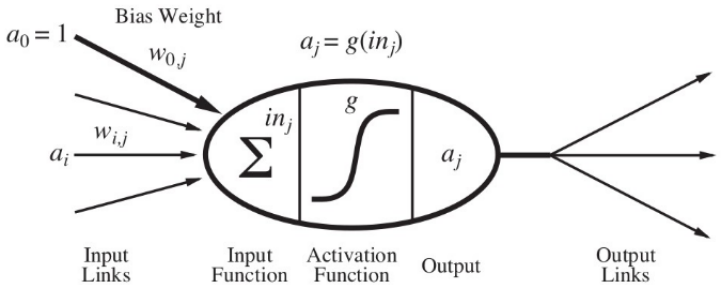

Figura 1.1: *Esquema general de una neurona. Imagen cortesía de [12].*

neuronales (ver figura 1.1) comprenden un conjunto de unidades conectadas entre sí llamadas

neuronas que tienen como función procesar datos de entrada y generar salidas. Las redes neuronales presentan tres tipos de capas interconectadas entre sí: *(i) capa de entrada*, que recibe los primeros datos de entrada sin modificar; *(ii) capa de salida*, que genera la salida o salidas de los datos procesados a lo largo de las anteriores capas; y *(iii) capas ocultas*, de número variable, son aquellas capas que no están conectadas directamente con las entradas o salidas.

En cada capa pueden existir una o más neuronas conectadas totalmente o parcialmente con las neuronas de la siguiente capa, de forma que cuando la salida de esta pase a la siguiente neurona la salida generada será multiplicada por el valor del enlace entre ambas denominado *peso*, y acumulado a cada una del resto de las entradas. A la salida de cada neurona puede existir una función de activación que se aplicará a los resultados obtenidos de las neuronas conectadas a ellas y lo modificarán para que no sobrepase cierto valor antes de enviárselo a las siguientes neuronas. Existen múltiples funciones de activación, como por ejemplo:

- La función *sigmoide*, que transforma los valores bajos que tienden a 0 y los altos a 1 de manera asintótica.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- La función $ReLU$, que solo modifica los valores introducidos que sean negativos a 0.

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases} \quad f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$$

- La función *softmax*, que cambia las salidas a una representación en forma de probabilidades, siendo así que el sumatorio de todas ellas sea 1. Suele utilizarse en las últimas capas, las de salida, en problemas de clasificación, por ejemplo.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_j}}$$

El cómputo necesario para obtener los resultados a través de distintas operaciones en la red neuronal puede llegar a ser ineficientes si ésta es demasiado grande, por lo que para obtener

un mejor rendimiento y optimizar el cálculo podemos realizar una representación matricial, representando a modo de matriz los pesos de los enlaces y un vector o matriz los datos o estímulos de entrada, haciendo que las operaciones realizadas como la multiplicación se hagan entre matrices y vectores o solo entre matrices, operaciones típicamente optimizadas en bibliotecas matemáticas.

Al final, para que una red neuronal esté entrenada y aprenda sola se necesitará reducir una función de pérdida mediante un proceso conocido como *back-propagation*, donde se actualizan los valores de los pesos de las neuronas para reducir el valor de dicha función de pérdida. La función de pérdida se utiliza para optimizar los valores de los parámetros de una red neuronal, asignando un conjunto de valores de parámetros para la red neuronal a un valor escalar indicando como de bien logran los parámetros realizar la tarea que la misma debe realizar. El proceso mediante el cual una función de pérdida se reduce se denomina propagación hacia atrás, algoritmo que funciona calculando el gradiente de la función de pérdida con respecto a cada peso por la regla de la cadena, calculando el gradiente de una capa a la vez, iterando hacia atrás desde la última capa para evitar cálculos redundantes propagándolo a las capas ocultas anteriores. Este proceso de entrenamiento es computacionalmente costoso, y por tanto voraz a nivel de capacidades de cómputo. Es por ello que suelen dedicarse gran cantidad de recursos computacionales para llevarse a cabo, normalmente en servidores dedicados dentro de grandes centros de proceso de datos.

Entre los múltiples tipos de redes neuronales encontramos las redes neuronales convolucionales, o CNNs. Las CNNs son aquellas que tienen por lo menos una capa convolucional. Una red de este tipo por lo general tienen combinaciones de capas convolucionales, capas de reducción y al final una capa que conecta con todas las neuronas de la siguiente capa. En estas capas se suelen implementar filtros convolucionales, que son matrices con unos valores añadidos, los cuales se aplican a una matriz de entrada a modo de filtro; cada operación convolucional trabaja sobre un fragmento de la matriz de entrada, por ejemplo en una matriz de 5x5 si usamos un filtro de 3x3 obtendríamos una matriz 3x3 en 9 operaciones

convolucionales resultando una nueva matriz de 3x3 para después sumar los valores de esta matriz y así obtener el resultado final de la operación. Después de esta capa se le aplica como hemos visto anteriormente una función de activación antes de enviar el dato a la capa de reducción, que consiste en reducir la matriz creada previamente por una capa convolucional. Al igual que una operación convolucional, esta capa divide esa matriz en rodajas y luego desliza esa operación convolucional progresivamente, obteniendo una matriz reducida; normalmente de la matriz reducida se obtiene el valor medio o máximo. La última capa será una capa totalmente conectada a la siguiente (implementada normalmente como un producto de matrices) enviando los datos obtenidos a través de las anteriores capas.

Existen dos procesos importantes a destacar en toda aplicación de redes neuronales: el *entrenamiento* y la *inferencia*. El entrenamiento es un proceso que consiste en la modificación de los pesos de la red para que consiga extraer los resultados deseados a partir de datos de entrada diferentes a los de entrenamiento. Uno de los hiperparámetros importantes es el tamaño de lote (*batch size*), que controla el número de muestras de entrenamiento para trabajar antes de que se actualicen los parámetros internos del modelo. También es relevante el número de épocas (*epoch*) hiperparámetro que controla el número de pases completos a través del conjunto de datos de entrenamiento. Este proceso es increíblemente costoso, ya que requiere para una buena precisión en el resultado una cantidad muy grande de datos, y por supuesto de cómputo a lo largo del ciclo de entrenamiento por lo que para este proceso es normal que se usen GPUs, ya que son capaces de realizar muchas más operaciones que una CPU. Las restricciones de precisión en el cómputo, además, son estrictas, para mejorar tanto el tiempo de entrenamiento como la calidad de la red (o modelo) finalmente entrenada.

La inferencia es el proceso el cual dado un modelo ya entrenado, devuelve el resultado obtenido a partir de nuevos datos de entrada distintos a los del entrenamiento y dando el resultado esperado. El tamaño de lote (*batch size*) es un hiperparámetro también usado en inferencia, que se refiere al número de muestras que se desean procesar en el modelo de forma simultánea. El ajuste de este parámetro tiene como objetivo conseguir un balance

óptimo entre la latencia y la cantidad de muestras procesadas en el momento. Este proceso no tiene tanto coste como el entrenamiento por lo que es necesario que sea rápido y barato, y posibilita su ejecución cerca de la sensorización, por ejemplo en un dispositivo móvil o en un sistema empotrado.

Las aplicaciones con redes neuronales son diversas, desde el reconocimiento facial hasta la clasificación de objetos; normalmente, a nivel de inferencia, suelen realizarse en dispositivos móviles, ya que, como se ha indicado, la inferencia es computacionalmente poco exigente, y por tanto perfecta para esta clase de dispositivos. Con el objetivo de obtener un alto rendimiento en los procesos de inferencia, y conseguir resultados de forma rápida y con bajo coste, surgen las arquitecturas de propósito específico centradas en la aceleración de algoritmos de aprendizaje automático y obteniendo un mayor rendimiento con un bajo consumo.

Un claro ejemplo es el acelerador Kendryte K210 sobre el que se trabaja en el presente proyecto; utilizando esta plataforma, se llevará a cabo una evaluación del proceso de inferencia sobre dos placas que integran este sistema y realizarán comparaciones con otras arquitecturas alternativas como los aceleradores en forma de USB, Google Coral y el Intel Neural Compute Stick 2, obteniendo resultados experimentales sobre el rendimiento (en términos de *fps* (fotogramas por segundo) y consumo energético (en vatios (W)). Este tipo de arquitecturas se denominan DSAs (*Domain Specific Accelerators*), y se consideran en gran auge en la actualidad en muchos ámbitos, incluida la Inteligencia Artificial.

