Evaluación de Aceleradores de Diseño Específico en Apple Silicon para Aplicaciones de Propósito General

Por

Álvaro Corrochano López

UNIVERSIDAD COMPLUTENSE
MADRID

Máster en Ingeniería Informática

Facultad de informática

Dirigido por

Carlos García Sánchez

Evaluation of Specific Design Accelerators in Apple Silicon for General-Purpose Applications

Madrid, 2024–2025

Calificación: 10

Convocatoria: Junio 2025

Evaluation of Specific Design Accelerators in Apple Silicon for General-Purpose Applications

Evaluación de Aceleradores de Diseño Específico en Apple Silicon para Aplicaciones de Propósito General

Convocatoria Junio 2025

Calificación: 10

Memoria que se presenta para el Trabajo de Fin de Máster

Álvaro Corrochano López

Dirigido por

Carlos García Sánchez

Facultad de informática

Universidad Complutense de Madrid

Madrid, 8 de julio de 2025

Parte A

Introducción

Resumen

La computación de altas prestaciones (*High Performance Computing*, HPC) trata de resolver problemas complejos gracias al uso eficiente de la potencia de los ordenadores.
A lo largo de la historia, algunos aparatos como las GPU han demostrado potencial en este campo para resolver problemas de propósito general en lugar de los específicos para los que se diseñaron. Siguiendo esta línea, el presente trabajo evalúa el potencial de los aceleradores neuronales de Apple para este tipo de cargas de trabajo.

Mediante la adaptación de algoritmos clásicos de HPC en el *Apple Neural Engine* (ANE), este trabajo demuestra que ofrece un rendimiento competitivo y, de forma destacada, una eficiencia energética superior a la de la CPU y la GPU.

Palabras clave: Aceleración, ANE, NUC, Evaluación, HPC, Apple, Jacobi, GEMM, Multigrid.

Abstract

High performance computing attempted to solve complex problems using computer power.
Throughout history, some devices like GPUs show the potential in this field to solve general-purpose problems instead of the specific problems that they were created for. Following this line, the present work evaluates the potential of Apple's neural accelerators for this type of workload.

By adapting classic HPC algorithms on the Apple Neural Engine (ANE), this work demonstrates that it offers competitive performance and, outstandingly, superior energy efficiency compared to the CPU and GPU.

Key Words: Hardware acceleration, ANE, NUC, Evaluation, HPC, Apple, Jacobi, GEMM, Multigrid.

Cita

"Corresponde al general ser tranquilo, reservado, justo y metodico."

Sun Tzu

Agradecimientos

Agradezco especialmente a mi tutor, Carlos, por todo el apoyo que me ha dado durante el proyecto. Sin su guía, esto no hubiera sido posible.
También quiero agradecer al grupo de investigación ArTeCS por todos estos meses donde hemos compartido espacio de trabajo y momentos divertidos que han ido más allá del laboratorio.
Gracias también a mis compañeros de máster Iulius, Álvaro, Cristian y Chechu por estos dos años. Sin duda, han sido más amenos gracias a ellos.
Agradezco también a mi pareja, Ariel, por animarme y apoyarme durante este camino.
Por último, quiero agradecer a mi familia, en especial a mi madre Josefina, por todo el apoyo durante el máster y mi vida en general.

Muchas gracias a todos.

Índice general

Parte B

Capítulos

Capítulo 1

Introducción

1.1. Conocimientos Previos

En esta sección se explicarán algunos conceptos previos relevantes para poder entender este trabajo, como la naturaleza de los dispositivos que se irán comentando.

1.1.1. Computación de altas prestaciones

La computación de altas prestaciones (conocida como HPC por sus siglas en inglés) es un tema de alto interés actual, especialmente en unidades de procesamiento neuronal (NPU por sus siglas en inglés) [1] [2].

La HPC siempre ha sido uno de los escenarios más punteros para el futuro de la informática. Consiste en utilizar computadores de alto rendimiento para solventar problemas enormes en tiempo real, como podría ser el tratamiento de datos masivos, utilizando generalmente servidores (computadores capaces de atender peticiones de clientes) [3].

Por poner algunos ejemplos, gracias a la computación de altas prestaciones podemos secuenciar un genoma humano en menos de un día, detectar fraudes rápidamente con métodos de análisis de riesgos, predecir el clima de manera más precisa usando una enorme cantidad de datos históricos, procesar datos sísmicos o simular el viento y el mapeo del terreno [3].

Figura 1.1: Servidores HPC[1]

La inteligencia artificial no se queda fuera de este campo, y menos aún ahora que es un tema candente, por lo que el *hardware* se está adaptando también a estos cambios mediante los ya mencionados aceleradores, que ayudan tanto con redes convolutivas (CNN)

[1] https://hpc.fau.de/systems-services/documentation-instructions/clusters/

como con los *transformers* [4] tan habituales hoy en día, especialmente en inteligencias artificiales generativas (*GenAI*) como ChatGPT[2] o el más reciente DeepSeek [5].

A continuación, se va a realizar un repaso por distintos dispositivos de *hardware* y su uso, para tener una visión más clara y completa sobre la importancia de las unidades de procesamiento neuronal.

1.1.2. CPU

Una unidad central de procesamiento (CPU por sus siglas en inglés) es el componente funcional central de un ordenador, que ejecuta el sistema operativo y las aplicaciones, entre otras muchas operaciones [6] [7].
Las CPU se pueden dividir en tres componentes principales:
- Unidad aritmético-lógica (ALU): encargada de realizar operaciones aritméticas y lógicas (sumas, restas, multiplicaciones, divisiones y comparaciones).
- Unidad de control: encargada de asignar las diferentes tareas a realizar.
- Unidad de memoria:encargada de gestionar las funciones relacionadas con el uso de la memoria.

De estos componentes, nos interesa especialmente la ALU, ya que en el estudio que se va a realizar, las operaciones aritmético-lógicas son clave.
Las CPU tienen más componentes importantes, tales como la memoria caché, a la que se accede con una velocidad mayor que a la RAM, o el reloj, que permite que los procesos de la CPU se sincronicen a la velocidad que éste determina [6] [8].

[2] https://chatgpt.com/

Figura 1.2: Aspecto de una CPU Intel Core i7 12700K por la parte superior[3]

Los algoritmos se pueden ejecutar enteramente en CPU, pero gracias a distintos aceleradores específicos, se puede lograr una mayor velocidad de lo que ésta consigue.

1.1.3. Aceleradores Matriciales

Los cálculos matriciales son algunas de las operaciones más utilizadas en el mundo de la computación. Presente en multitud de contextos, se utiliza en gran medida en aprendizaje automático e incluso como *benchmarking* para pruebas de computación de alto rendimiento.

Esto hace que no sea extraño que se hayan desarrollado diversos repertorios de instrucciones con el propósito de acelerar estas tareas, como puede ser *Intel Advanced Matrix Extensions (AMX)* de Intel, que está presente en el mercado desde el lanzamiento de los procesadores Xeon Sapphire Rapids [9], o *Scalable Matrix Extension (SME)* [10] para arquitecturas ARM.

[3] https://commons.wikimedia.org/wiki/File:Intel_CPU_Core_i7_12700K_Alder_Lake_top.jpg

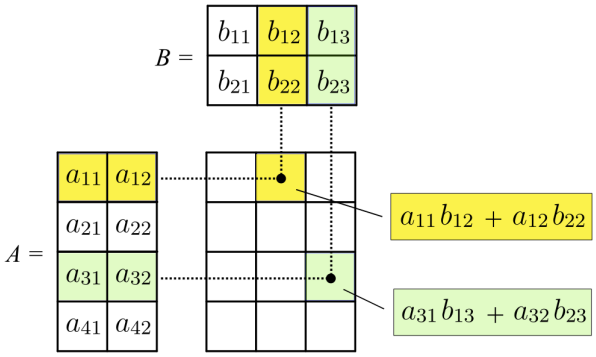


Figura 1.3: Ejemplo de una multiplicación de matrices[4]

Los chips Apple M4 son los primeros en soportar las instrucciones SME de ARM, algo muy positivo dado que los chips Apple Silicon también incluyen un acelerador específico para realizar operaciones matriciales desde su modelo M1, los llamados *Apple Matrix eXtension* (AMX) [11] [12] [13].

En la sección 4 se tratará más a fondo este dispositivo, dada la importancia de Apple en este trabajo.

1.1.4. GPU

Las unidades de procesamiento gráfico (GPU por sus siglas en inglés) son un circuito electrónico diseñado para acelerar el procesamiento de imágenes y gráficos informáticos. Aunque originalmente diseñadas con ese propósito, se utilizan en aplicaciones modernas como *Blockchain* o *Machine Learning*, especialmente desde que salieron herramientas públicas como CUDA de Nvidia, que permiten a los desarrolladores programar las GPUs según sus exigencias a un nivel muy bajo [14] [15].

[4] https://commons.wikimedia.org/wiki/File:MatrixMultiplication.png

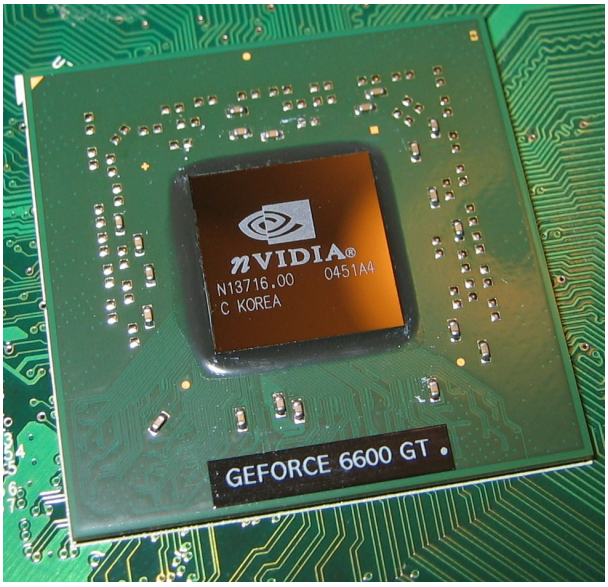


Figura 1.4: Aspecto de una GPU Nvidia 6600GT por la parte superior[5]

Tal y como se ha mencionado, el hecho de que las GPU sean más programables las ha vuelto más versátiles, lo que ha llevado a que se usen para otro tipo de tareas para las que originalmente no estaban diseñadas [15].