1.2. Objetivos del trabajo

El objetivo de este TFG es realizar un *análisis de rendimiento y consumo energético de procesos de inferencia sobre redes neuronales sobre una arquitectura de propósito específico y compararla con otras arquitecturas alternativas*. Para ello primero entrenaremos varios modelos, los convertiremos al formato KMODEL soportado por el Kendryte K210, y realizaremos inferencia en un marco experimental reproducible para obtener resultados que

también lo sean, y que por tanto permitan un análisis aislado y comparativo fiable con otras DSAs.

1.3. Estructura de la memoria

La memoria se estructura en 6 capítulos:

- Capítulo 1: se realiza una breve introducción al aprendizaje automático, las redes neuronales y a las arquitecturas de propósito general y específico para redes neuronales además de hablar sobre los objetivos del trabajo y la estructura de la memoria.
- Capítulo 2: se describe la arquitectura Kendryte K210, sus características, las placas con las que hemos realizado los experimentos y otras placas alternativas que usaremos para la comparación.
- Capítulo 3: en este capítulo se describe la infraestructura software que hemos utilizado en este proyecto, así como las herramientas utilizadas para el entrenamiento, compilación y ejecución del mismo.
- Capítulo 4: lista los pasos a desarrollar desde que elegimos el conjunto de datos y los requisitos previos antes de empezar el entrenamiento, hasta la ejecución del modelo pasando previamente por la compilación del modelo.
- Capítulo 5: se centra en los experimentos realizados en ambas placas, los resultados obtenidos y una comparación con otras arquitecturas alternativas.
- Capítulo 6: se extraen una serie de conclusiones finales tras la realización del trabajo.

1.4. State of the art

Artificial intelligence is the area that designs and creates systems that can imitate human abilities by perceiving the environment and performing actions. Currently it has become a necessary study to facilitate many tasks in a quick and efficient fashion. One of its main branches is Machine Learning (ML) that aims at making a machine learn by itself from an initial data set performing increasingly better based on the experience it obtains.

One of the most widespread tools in recent times in the field of AI in general, and machine learning in particular, are *neural networks*.

Neural networks (see figure 1.1) comprise a set of units connected to each other called neurons whose function is to process input data and generate outputs. Neural networks present three types of interconnected layers: *(i) input layer*, there is one that receives the first unmodified input data, *(ii) output layer* that generates the output or outputs of the data processed along the previous layers and *(iii) hidden layers*, varying number, are those layers that are not directly connected to the inputs or outputs.

In each layer there can be one or more neurons connected totally or partially with the neurons of the next layer, so that when the output of the latter passes to the next neuron the generated output will be multiplied by the value of the link between the two called *weight*, and accumulated to each of the other inputs. At the output of each neuron, there may be an activation function that will be applied to the results obtained from the neurons connected to them and will modify it so that it does not exceed a certain value before sending it to the following neurons. There are multiple activation functions, such as:

- The *sigmoid* function, that transforms low values with 0 and high values with 1 asymptotically.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- The $ReLU$ function, that only modifies the entered values that are negative by cance-

ling them.

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases} \quad f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$$

- The *softmax* function, that changes the outputs to a representation in the form of probabilities, so that the summation of all of them is 1. It is usually used in the last layers, the output layers, in classification problems, for example.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_j}}$$

The computation necessary to obtain the results through different operations in the neural network it can become inefficient if it is too large, so to obtain a better performance and modifications to the calculation we can make a representation matrix representing as a matrix the weights of the links and a vector or matrix the input data or stimulus, doing the operations performed as the multiplication are done between matrices and vectors or only between matrices, operations typically optimized in mathematical libraries.

At the end, for a neural network to be trained and learn on its own, it will be necessary to reduce a loss function through a process known as *back-propagation*, where the values of the weights of the neurons are updated to reduce the value of said loss function. The loss function is used to optimize the values of the parameters of a neural network, assigning a set of parameter values for the neural network to a scalar value indicating how well the parameters achieve the task that it must perform. The process by which a loss function is reduced is called backward propagation, an algorithm that works by calculating the gradient of the loss function with respect to each weight by the chain rule, calculating the gradient of one layer at a time, iterating backward from the last layer to avoid redundant calculations by propagating it to the previous hidden layers.This training process is computationally costly, and therefore voracious in terms of computing capabilities. This is why a large amount of computing resources are usually dedicated to this process, usually on dedicated servers within large data processing centers.

Among the multiple types of neural networks we find the convolutional neural networks, or CNNs. CNNs are those that have at least one convolutional layer. A network of this type usually have combinations of convolutional layers, reduction layers, and finally one layer that connects to all the neurons in the next layer. In these layers, convolutional filters are usually implemented, which are matrices with added values, which are applied to an input matrix as a filter; each convocational operation works on a fragment of the input matrix, for example in a 5x5 matrix if we use a 3x3 filter we would obtain a 3x3 matrix in 9 convolutional operations resulting in a new 3x3 matrix and then add the values of this matrix and thus obtain the final result of the operation. After this layer, an activation function is applied as we have previously seen before sending the data to the reduction layer, which consists of reducing the matrix previously created by a convolutional layer. As a convolutional operation, this layer divides that matrix into slices and then slides that convolutional operation by strides, obtaining a reduced matrix; normally from the reduced matrix the average or maximum value is obtained. The last layer will be a layer fully connected to the next (normally implemented as a matrix product) sending the data obtained through the previous layers.

There are two important processes to highlight in any neural network application: *training* and *inference*. Training is a process that consists of modifying the weights of the network so that it can extract the desired results from input data other than the training data. One of the important hyper parameters is the *batch size*, which controls the number of training samples to work with before the internal model parameters are updated. It is also relevant the number of *epochs* which is a hyper parameter that controls the number of complete passes through the training data set. This process is incredibly expensive, since it requires for a good precision in the result a very large amount of data, and of course computation throughout the training cycle, so for this process it is normal that GPUs are used, since they are capable of performing many more operations than a CPU.The precision restrictions in the computation, in addition, are strict, to improve both the training time

and the quality of the network (or model) finally trained

Inference is the process which, given a model already trained, returns the result obtained from new input data other than the training data and giving the expected result. *Batch size* is a hyper parameter also used in inference that refers to the number of samples you wish to process in the model simultaneously.The adjustment of this parameter aims at an optimal balance between latency and the number of samples processed at the moment. This process does not have as much cost as training so it needs to be fast and cheap, and enables its execution close to the sensor, e.g. in a mobile device or in an embedded system.

The applications are diverse, from facial recognition to classification of objects; usually, at the level of inference, they are carried out on mobile devices since, as stated, the inference is computationally undemanding, and therefore perfect for this class of devices. With the objective of obtaining a high performance in the inference processes, and achieving results, and to obtain results, quickly and at low cost, specific-purpose architectures are emerging that are focused on accelerating machine learning algorithms and obtaining greater performance with low consumption.

A clear example is the Kendryte K210 accelerator that is being worked on in the present project; using this platform, an evaluation of the inference process will be carried out on two boards that make up this system and perform comparisons with other architectures such as USB-shaped accelerators, Google Coral and the Intel Neural Compute Stick 2, obtaining experimental results on performance ((frames per second)) and energy consumption (Watt (W)). These types of architectures are called DSAs (Domain Specific Accelerators), and are considered to be booming today in many areas, including Artificial Intelligence

1.5. Work objectives

The objective of this TFG is to do a performance and consumption analysis on a domain specific architecture and compare with other alternative architectures. To do this, we will first train several models, convert to the KMODEL format supported by the Kendryte K210,

and we will make inference in a reproducible experimental framework to obtain a sure result, analyze them and carry out a reliable comparison with the other DSAs.

1.6. Memory structure

- Chapter 1: A brief introduction to machine learning, neural networks, and general-purpose and neural network-specific architectures will be given in addition to discussing work objectives and memory structure.
- Chapter 2: Kendryte K210 architecture is described, its characteristics, the plates with which we have carried out the experiments and other alternative boards that we will use for comparison.
- Chapter 3: the software infrastructure that we have used in this project is described, as well as the tools used for training, compilation and execution of it.
- Chapter 4: list the steps to develop, since we chose the data set and the prerequisites before starting the training, up to the execution of the model, previously going through the compilation of the model.
- Chapter 5: focuses on the experiments carried out on both boards, the results obtained and a comparison with other alternative architectures.
- Chapter 6: a series of final conclusions are drawn after the completion of the work.

Capítulo 2

Infraestructura hardware. La arquitectura Kendryte K210

2.1. Descripción y características

Kendryte K210 [14] es un sistema en chip (SoC) que integra capacidades de procesamiento para visión y audio. K210 implementa dos soluciones de inteligencia artificial y una tercera híbrida; la primera es la *visión automática*, con capacidad para detectar objetos, clasificar imágenes, detectar y reconocer rostros, obtener tamaño y coordenadas de un objetivo en tiempo real y obtener el tipo del objetivo detectado también en tiempo real. La segunda solución es *audición automatizada*, ya que el chip viene con un procesador de audio de matriz de micrófonos de alto rendimiento para la orientación y formación de la fuente en tiempo real, la cual nos permite detectar la orientación de la fuente del sonido, despertador de voz y reconocimiento de voz. La tercera solución es una forma híbrida de las dos anteriores que combina ambas para lograr un mayor rendimiento. Además el SoC ofrece un rendimiento de 0.25 TOPS (TeraOperaciones Por Segundo) en su frecuencia base 400 MHz en procesamiento de redes neuronales. A grandes rasgos, incorpora los siguientes elementos principales, tal y como se observa en la figura 2.1:

- CPU: Equipa dos núcleos, cada uno con una FPU independiente de 64 bits de alto rendimiento y bajo consumo basado en el ISA RISC-V. La frecuencia de funcionamiento es regulable desde 400 MHz hasta 800 MHz, y una SRAM en chip de 8 MiB, además

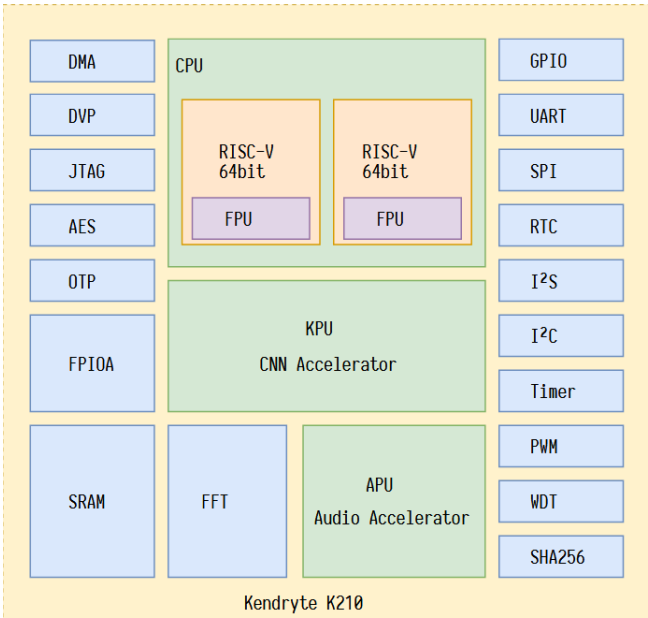

Figura 2.1: *Arquitectura del chip Kendryte K210 [14].*

de soportar cálculos de coma flotante de precisión doble y simple. Tiene 2 memorias caches, una para instrucciones y otra para datos, ambas de 32 KiB por núcleo.

- KPU: Es un procesador de redes neuronales capaz de acelerar procesos de inferencia sobre modelos previamente entrenados. Alguna de sus características son la integración de operaciones de convolución, normalización y aplicación de funciones de activación. El tamaño máximo del modelo en punto fijo para el trabajo a tiempo real es de 5 MiB a 5.9 MiB y además soporta kernels de convolución 1x1 y 3x3.

- APU: Es un módulo de preprocesamiento cuya tarea es el preproceso de la dirección de la voz y su salida de datos. Algunas de sus características son la capacidad para 8 canales de datos de entrada y el soporte de datos de ancho de entrada de 12, 16, 24 y 32 bits. Tiene una frecuencia de muestreo de hasta 182 KHz y utiliza DMAC para almacenar en la memoria del sistema datos de salida.

Estos son los periféricos que incorpora el chip:

- Memoria estática de acceso aleatorio (SRAM) de 8 MiB, divida en 2 partes: una de

uso general de 6 MiB donde se almacenarán los pesos del modelo y otros parámetros y otra de 2 MiB donde se almacenarán los mapas de características de entrada y salida.

- Memoria programable de una sola vez (OTP).
- Acelerador AES.
- Puerto de video digital (DVP).
- Acelerador FFT.
- Acelerador SHA256.
- Puerto UART.
- Temporizador de vigilancia (WDT).
- Interfaz de entrada / salida de uso general (GPIO).
- Controlador de acceso directo a memoria (DMAC).
- Bus de circuito inter-integrado (I2C).
- Interfaz periférica en serie (SPI).
- Sonido Inter-Integrado (I2S).
- Temporizador.
- Memoria de solo lectura (ROM).
- Reloj de tiempo real (RTC).
- Modulación de ancho de pulso (PWM).

2.2. Placas utilizadas

2.2.1. Maix Go

Está estructurada en un módulo principal, M1W (ver Figura 2.3) además de un administrador de energía ETA6002, una interfaz GPIO, una ranura para una tarjeta Micro-SD, un micrófono MEMS integrado, interfaces para el LCD y cámara DVP, un interruptor de marcación de tres vías y un botón de reinicio, un acelerómetro triaxial digital y un RTC (véase Figura 2.2).

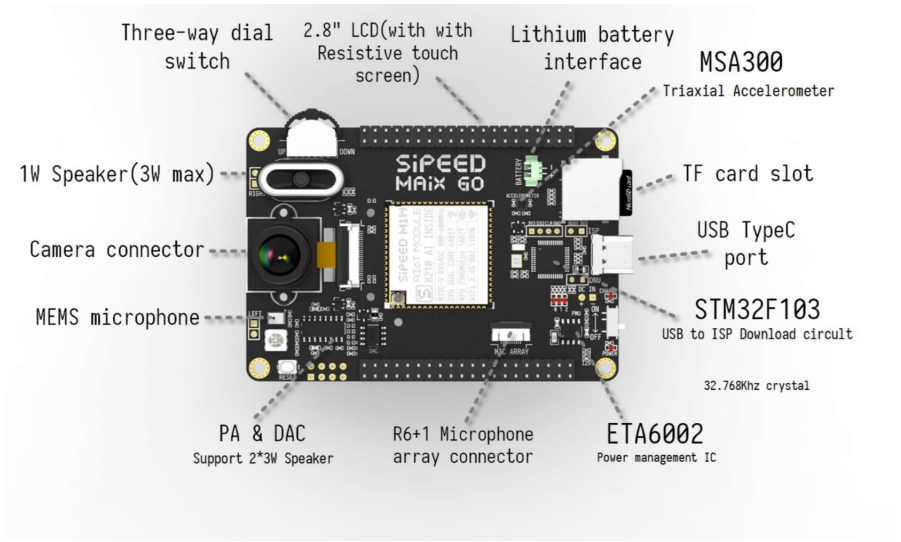

Figura 2.2: *Imagen de la placa Maix Go [7].*

El módulo M1W tiene incorporando el Kendryte K210, además de una memoria flash de 16 MB, un chip MCU WIFI (ESP8285) además de un DC-DC de 3 canales y un conector de antena IPEX (véase Figura 2.3).

2.2.2. Maix Bit

La placa Maix Bit está formada por el Kendryte K210, una memoria flash de 128 Mb, una ranura para una tarjeta Micro-SD, un micrófono MEMS integrado, interfaces para el LCD y cámara DVP y un DC-DC de 3 canales (véase Figura 2.4).

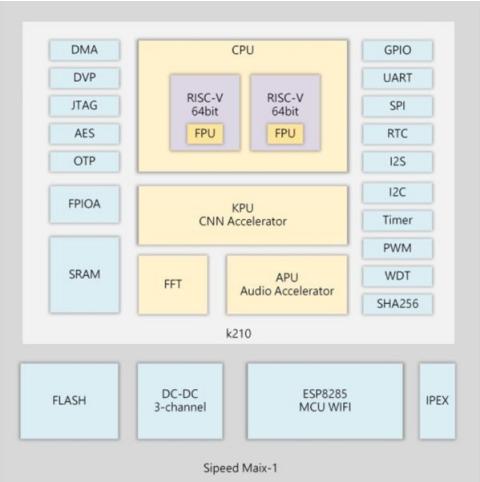

Figura 2.3: *Módulo M1W [4]*

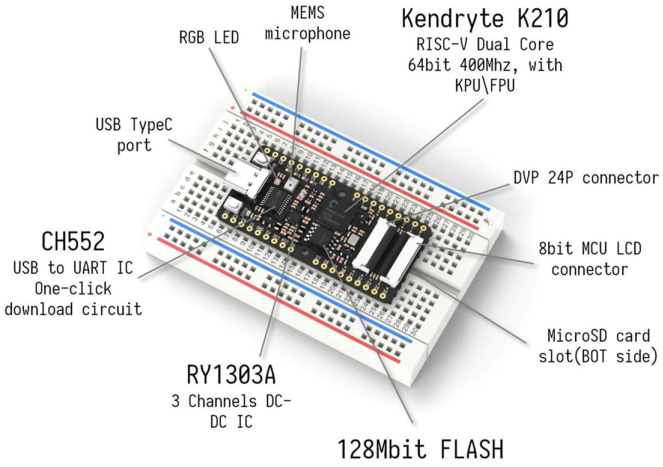

Figura 2.4: *Imagen de la placa Maix Bit [20]*

.