Nvidia permite dicha programación gracias al uso de CUDA, su toolkit para programar la GPU según las necesidades de cada programador[6], algo que se puede aplicar a modelos de machine learning gracias a su implementación en Python, lenguaje por excelencia para el desarrollo de modelos de aprendizaje automático[7].

De forma similar, las GPU de AMD han desarrollado ROCm para poder programar sus dispositivos[8], manteniéndose así en la vanguardia de la inteligencia artificial que estamos viviendo hoy en día.

[5] https://commons.wikimedia.org/wiki/File:6600GT_GPU.jpg
[6] https://developer.nvidia.com/cuda-toolkit
[7] https://developer.nvidia.com/cuda-python
[8] https://www.amd.com/es/products/software/rocm.html

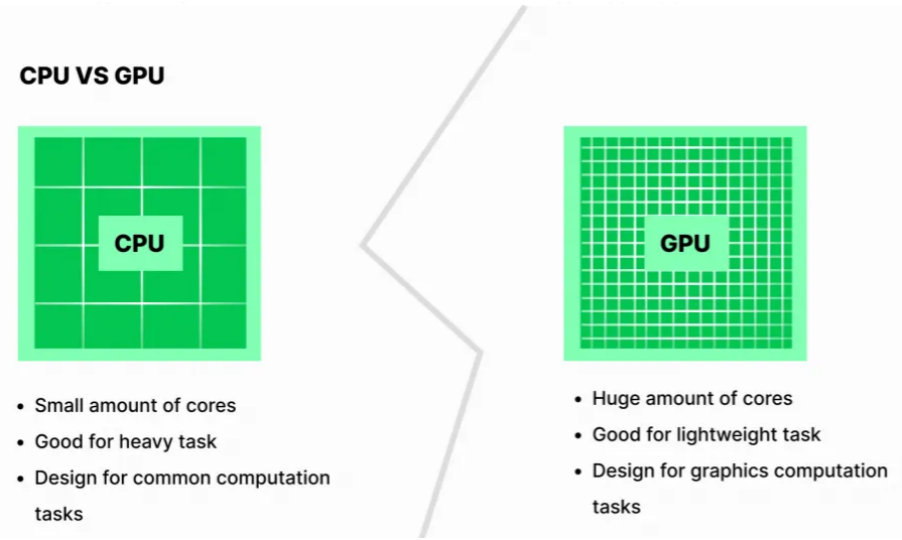


Figura 1.5: Comparativa CPU frente a GPU [16]

Estas herramientas permiten un entrenamiento notablemente más rápido en GPU que en CPU al tener un mayor número de núcleos y una mayor velocidad de reloj, además de mayor capacidad de memoria. Sumado a esto, la eficiencia energética obtenida también suele ser mejor en la GPU, lo que lleva a utilizar estos aparatos siempre que sea posible [16] [17].

1.1.5. Aceleradores Neuronales (NPU)

Las unidades de procesamiento neuronal (NPU por sus siglas en inglés) son aceleradores *hardware* pensados para que se realicen a mayor velocidad operaciones relacionadas con redes neuronales, aprendizaje profundo (*deep learning*) y *machine learning*.

Las NPU nacen de forma natural como respuesta al panorama actual y a antecedentes como las GPU, de las que se ha hablado en la sección 1.1.4. Dada su importancia actual, se han realizado estudios que determinen su efectividad en diferentes propósitos, destacando especialmente en multiplicaciones de vectores y matrices y algunas tareas de aprendizaje automático como clasificación de vídeo o modelos de lenguaje grandes. No obstante, otros componentes pueden ser mejores para otras tareas, como las GPU en temas de redes LSTM o las CPU en producto escalar [18].

Estos aceleradores son importantes ya que permiten una mejora significativa tanto en velocidad como en eficiencia para los algoritmos relacionados con el área de la inteligencia artificial, facilitando así el avance en la misma [19].

1.1.6. Tipos de Datos

Para entender este trabajo, hay que saber comprender qué es un tipo de dato y qué tipos de datos podemos utilizar.