2.3. Comparativa con otras DSAs

2.3.1. Google Coral (Google Edge TPU)

Este *hardware* denominado Google Coral (ver 2.5) en formato USB 3.0 contiene un pequeño ASIC, el Edge TPU, desarrollado por Google el cual está diseñado específicamente para acelerar y ejecutar inferencia dando un consumo muy bajo de energía, es por esto que se puede encontrar en un tamaño tan pequeño como en un USB. Edge TPU puede realizar 4

TOPS (Tera Operaciones Por Segundo) usando una precisión de enteros de 8 bits en uno de los dos modos, frecuencia estándar. Usando el segundo modo, en máxima frecuencia puede multiplicar por 2 este rendimiento, por lo que podemos decir a primera vista que, comparado con el Kendryte k210, en cuestión de TOPS Google Coral estaría por encima, ya que el k210 es capaz de realizar 0.25 TOPS (Tera Operaciones Por Segundo), aunque en el capítulo 5 compararemos el rendimiento en fps y el consumo entre ambos.

Figura 2.5: *Google Coral USB. Imagen cortesía de https://coral.ai/products/accelerator/.*

2.3.2. Intel NCS2 (Myriad 2 VPU)

El Intel® Neural Compute Stick 2 (ver 2.6) contiene una versión del Myriad 2 VPU en formato USB que integra un procesador RISC con dos núcleos y una memoria LPDDR3 de 4 Gbytes. Este VPU es una unidad de procesamiento virtual destinado a la aceleración de inferencia, el cual utiliza a su favor los 12 núcleos que tiene integrados para obtener un mejor rendimiento. Puede alcanzar una frecuencia de 600 Mhz y alcanzar 1 TOPS (Tera Operaciones Por Segundo), por lo que se situaría por encima del Kendryte K210 en TOPS(Tera Operaciones Por Segundo), además tiene soporte para operaciones de enteros de 8, 16 ,32 y 64 bits y operaciones de punto flotante de 16 y 31 bits.

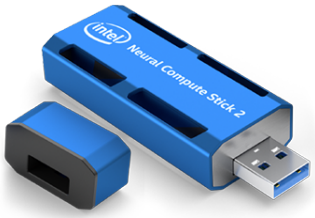

Figura 2.6: *Intel NCS2. Imagen cortesía de https://ark.intel.com/content/www/us/en/ark/products/140109/intel-neural-compute-stick-2.html .*

Capítulo 3

Infraestructura software

3.1. TensorFlow y TensorFlow Lite

TensorFlow [21] es una plataforma de código abierto que proporciona herramientas, librerías y una colección de flujos de trabajo para desarrollar y entrenar modelos usando lenguajes como Python o JavaScript sobre CPUs, GPUs o TPUs (*Tensor Processing Units*). Con ello, es capaz de construir y entrenar redes neuronales con el fin de desarrollar aplicaciones que puedan, por ejemplo, detectar y clasificar objetos. Esta plataforma nos proporciona una API de alto nivel llamada Keras [15], para construir y entrenar modelos de aprendizaje profundo.

En las versiones actuales de Tensorflow (2.0 en adelante), Keras está integrado dentro de Tensorflow, caso contrario de las versiones anteriores a 2.0. Tensorflow Lite [17] es un marco de trabajo de aprendizaje profundo de código abierto para la *inferencia* en dispositivos, donde podemos convertir un modelo TensorFlow en un buffer plano comprimido con el convertidor de TensorFlow Lite [10] (véase Figura 3.1); con ello, es posible partir de un comprimido en formato .TFLITE y cargarlo en un dispositivo móvil o empotrado y también si fuera necesario cuantizar el modelo convirtiendo los floats de 32 bits (la precisión por defecto) a enteros de 8 bits para una mayor optimización y eficiencia.

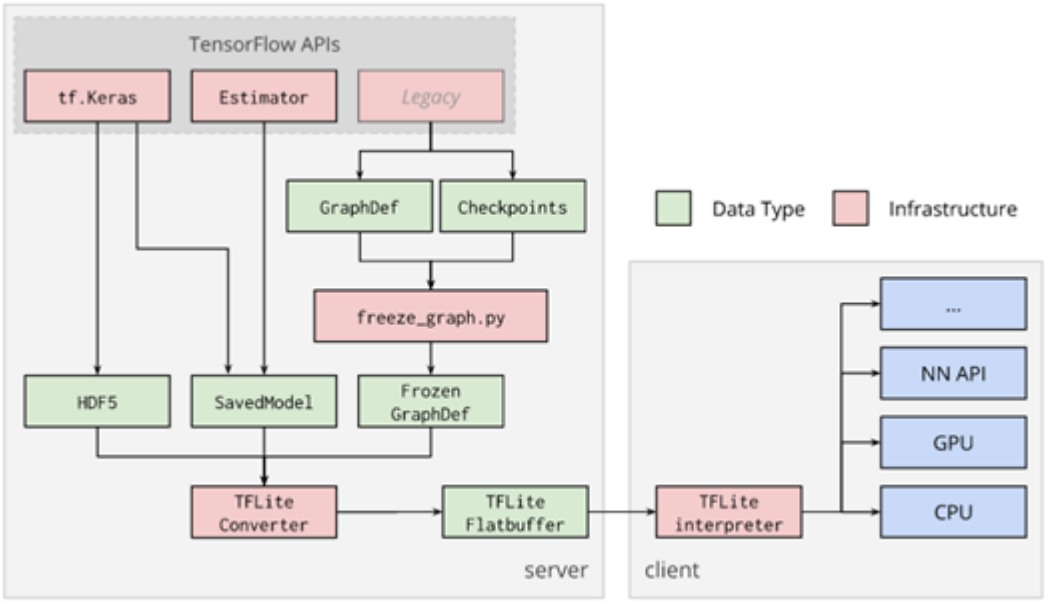

Figura 3.1: *Flujo de trabajo de TensorFlow Lite Converter [10]*

3.2. Kmodel

KModel es un formato de modelo empotrado utilizado actualente en placas Maix equipadas con el coprocesador Kendryte K210. Este modelo nos permite utilizar dos tipos de precisiones: *float32*, que ofrece mayor precisión a coste de más memoria y menos velocidad; e *int8* (enteros de 8 bits) que proporcionan una mayor velocidad a un bajo coste de memoria, pero con menor precisión.

Podemos encontrar dos versiones de este modelo dependiendo de la versión del compilador que usemos al compilar el modelo en formato TFLite. Usando la versión 0.1.0 RC5 o anteriores del compilador NNcase (véase Sección 3.3) resultaría en un KModel versión 3 y con la versión 0.2.0 Alpha1 en adelante del compilador un KModel versión 4. Hay que tener en cuenta que, por el momento, la versión 4 del KModel no soporta algunas de las operaciones claves en inferencia, y por tanto éstas son mapeadas sobre la CPU; entre ellas destaca la operación MatMul, por lo que se obtiene un peor rendimiento frente a la versión 3. Su consumo de memoria es, además, mayor. Sin embargo, soporta un número de

modelos de red neuronal mucho mayor que las versiones anteriores. KMODEL actualmente soporta operadores de TFLite,[19], Caffe [18] y ONNX [1].

3.3. La herramienta NNcase

NNcase [2] es una herramienta para compilar redes neuronales para aceleradores de IA (Inteligencia Artificial) y para realizar inferencia, siendo `ncc` la orden de línea de comando ofrecida por la herramienta para interactuar con el usuario. `ncc` ofrece dos comandos diferenciados (véase 3.2 para más información):

- `compile`: compila los modelos entrenados (TFLite, Caffemodel y Onnx) a KMODEL.
- `infer`: ejecuta el KMODEL; en este caso, `ncc` almacenará los tensores de salida del modelo en archivos `.bin` en formato `NCHW`.

```
DESCRIPTION
NNCASE model compiler and inference tool.

SYNOPSIS
    ncc compile <input file> <output file> -i <input format> [-o <output
        format>] [-t <target>] [--dataset <dataset path>] [--dataset-format
        <dataset format>] [--inference-type <inference type>] [--input-mean
        <input mean>] [--input-std <input std>] [--dump-ir] [--input-type <input
        type>] [--max-allocator-solve-secs <max allocator solve secs>]
        [--calibrate-method <calibrate method>] [-v]

    ncc infer <input file> <output path> --dataset <dataset path>
        [--dataset-format <dataset format>] [--input-mean <input mean>]
        [--input-std <input std>] [-v]
```

Figura 3.2: *Parámetros de los comandos* `compile` *e* `infer` *[3].*

La herramienta admite la cuantificación posterior del modelo con el conjunto de datos de calibración, y admite `float` y cuantificación de enteros de 8 bits. Como ya hemos comentado en el anterior apartado, según la versión que utilicemos del compilador obtendremos una versión distinta del KMODEL, aunque en el momento de realizar este trabajo solo existen las

mencionadas. Para la compilación en el caso de usar modelos TFLite tendremos que usarlos en punto flotante, ya que `nncase` se encargará de la cuantización.