En primer lugar, un tipo de datos es una clasificación que determina el conjunto de valores que una variable puede tomar y las operaciones que se pueden realizar sobre esos valores [20]. Los tipos de datos también establecen cómo se interpreta la información almacenada en memoria y cómo se procesan los datos en la unidad de cálculo.
Algunos tipos de datos comunes son:

- Enteros (*integers* en inglés): valores numéricos sin parte decimal, los hay con y sin signo. También los hay de diferentes longitudes (8, 16, 32, 64 bits, etc). Los enteros de 8 bits (INT8) son un tipo de datos usado frecuentemente para cuantizar modelos de IA.
- Punto flotante *floating point* en inglés: representan números reales. El estándar IEEE 754 establece que usualmente se distinguen en precisión simple (32 bits) y precisión doble (64 bits) [21], aunque en IA se utiliza con frecuencia la baja precisión (*half precision*), de 16 bits.
- Booleanos: valores binarios fundamentales en lógica. Representan verdadero o falso.
- Vectores y matrices: estructuras de datos que almacenan múltiples valores del mismo tipo. Se utilizan mucho en álgebra lineal y operaciones de aprendizaje automático, entre otros usos.

El correcto uso de los diferentes tipos de datos es clave para maximizar el rendimiento y minimizar el consumo energético de los programas. Es por ello por lo que las NPU y las GPU utilizan la ya mencionada baja precisión optimizada para aplicaciones de aprendizaje automático [22].
La compatibilidad del software también es importante en la elección de los tipos de datos, es por ello por lo que en este trabajo se han utilizado flotantes de 16 y 32 bits (FP16

y FP32) debido al soporte de la librería CoreML de Apple. Esto se tratará en mayor profundidad en el apartado 6.

1.2. Objetivos

Este trabajo tiene como objetivo comparar tanto la eficiencia en rendimiento como en consumo energético de los diferentes dispositivos disponibles en Apple Silicon: *Apple Neural Engine* (ANE), GPU y CPU en el uso de algoritmos de propósito general. Este terreno está prácticamente inexplorado, y existe escasa documentación de Apple con respecto al ANE.

En concreto, se dispone de un Mac Mini M1 y un Mac Mini M4 Pro.

Se han seguido los siguientes pasos:

- Probar modelos de inteligencia artificial (YOLOv3 y YOLOv11), para asegurar el correcto funcionamiento de la ANE en modelos de aprendizaje automático, así como analizar su tiempo de ejecución y consumo.
- Convertir modelos en PyTorch a CoreML y ejecutarlos, investigando las posibles opciones (tipo de datos de ejecución, cómo ejecutar en la ANE, etc.)
- Analizar el tiempo de ejecución y el consumo medio de los modelos YOLOv3 y YOLOv11.
- Crear diversos modelos de *machine learning* en formato Core ML con Python que realicen operaciones de propósito general. En concreto: multiplicación de matrices, Jacobi y Multigrid.
- Comparar el rendimiento y consumo energético tanto en el Mac Mini M1 como en el Mac Mini M4 Pro en todos los dispositivos disponibles para todos los algoritmos creados.
- Discutir los resultados obtenidos, y concluir sobre la eficacia de la ANE en los mismos y qué posibles pasos a seguir podrían darse a continuación.

Con estos pasos en mente, se discutirá la eficiencia de uso de la ANE en casos de uso general, así como la de otros dispositivos más comunes: la CPU y la GPU, discutiendo

cuál es el mejor en cada uno de los casos analizados.

1.3. Plan de trabajo

Para realizar este trabajo se han seguido numerosas tareas, que podríamos dividir en:

- Documentación sobre programación en la ANE: leer documentación sobre trabajos relacionados y tutoriales. Esta tarea se ha realizado a lo largo de todo el tiempo de desarrollo cuando se han encontrado problemas. No obstante, en la planificación se ha tenido en cuenta una etapa previa de documentación como preparación para comenzar el trabajo.
- Preparación del entorno de trabajo: esta tarea incluye la instalación de todas las librerías y entornos necesarios para trabajar, así como comprobaciones previas que demuestren su correcto funcionamiento.
- Implementación de los distintos modelos: adaptación de cada uno de los modelos para su funcionamiento en la ANE. Esto incluye su programación e investigación sobre cómo realizarla.
- Realización de pruebas: realización de las diferentes pruebas de evaluación de los modelos, que incluyen la toma de medidas a analizar en los resultados. Al realizar estas pruebas se analizan de forma preliminar los resultados de manera individual, pero posteriormente se dedicó tiempo a realizar un análisis exhaustivo de todos los resultados en su conjunto. Los modelos YOLO, por su parte, pasan directamente a esta etapa, ya que el modelo YOLOv3 viene ya adaptado oficialmente por Apple y los YOLOv11 tan solo deben ser transformados.
- Análisis de resultados: Esta etapa consiste en analizar los resultados obtenidos y sacar conclusiones claras sobre ellos.
- Escritura de la memoria: escritura de este documento. Algunas secciones se han ido redactando en paralelo de manera no definitiva, pero en la planificación se ha dejado margen de tiempo para dedicar tiempo exclusivo a esta tarea.