Capítulo 4

Entrenamiento, conversión e inferencia de un modelo sobre la arquitectura K210

4.1. Fase de entrenamiento

Para entrenar un modelo Tensorflow, el primer paso consiste en preparar el entorno; para ello es necesario instalar Tensorflow (para este proyecto se ha utilizado la versión 1.13.1 debido a que la versión 2.1 presentaba problemas claros de rendimiento que se comentarán en el capítulo 5 a la hora de ejecutar el KModel v4). También necesitaremos instalar los drivers de Nvidia, CUDA y varias herramientas de compilación de la misma para el entorno de entrenamiento; para ello, es posible utilizar una imagen *Docker* que ofrece TensorFlow o, como en nuestro caso, instalar Conda y usar el comando:

```
conda install tensorflow-gpu==1.13.1
```

Dicho comando instalará todas las dependencias necesarias y las versiones correspondientes. Si se desea modificar la versión, solo es necesario modificar el número de versión indicado anteriormente. De forma similar, si se desea usar la CPU en vez de la tarjeta gráfica, sería necesario eliminar la parte de GPU de la orden.

Una vez instalado el entorno, es necesario un conjunto de datos de entrada. En nuestro caso, dicho conjunto es `ImageNet ILSVRC 2012` que corresponde a un conjunto etiquetado

con mil clases distintas, y miles de imágenes en cada clase. Este conjunto será el utilizado en el proceso de entrenamiento. Con respecto al modelo, se utilizará el modelo preentrenado *Mobilenet v1* de Keras, que será reentrenado con nuestro conjunto de datos.

Mobilenet [13] es una clase de modelos eficientes llamados MobileNets para aplicaciones de visión integradas y móviles, basado en una arquitectura optimizada que utiliza convoluciones separables para construir redes profundas con un tamaño contenido. Se introducen 2 hiperparámetros:

1. *Multiplicador de anchura* (α): Se usa para "adelgazar" una red de manera uniforme en cada capa. Dada una capa y un multiplicador de anchura α, el número de canales de entrada M sería αM y el número de canales de salida N sería αN. Esto tiene el efecto de reducir el coste computacional y el número de parámetros cuadráticamente (alrededor de α^2). Se puede aplicar a cualquier estructura de modelo para definir un nuevo modelo más pequeño. Los valores de alpha están entre 0 y 1, con valores típicos de 0.25, 0.5, 0.75, que corresponden a modelos reducidos, y 1 es el estándar.

2. *Multiplicador de resolución* (ρ): Se aplica en la imagen de entrada, reduciendo la representación interna de cada capa por dicho factor multiplicador. Aplicamos este hiperparámetro implícitamente cuando ajustamos la resolución de entrada, es decir al poner como resolución 224, 192, 160 o 128. Esto hace reducir el coste computacional alrededor de ρ^2.

La arquitectura de MobileNet se puede apreciar en la figura 4.1:

En este trabajo hemos entrenado 3 modelos, usando un $\rho = 224$ y $\alpha = 0.25$, 0.5 y 0.75, por lo que tendremos 3 modelos distintos reducidos para comparar, tal y como se muestra en la Tabla 4.1.

Seleccionamos los modelos pre-entrenados MobileNet v1 de Keras sin las capas *top*, es decir, sin las capas totalmente conectadas y sin las de salida, ya que las añadiremos nosotros

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figura 4.1: *Arquitectura del cuerpo MobileNet [13]*

Modelos	Millones de MACS	Millones de parámetros
mobilenet_7_5_224_tf_no_top.h5	325	2.6
mobilenet_5_0_224_tf_no_top.h5	149	1.3
mobilenet_2_5_224_tf_no_top.h5	41	0.5

Cuadro 4.1: *Modelos MobileNet v1 seleccionados para el entrenamiento [9].*

manualmente. Antes de comenzar el proceso de entrenamiento, es necesario modificar el archivo `mobilenet.py` y cambiar el relleno (*padding*), debido a que K210 usa un método de relleno diferente al predeterminado de Keras, lo que hace imprescindible su adaptación. Este método rellena los ceros de alrededor (izquierda, arriba, derecha, abajo), pero Keras utiliza un relleno de derecha y abajo. Para ello usaremos la función `ZeroPadding2D` para establecer el método de relleno que deseamos. Después, ya que habíamos elegido un modelo sin las capas totalmente conectadas y sin las de salida tenemos que añadirlas siendo así la adición de una capa denominada *dropout* para reducir el sobreajuste y una capa *dense* (una capa oculta totalmente conectada) con activación *softmax*.

Con estas modificaciones, es posible comenzar el entrenamiento; en este caso se utilizará

un número de iteraciones de 20 (*epoch*), 50 pasos por cada iteración, y un tamaño de lote de 256 para el entrenamiento. Una vez terminado el entrenamiento (en un tiempo aproximado de 4 horas cada modelo), disponemos del modelo Keras listo para ser convertido a TFLite.

Los modelos entrenados al ser reducidos por el parámetro α tienen menor número de parámetros y MACS, por lo que conllevan a un tamaño menor a cambio de menor precisión como se puede apreciar en la tabla 4.2.

Modelo	Precisión
mobilenet_7_5_224.h5	0.60
mobilenet_5_0_224.h5	0.54
mobilenet_2_5_224.h5	0.38

Cuadro 4.2: *Precisiones obtenidas en los 3 modelos.*

Podemos observar en la Figura 4.2 como únicamente modificando el número de iteraciones, la precisión del modelo se ve afectada, llegando a un número de iteraciones cercano a 20 donde se empieza a estabilizar.

4.2. Fase de conversión

Una vez se dispone de los modelos entrenados, procederemos a usar la herramienta TFLite Converter que nos proporciona TensorFlow para convertir un modelo Keras en un modelo TFLite. Para ello, introduciremos en el comando TFlite_convert [10] el parámetro keras_model_file para indicar el archivo de entrada que contendrá nuestro modelo Keras, el archivo de salida (que será el archivo con formato `tflite`), con un tipo de precisión en punto flotante y la forma de la entrada (input_shape) de `1,224,224,3` siendo el tamaño de lotes (*batch size*), altura y anchura de 224 y 3 canales, respectivamente.

A partir de los tres modelos, es posible proceder con la compilación con la herramienta NNcase. Como uno de los objetivos es llevar a cabo una comparación de las dos versiones del KModel, usaremos la versión 0.2.0 Beta3 par obtener un KModel v4 y la versión v0.1.0 RC5 para un KModel v3 (hay que tener en cuenta que los modelos entrenados con

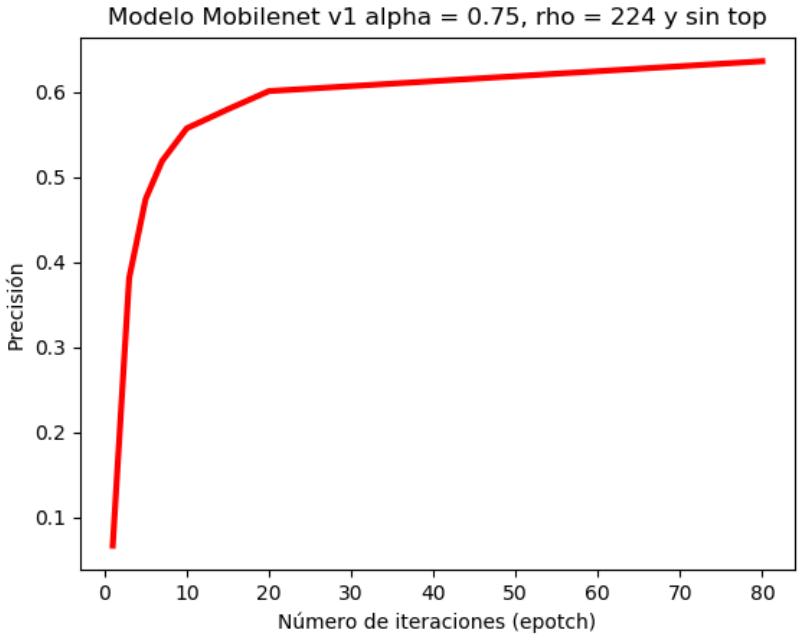

Figura 4.2: *Efecto del número de iteraciones(*epoch*) sobre la precisión de un modelo MobileNet v1 en la fase de entrenamiento*

la versión de Tensorflow 2.1, no pueden ser compilados por esta versión del compilador). En ambos casos, no podemos comprobar la precisión *float32* debido a que no hay suficiente memoria en el KPU para un modelo tan grande, luego vamos a cuantizarlo con la precisión de enteros de 8 bits y para ello necesitamos un pequeño conjunto de imágenes a modo de perfilado. Con todo esto podemos proceder a la compilación y obtención de los 6 modelos que usaremos para inferencia.

4.3. Fase de inferencia

En este apartado realizaremos la inferencia en tiempo real con las dos placas, Maix Bit y Maix Go realizada en dos enternos: el de MicroPython y Standalone SDK C. Para el entorno de MicroPython necesitaremos almacenar en las memorias flash de las placas el firmware correspondiente (en la dirección 0x00000000) y el modelo (en la dirección 0x00200000).

La herramienta MaixPy IDE [5] nos permitirá la edición del script de ejecución de la

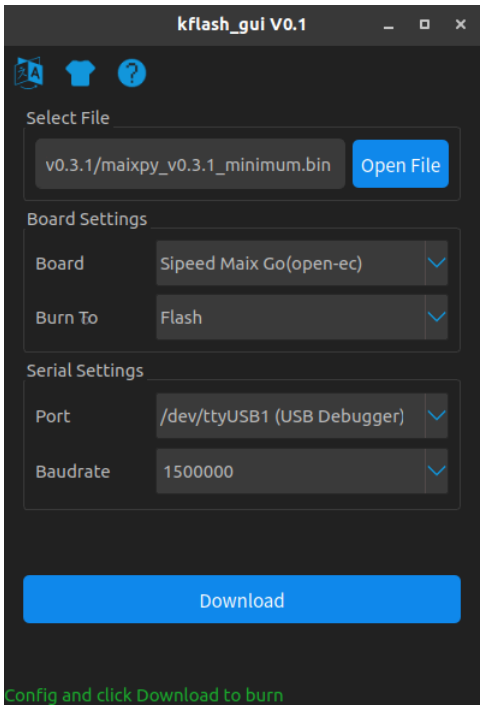

Figura 4.3: *Ejemplo de la herramienta Kflash introduciendo el firmware en la placa Maix Go [6].*

aplicación, así como también observar las imágenes de la cámara en tiempo real; para ello la versión de este firmware tendrá que ser superior a 0.3.1 y tendremos que usar la versión mínima compatible con el IDE, ya que no hay suficiente memoria para un modelo y firmware de gran tamaño. En el siguiente enlace se pueden encontrar las versiones del firmware utilizadas: . Para introducir tanto el firmware como el modelo en la memoria flash usaremos la herramienta `Kflash` [6], que nos permitirá almacenar modelo y firmware en la memoria *flash*, como se puede observar en la Figura 4.3.

Una vez preparados tanto el firmware como el modelo, simplemente será necesario utilizar el IDE proporcionado y ejecutar un *script* en el que realizaremos la inferencia en tiempo real, pudiéndose observar los resultados a través de la pantalla LCD.

Con el KModel versión 4 es necesario, de momento, usar un firmware especial (MaixPy_support_Kmo V4) que nos proporciona el fabricante, ya que las últimas versiones disponibles no son compatibles con esta versión del KModel. Si usamos el SDK Standalone V0.5.6 el proceso es

similar, pero el firmware debe ser compilado manualmente, e incluye la funcionalidad de inferencia. En este caso, debido a su reducido tamaño, no existe ningún problema de espacio para almacenar tanto el modelo como el firmware.

Capítulo 5

Resultados experimentales

En este capítulo evaluaremos mediante los resultados experimentales realizados el rendimiento, consumo y eficiencia energética de la arquitectura Kendryte K210, dividiendo el capítulo en 4 secciones:

- La primera sección está dedicada a la descripción de los modelos MobileNetV1 elegidos y a las distintas versiones del KModel.
- La segunda sección aporta la descripción de los experimentos realizados y la exposición de los resultados obtenidos en los mismos.
- La tercera sección relata la descripción de la infraestructura utilizada en los experimentos para la medición del consumo energético así como la exposición de sus resultados.
- La cuarta sección realiza una comparación entre las distintas arquitecturas de propósito específico tanto a nivel de rendimiento, como de consumo energético y eficiencia energética.

5.1. Descripción del modelo seleccionado y datos de entrada

Los modelos elegidos antes del entrenamiento son los modelos MobileNetV1 de ρ= 244 y de $\alpha = 0.75$, 0.25 y 0.5; éstos son modelos reducidos del base ($\alpha = 1$). Una vez entrenados

y convertidos a `TFlite`, usaremos dos versiones del compilador nncase para conseguir 6 modelos: 3 KModel versión 1 (usando la versión NNcase 0.1.0 RC5) y 3 KModel versión 2 (usando la versión NNcase v0.2.0 Beta3), los 6 con una precisión de enteros de 8 bits, ya que por las limitaicones en memoria no se permite, para estos modelos, la implementación con precisión en punto flotante de 32 bits. Además, el valor del tamaño de lote (*batch size*) no ha podido ser aumentado a más de 1 por el mismo problema, luego todos los resultados mostrados son para un valor de *batch size* igual a 1. Es necesario destacar que que los experimentos son realizados en ambas placas, Maix Go y Maix Bit, pero en la Maix Bit se eliminó la pantalla LCD para conseguir un resultado más preciso sobre el Kendryte K210 tanto a nivel de rendimiento como de consumo energético.

Como se apuntó en el Capítulo 4, según el modelo con mayor reducción obteníamos una menor precisión; en cambio, según hemos observado a través de 10 repeticiones de inferencia sobre las placas en el entorno de MicroPython para obtener un resultado fiable, conseguimos una mayor velocidad, siendo así unos 24 FPS, 44 FPS y 101 FPS en los modelos de $\alpha = 0.75$, 0.50 y 0.25, respectivamente, con versión del KModel 3. En la versión 4 del KModel, debido a que algunas operaciones están siendo ejecutadas en la CPU en vez del KPU obtenemos velocidades menores, siendo así 8.3 FPS, 17.50 FPS y 37.15 FPS en los modelos de $\alpha = 0.75$, 0.50 y 0.25 respectivamente, además de consumir esta versión algo más de memoria (unos 360KB).

Por problemas con la versión 4 del KModel no ha sido posible ejecutar el firmware en C por lo que no hay datos de rendimiento de dicho modelo.

5.2. Evaluación de rendimiento

5.2.1. Descripción de los experimentos

Los experimentos se han realizado en las placas Maix Go y Maix Bit con los modelos mencionados en el anterior apartado en una distribución Ubuntu 18.04. El entrenamiento se ha llevado a cabo utilizando Tensorflow versión 1.13.1, ya que los modelos entrenados con

la versión de Tensorflow 2.1 presentaban una gran disminución de rendimiento en términos de *frames* por segundo (en el KMODEL v4 más grande, con un α=0.75 se obtenía 1.84 FPS frente a los 8.24 FPS que se obtenían con la anterior versión de Tensorflow), probablemente debido al hecho de que la herramienta de compilación está todavía en fase de desarrollo. Los resultados se han obtenido después de 10 repeticiones en ambas placas para asegurar unos resultados estables.

5.2.2. Resultados experimentales

Los resultados de rendimiento obtenido varían dependiendo de la versión usada del KMODEL y del modelo MobileNetV1 utilizado. Como podemos observar en la figura 5.1, la diferencia entre los distintos modelos MobileNetV1 es clara, observando alrededor de 24 FPS en el modelo más grande con $\alpha = 75$ hasta 101 FPS con el modelo más pequeño, por lo que a pesar de que la reducción del tamaño del modelo afecte a su precisión disminuyéndola, por otro lado aumenta considerablemente su velocidad. La diferencia que se observa entre las 2 versiones del KMODEL se debe a que ciertas operaciones son realizadas en el CPU en vez de la KPU, por lo que la velocidad disminuye, aunque una vez se termine de desarrollar completamente la versión 4, en teoría tendremos el mismo o mejor rendimiento que en la versión 3 del KMODEL.