Para poder realizar todas estas tareas de forma coordinada y clara, se han realizado dos tareas clave:

La primera es la creación de un repositorio de GitHub donde se suban de manera constante los avances en el proyecto, con *commits* claros y una estructura bien organizada siguiendo un correcto flujo de trabajo Git. Este repositorio se creó en el momento de comienzo del proyecto y contiene un archivo *README* con algo de información sobre el proyecto.

La segunda es el desarrollo de un diagrama de Gantt que permita organizar la carga de trabajo a lo largo del tiempo, de forma que se manejen ciertos plazos para que el ritmo de trabajo sea constante y eficiente.

1.3.1. Diagrama de Gantt

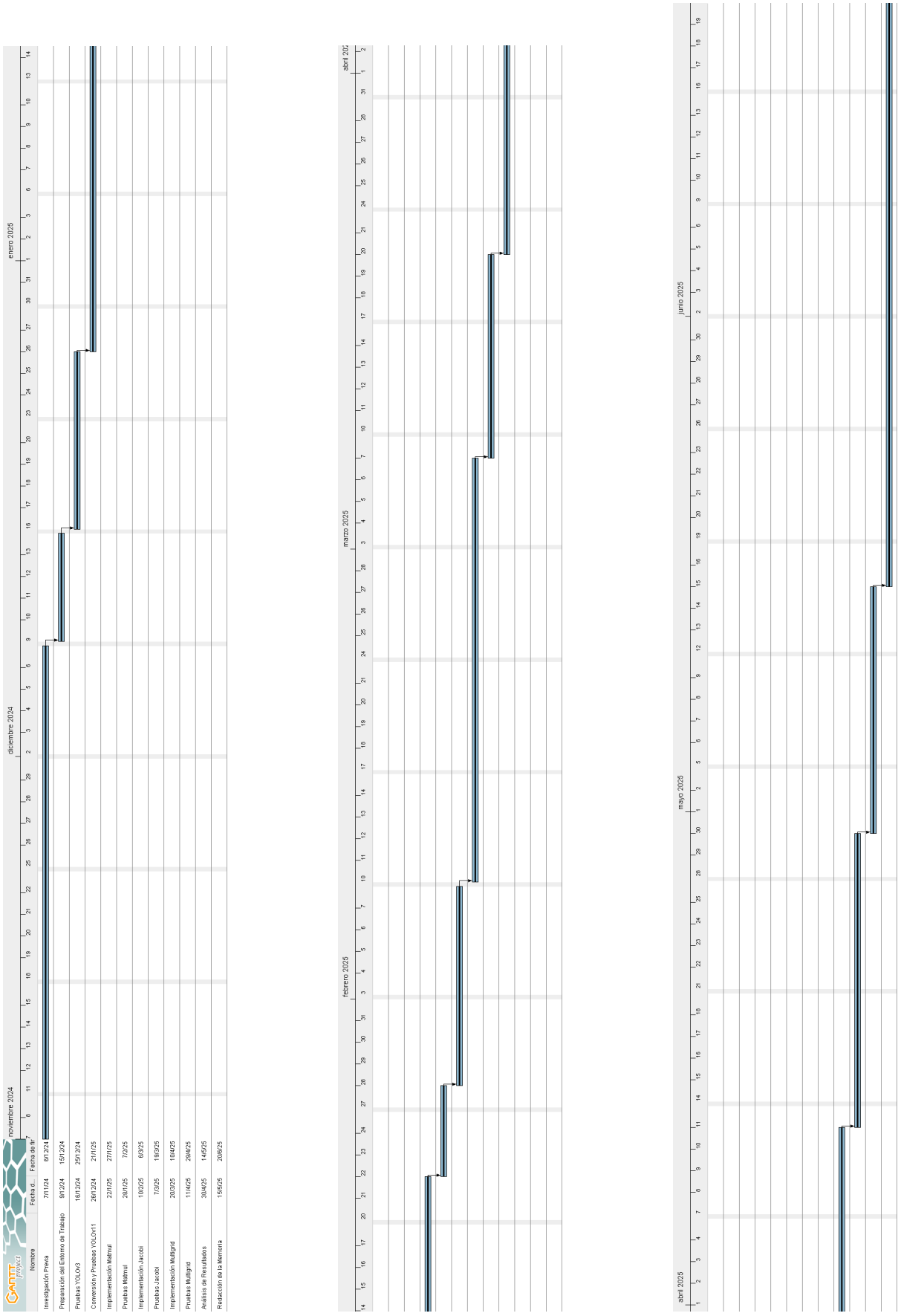
En esta sección encontramos el diagrama de Gantt utilizado para el desarrollo de este trabajo en la figura 1.6. El diagrama se ha realizado con GanttProject[9], un programa gratuito y de código abierto que permite la creación de este tipo de diagramas para gestionar proyectos.

Este diagrama muestra toda la planificación del proyecto, la cual se ha seguido durante todo el desarrollo del mismo.

Algunas tareas se han compatibilizado en algunos momentos, como la repetición de alguna prueba, pero por simplificación, dada la longitud del diagrama, este se ha mantenido con el contenido inicial del mismo.