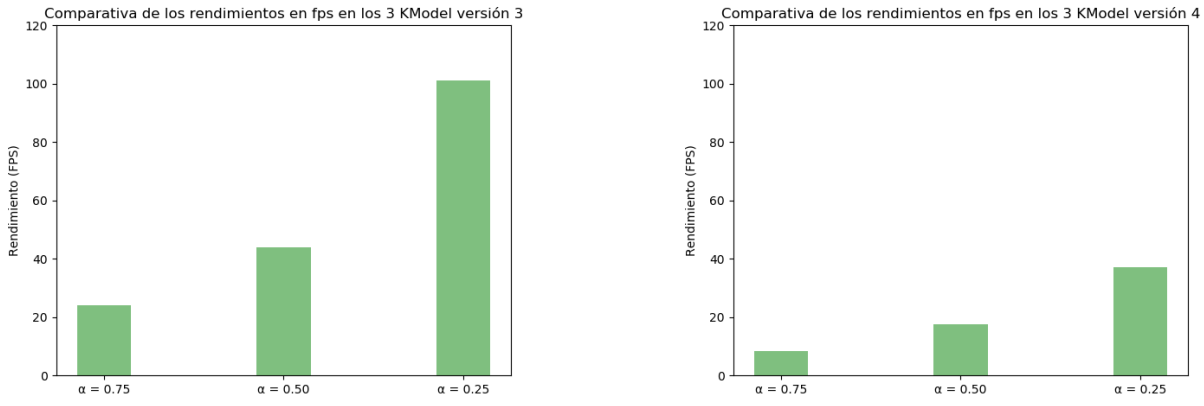

Figura 5.1: *Comparación rendimiento en fps de los* KMODEL *v3 (izquierda) y* KMODEL *v4 (derecha).*

5.3. Consumo energético

5.3.1. Infraestructura de medición

Para el análisis de consumo hemos usado la placa SmartPower2 [11]. SmartPower2 (ver figura 5.2) es un dispositivo de bajo consumo dedicado específicamente a la medición de consumo en continua. Además, ofrece los datos de medición a través de una interfaz WIFI, por lo que su uso es muy conveniente para nuestro escenario. Para llevar a cabo las mediciones, hemos conectado mediante cable USB a USB tipo C a las placas y mediante el módulo WIFI ESP12E monitorizaremos el consumo mientras realizamos las pruebas experimentales.

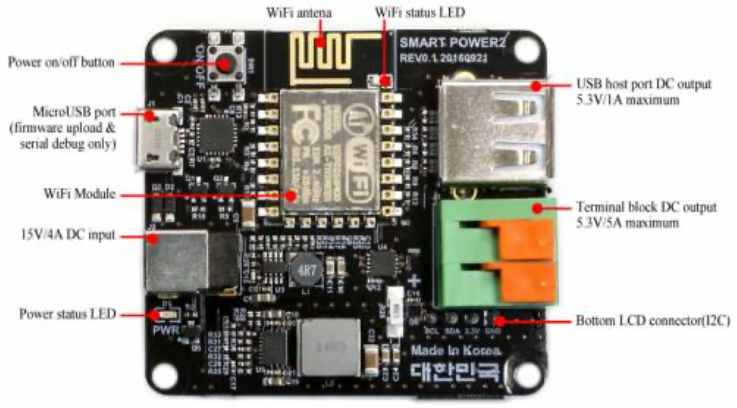

Figura 5.2: *Imagen de la infraestructura de medición SmartPower2[11]*

5.3.2. Resultados experimentales

En este apartado medimos el consumo mientras se realiza la inferencia en ambas placas, observamos que no hay cambio en el consumo entre modelos reducidos y que en el entorno de MicroPython es algo más costoso que en el de C, siendo este último alrededos de 0.7W y 140mA en la placa Maix Bit y 1.1W y 0.220mA en la placa Maix Go, mientras que en el entorno de MicroPython nos encontramos con 0.880W y 180mA en la Bit y 1.6W y 0.320A en la Go, como era de esperar, la Maix Bit tiene un consumo menor a pesar de tener los mismos componentes que la Maix Go y es por que para acercarnos más al rendimiento del K210 no se usó la pantalla LCD para estas mediciones por lo que tenemos un menor consumo

(ver figura 5.3).

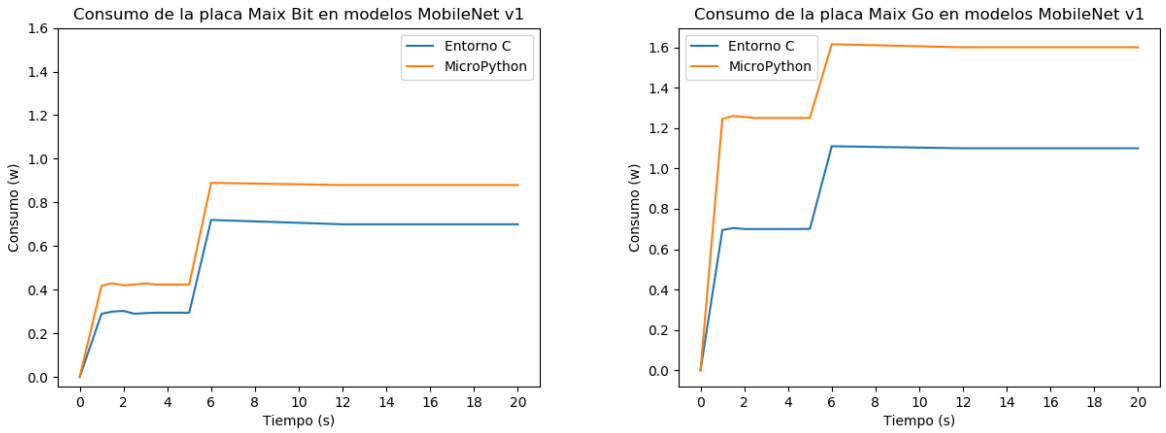

Figura 5.3: *Comparación de consumo entre las placas Maix Bit (izquierda) y Go (derecha). Se obtiene el mismo resultado en los 3 modelos, por lo que se muestra el consumo del modelo MobileNet v1 $\rho = 224$ y α=75. De los 20 segundos de ejecución los primeros 5 segundos está en estado inactivo, y el resto realizando inferencia.*

5.4. Comparativa con DSAs (Arquitecturas de Propósito Específico) alternativas

En esta sección se realiza una comparativa de rendimiento (en términos de fps y eficiencia energética usando operaciones pico teóricas en este último término) y de consumo en vatios con respecto a otras dos arquitecturas de propósito específico (Google Coral e Intel NCS2). Los resultados experimentales para ellas han sido extraídos de [16] y [8], siempre realizando comparaciones con un tamaño de lote (batch size) de 1.

En los resultados recogidos de [8] utilizan en sus experimentos el modelo de MobileNetV1 con $\alpha = 0.75$ en las placas Coral acelerador USB y en el Intel NCS2 obteniendo unos resultados de 20 FPS y 11,4 FPS, respectivamente, mientras que nosotros hemos obtenido 24 FPS en el KModel v1 y 8,4 FPS en el KModel v2 (los FPS son los mismos en las placas Maix Go y Bit). Es posible concluir que en cuestión de rendimiento el Kendryte K210 es más rápido que las otras dos arquitecturas de propósito específico. Por otro lado,

el consumo obtenido de los resultados de [16] dan un consumo de alrededor de 1.1 W en el acelerador USB Coral y alrededor de 2 W en el Intel NCS2.

En nuestro caso hemos obtenido en el entorno de MicroPython 0.880W y 1.6W en las Maix Bit y Go respectivamente mientras que en el entorno de C se obtiene menor consumo resultando así en 0.7W y 1.1W en las Maix Bit y Go respectivamente, por lo que en general nuestro hardware tiene un consumo menor exceptuando en la placa Maix Go usándose un firmware y ejecutándose en un entorno de MicroPython como se puede apreciar en la Figura 5.4.

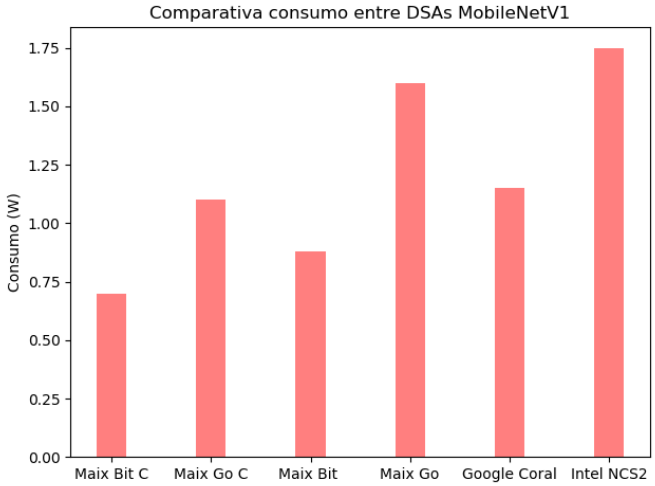

Figura 5.4: *Comparación de consumo entre las distintas DSAs, las 2 primeras representan las placas usadas en este proyecto en el entorno de C (firmware y script de ejecución) y las 2 siguientes en el entorno de MicroPython.*

También podemos comparar la eficiencia energética, en este caso el rendimiento pico teórico que, aunque en la práctica sea complicado llegar a estos niveles, podemos hacernos una idea que como de cerca o lejos está la arquitectura K210 con respecto a las otras alternativas. Como se observa en la figura 5.5 vemos que la K210 es capaz de realizar 0.833 TOPS/W siendo el rendimiento más bajo de todas las arquitecturas de propósito específico, pero no muy alejada de la Intel NCS2 que ofrece 1 TOPS/W; comparando con la Google Coral hay un margen más amplio ya que esta última da un rendimiento de 2 TOPS/W siendo teóricamente la arquitectura que más eficiencia energética proporciona. A pesar de

que la K210 (una arquitectura en desarrollo) es la que menor rendimiento proporciona en términos de TOPS/W, recientemente los desarrolladores lanzaron una nueva versión de esta arquitectura para nuevas placas que, con la misma frecuencia y consumo, es capaz de alcanzar 3.33 TOPS/W haciendo que sea la arquitectura con más eficiencia energética de entre las nombradas previamente.