Algunas tareas conllevan una longitud superior debido a que sus fechas coinciden con periodos como Navidad, Semana Santa o la época de exámenes del primer cuatrimestre.

[9] https://www.ganttproject.biz/



Parte 1 de 3 Parte 2 de 3 Parte 3 de 3

Figura 1.6: Diagrama de Gantt del Proyecto

1.3.2. Acceso al Repositorio de GitHub

El código utilizado para este proyecto se encuentra disponible en GitHub en el siguiente enlace: https://github.com/Corrochano/GPNPU

En este repositorio se encuentra una carpeta Apple, que se divide en las subcarpetas M1 y M4 Pro, cada una de ellas con las cinco carpetas correspondientes a los algoritmos utilizados: YOLOv3, YOLOv11, GEMM, Jacobi y Multigrid.

También hay una carpeta *Accelerate* con el código para utilizar este *framework*, que permite el uso del AMX.

Cada una de estas carpetas contiene los archivos Python y Shell utilizados para realizar este trabajo, permitiendo su total replicabilidad.

1.4. Descripción de los Capítulos

En esta sección se va a explicar brevemente en qué va a consistir cada uno de los capítulos que componen esta memoria.

La memoria consiste en siete capítulos, incluyendo el presente capítulo de introducción, donde se abordan algunos conocimientos previos necesarios para entender el trabajo, sobre sus objetivos y la metodología de trabajo. Los otros seis capítulos que lo componen son los siguientes:

- Motivación: en esta sección se repasará la evolución histórica de los microchips hasta llegar a los aceleradores neuronales y su importancia.
- Apple Silicon y Herramientas: esta sección tratará en profundidad qué es Apple Silicon, trabajos relacionados que aborden este tema y las herramientas de las que dispone.
- Algoritmos: aquí se explican los diferentes algoritmos utilizados en este trabajo.
- Metodología: este capítulo explica cómo se ha implementado el código del proyecto, así como todos los procesos que se han seguido en el mismo para su realización.
- Resultados: aquí se hallan los resultados obtenidos, así como el análisis de los mismos.
- Conclusiones: por último, este capítulo resume las conclusiones del proyecto así

como el trabajo futuro propuesto.

Capítulo 2

Introduction

2.1. Background

This section will explain some relevant background concepts to understand this work and the devices we will discuss.

2.1.1. High-Performance Computing

High-Performance Computing (HPC) is a topic of great current interest, especially in Neural Processing Units (NPUs) [1] [2].

HPC has always been one of the most cutting-edge scenarios for the future of computing, consisting of using high-performance computers to solve enormous problems in real-time, such as massive data processing, generally using servers (computers capable of handling client requests) [3].

To give a few examples, thanks to high-performance computing, we can sequence a human genome in less than a day, quickly detect fraud with risk analysis methods, predict the weather more accurately using a huge amount of historical data, process seismic data, or simulate wind and terrain mapping. [3]

Figura 2.1: HPC Servers[1]

Artificial intelligence is not left out of this field, and even less so now that it is a hot topic. Therefore, hardware is also adapting to these changes through the aforementioned accelerators, which help with both convolutional neural networks (CNNs) and the transformers [4] so common today, especially in generative artificial intelligences (GenAI) like

[1] https://hpc.fau.de/systems-services/documentation-instructions/clusters/

ChatGPT[2] or the more recent DeepSeek [5].

Next, we will review different hardware devices and their use, to get a clearer and more complete vision of the importance of neural processing units.

2.1.2. CPUs

A Central Processing Unit (CPU) is the central functional component of a computer, executing the operating system and applications, among many other operations [6] [7]. CPUs can be divided into three main components:
- Arithmetic Logic Unit (ALU): responsible for performing arithmetic and logical operations (additions, subtractions, multiplications, divisions, and comparisons).
- Control Unit: Assigns the different tasks to be performed.
- Memory Unit: Manages functions related to memory usage.

Of these components, we are particularly interested in the ALU, as arithmetic-logic operations are key in the study we are going to conduct.
CPUs have other important components, such as the cache memory, which is accessed faster than RAM, or the clock, which allows CPU processes to be synchronized at the speed it determines [6] [8].

[2] https://chatgpt.com/

Figura 2.2: Top view of an Intel Core i7 12700K CPU[3]

Algorithms can be executed entirely on the CPU, but thanks to different specific accelerators, a higher speed than what the CPU can achieve.

2.1.3. Matrix Accelerators

Matrix calculations are some of the most widely used operations in the world of computing. Present in a multitude of contexts, they are used extensively in machine learning and even as a benchmark for high-performance computing tests.

This makes it not surprising that various instruction sets have been developed for the purpose of accelerating these tasks, such as Intel Advanced Matrix Extensions (AMX), which have been on the market since the launch of the Xeon Sapphire Rapids processors [9], or Scalable Matrix Extension (SME) [10] for ARM architectures.