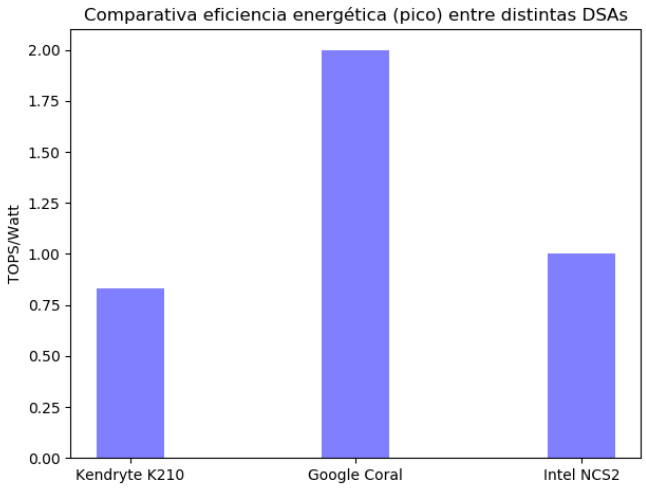

Figura 5.5: *Comparación de eficiencia energética (rendimiento pico teórico) en términos de TOPS/Watt entre las distintas DSAs.*

Capítulo 6

Conclusiones

6.1. Conclusión

En este trabajo hemos realizado experimentos sobre las dos placas que contienen la arquitectura Kendryte K210, las placas Maix Go y Bit para poder obtener los resultados de rendimiento y consumo y poder compararlas con otras arquitecturas de propósito específico como Google Coral (equipada con el procesador Google Edge TPU) y el Intel NCS2 (equipada con el procesador Myriad 2 VPU), usando modelos similares[1]. Hemos podido comprobar como, en general, la arquitectura del K210 tiene un consumo bastante menor al de las dos alternativas mientras que el rendimiento en términos de FPS el acelerador USB Coral tiene un rendimiento muy cercano al del k210 estando este último algo por encima, aunque es bastante superior al Intel NCS2 (24 FPS vs 11.4 FPS) en la versión del KMODEL 3 ya que la versión 4, una vez arreglados los problemas que tiene actuales, tendrán un rendimiento igual al de la versión 3 como hemos visto en el capítulo 5. A pesar de los buenos resultados del Kendryte K210 tiene sus limitaciones ya que al no tener suficiente memoria no se podía realizar experimentos modificando el tamaño de lote (*batch size*) o utilizar modelos más grandes, sumándole a los problemas que se irán solucionando con el tiempo sobre la herramienta de compilación ya que está en desarrollo.

[1]La diferencia en las comparaciones ha sido la reducción del modelo con el parámetro α. En [16] se utiliza un valor de 1.0 mientras que en este proyecto hemos usado 3 versiones posibles de reducción 0.75, 0.5 y 0.25).

Por último, mencionar que, como se ha comentado anteriormente, el Kendryte K210 tiene capacidad para realizar 0.83 TOPS/W (rendimiento pico teórico) a una frecuencia de 400 MHZ por lo que está algo por debajo del Intel NCS2 y bastante alejado del Google Coral en estos términos, pero recientemente han desarrollado nuevas placas con una nueva versión de esta arquitectura (la Kendryte K210) que consigue en la misma frecuencia alrededor de 3,33 TOPS/W (rendimiento pico teórico) llegando a superar a ambas arquitecturas.

6.2. Conclusion

In this work we have carried out experiments on the two boards that contain the Kendryte K210 architecture, the Maix go and Bit boards in order to obtain the performance and consumption results and be able to compare them with other specific purpose architectures such as Google Coral (Google Edge TPU) and the Intel NCS2 (Myriad 2 VPU), using similar models[2]. We have verified how in general the architecture of the K210 has a considerably lower consumption than the two alternatives (The Maix Go board gives a little more consumption because the LCD screen was used) while in the trend in terms of FPS the Coral USB accelerator has a performance very close to the K210, the latter is somewhat higher, although it is much higher than the Intel NCS2 (24 FPS vs 11.4 FPS)in the version of the KModel 3 since version 4, once the current problems are fixed, will have a performance equal to that of version 3 as we have seen in chapter 5. Despite the good results of the Kendryte K210, it has its limitations since it does not have enough memory, it will not be possible to carry out experiments to modify the *batch size* or use larger models, adding to the problems that will be solved over time. about the build tool already which is in development.

Finally, it is worth mentioning that, as previously mentioned, the Kendryte K210 has the capacity to perform 0.83 TOPS/W (theoretical peak performance) at a frequency of 400

[2]The difference in the comparisons has been the reduction of the model with the parameter-meter where in the paper of [16] they have used a value of 1.0 while in this project we have used 3 possible versions of reduction 0.75, 0.5 and 0.2)

MHZ, so it is somewhat below the Intel NCS2 and quite far from the Google Coral in these terms, but recently they have developed new boards with a new version of this architecture (Kendryte K210) that achieves at the same frequency around 3.33 TOPS / W (theoretical peak performance), exceeding both architectures.

A continuación se proporciona el enlace al repositorio donde se encuentra el código usado en el entrenamiento, inferencia y crear el firmware; los modelos usados, etc.
https://github.com/SalvadorAlbarran/TFG2020

Bibliografía

[1] Nncase. https://github.com/kendryte/nncase/blob/master/docs/onnx_ops.md, accedido 2020.

[2] Nncase. https://github.com/kendryte/nncase, accedido 2020.

[3] Nncase. https://github.com/kendryte/nncase/blob/master/docs/USAGE_EN.md, accedido 2020.

[4] Sipeed. Sipeed M1W Specification V1.11, accedido 2020.

[5] Sipeed. https://github.com/sipeed/MaixPy, accedido 2020.

[6] Sipeed. https://github.com/sipeed/kflash_gui, accedido 2020.

[7] Sipeed. sipeed maixgo datasheet v1.1, accedido 2020.

[8] Alasdair Allan. Benchmarking edge computing, comparing google, intel, and nvidia accelerator hardware. 2019.

[9] François Chollet. https://github.com/fchollet/deep-learning-models/releases/tag/v0.6/, accedido 2020.

[10] TensorFlow Lite Converter. https://www.tensorflow.org/lite/convert. (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/g3doc/convert), accedido 2020.

[11] Hardkernel. https://wiki.odroid.com/accessory/power_supply _battery/smartpower2, Accedido 2020.

[12] Juan Rodríguez Hortalá. Aprendizaje a partir de ejemplos. Apuntes de la asignatura Sistemas inteligentes 2020.

[13] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, and Hartwig Adam Marco Andreetto. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861, https://arxiv.org/abs/1704.04861.

[14] Canaan Inc. Kendryte k210 datasheet, 2018-10-14. versión 0.1.5, accedido 2020.

[15] Keras. https://keras.io/, accedido 2020.

[16] Leandro Ariel Libutti, Francisco D. Igual, Luis Piñuel, Laura De Giusti, and Marcelo Naiouf. Benchmarking performance and power of usb accelerators for inference with mlperf*. 2020.

[17] TensorFlow Lite. https://www.tensorflow.org/lite, accedido 2020.

[18] NNcase. https://github.com/kendryte/nncase/blob/master/docs/caffe_ops.md, accedido 2020.

[19] NNcase. https://github.com/kendryte/nncase/blob/master/docs/tflite_ops.md, accedido 2020.

[20] Sipeed. Sipeed Maix-Bit Specifications_EN V2.0, accedido 2020.

[21] TensorFlow. www.tensorflow.org, accedido 2020.