[3] https://commons.wikimedia.org/wiki/File:Intel_CPU_Core_i7_12700K_Alder_Lake_top.jpg

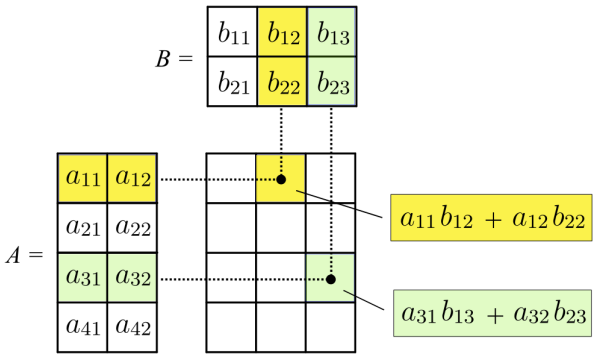


Figura 2.3: Example of a matrix multiplication[4]

Apple M4 chips are the first to support ARM's SME instructions, which is very positive given that Apple Silicon chips also include a specific accelerator for performing matrix operations since the introduction of their M1 model, the so-called *Apple Matrix eXtension* (AMX). [11] [12] [13].

Section 4 will discuss this device in more depth, given the importance of Apple in this work.

2.1.4. GPUs

Graphics Processing Units (GPUs) are electronic circuits designed to accelerate the processing of computer graphics and images.

Although originally designed for that purpose, they are used in modern applications such as Blockchain or Machine Learning, especially since public tools like Nvidia's CUDA were released, allowing developers to program GPUs to meet their specific demands at a very low level [14] [15].

[4] https://commons.wikimedia.org/wiki/File:MatrixMultiplication.png

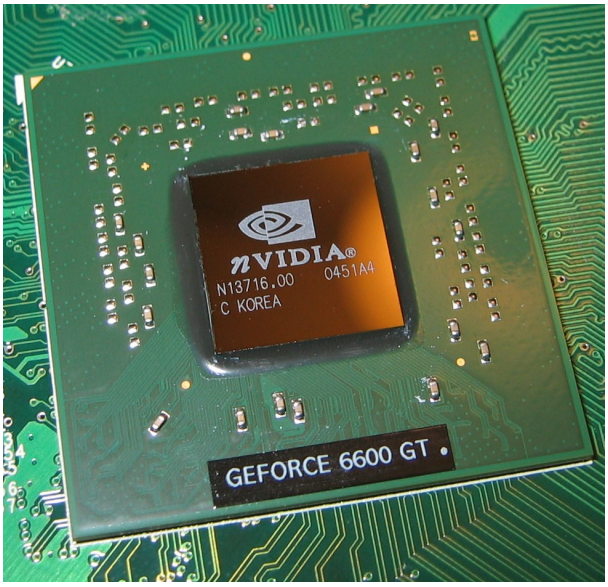


Figura 2.4: Top view of an Nvidia 6600GT GPU[5]

As mentioned, the fact that GPUs have become more programmable has made them more versatile, being used for other types of tasks for which they were not originally designed [15].

Nvidia enables such programming through the use of CUDA, its toolkit for programming the GPU according to each programmer's needs[6], which can be applied to machine learning models thanks to its implementation in Python, the quintessential language for developing machine learning models[7].

Similarly, AMD graphics cards have developed ROCm to program their devices[8], keeping pace with the artificial intelligence wave we are experiencing today.

[5] https://commons.wikimedia.org/wiki/File:6600GT_GPU.jpg
[6] https://developer.nvidia.com/cuda-toolkit
[7] https://developer.nvidia.com/cuda-python
[8] https://www.amd.com/es/products/software/rocm.html

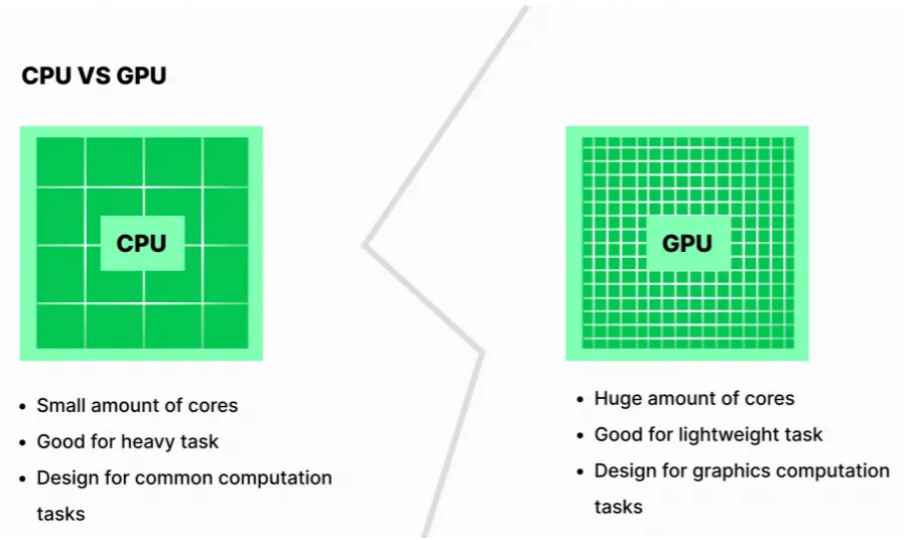


Figura 2.5: CPU vs GPU Comparison [16]

These tools allow for significantly faster training on GPUs than on CPUs due to a higher number of cores and a faster clock speed, in addition to greater memory capacity. Added to this, the energy efficiency obtained is also usually better on the GPU, which leads to using these devices whenever possible [16] [17].

2.1.5. Neural Accelerators (NPUs)

Neural Processing Units (NPUs) are hardware accelerators designed to speed up operations related to neural networks, deep learning, and machine learning.

These units arise naturally in response to the current landscape and precedents such as GPUs, which were discussed in section 2.1.4. Given their current importance, studies have been conducted to determine their effectiveness for different purposes, highlighting their performance especially in matrix-vector multiplication and some machine learning tasks such as video classification or large language models. However, other components may be better for other tasks, such as GPUs for LSTM networks or CPUs for dot products. [18]

These accelerators are important because they allow for a significant improvement in both speed and efficiency for algorithms related to the field of artificial intelligence, thus facilitating its advancement. [19]

2.1.6. Data Types

To understand this work, it is necessary to understand what a data type is and what data types we can use.

First, a data type is a classification that determines the set of values a variable can hold and the operations that can be performed on those values. [20] Data types also establish how information stored in memory is interpreted and how data is processed in the calculation unit.
Some common data types are:

- Integers: numerical values without a decimal part, available as signed and unsigned. They also come in different lengths (8, 16, 32, 64 bits, etc). 8-bit integers (INT8) are a data type frequently used for quantizing AI models.
- Floating point: represent real numbers. The IEEE 754 standard specifies that they are usually distinguished by single precision (32 bits) and double precision (64 bits), [21] although in AI, low precision (*half precision*), 16 bits, is widely used.
- Booleans: fundamental binary values in logic. They represent true or false.
- Vectors and tensors: data structures that store multiple values of the same type. They are widely used in linear algebra and machine learning operations, among other uses.

The correct use of different data types is key to maximizing performance and minimizing the energy consumption of programs. This is why NPUs and GPUs use the aforementioned low precision optimized for machine learning applications [22].
Software compatibility is also important in the choice of data types, that is why 16-bit and 32-bit floats (FP16 and FP32) have been used in this work due to the support of Apple's CoreML library. This will be discussed in more depth in the 6 section.

2.2. Objectives

This paper aims to compare both the performance and power efficiency of the different devices available on Apple Silicon: the Apple Neural Engine (ANE), GPU, and CPU, using general-purpose algorithms. This area is virtually unexplored, and there is almost no documentation from Apple regarding ANE.
Specifically, we have a Mac Mini M1 and a Mac Mini M4 Pro.
The following steps have been followed:

- Test artificial intelligence models (YOLOv3 and YOLOv11), to ensure the correct functioning of the ANE in machine learning models, as well as analyze its execution time and consumption.
- Convert models from PyTorch to CoreML and run them, investigating the possible options (execution data type, how to run on the ANE, etc.)
- Analyze the execution time and average consumption of the YOLOv3 and YOLOv11 models.
- Create various machine learning models in Core ML format with Python that perform general-purpose operations. Specifically: matrix multiplication, Jacobi, and multigrid.
- Compare the performance and energy consumption on both the Mac Mini M1 and the Mac Mini M4 Pro across all available devices for all created algorithms.
- Discuss the results obtained, concluding on the effectiveness of ANE in these results and what possible next steps could be taken.

With these steps in mind, the efficiency of using the ANE in general-purpose use cases will be discussed, as well as that of other more common devices: the CPU and the GPU, discussing which is the best for each of the analyzed cases.

2.3. Work Plan

To carry out this work, numerous tasks have been followed, which could be divided into:

- Documentation on ANE programming: Reading documentation on related works and tutorials. This task has been carried out throughout the development time whenever problems were encountered. However, in the planning, a prior documentation stage was taken into account as preparation to start the work.
- Preparation of the work environment: this task includes the installation of all necessary libraries and environments to work, as well as preliminary checks to ensure their correct functioning.
- Implementation of the different models: adaptation of each of the models to work on the ANE. This includes their programming and research on how to do it.
- Tests: Carrying out the different evaluation tests of the models, which include taking measurements to be analyzed in the results. When carrying out these tests, the results are lightly analyzed individually, but later time was dedicated to conducting an exhaustive analysis of all the results together. The YOLO models, for their part, go directly to this stage, since the YOLOv3 model is already officially adapted by Apple and the YOLOv11 models only need to be converted.
- Analysis of results: This stage consists of analyzing the obtained results and drawing clear conclusions from them.
- Thesis writing: Writing this document. Some sections have been drafted in parallel in a non-definitive way, but in the planning, a time margin has been left to dedicate exclusive time to this task.

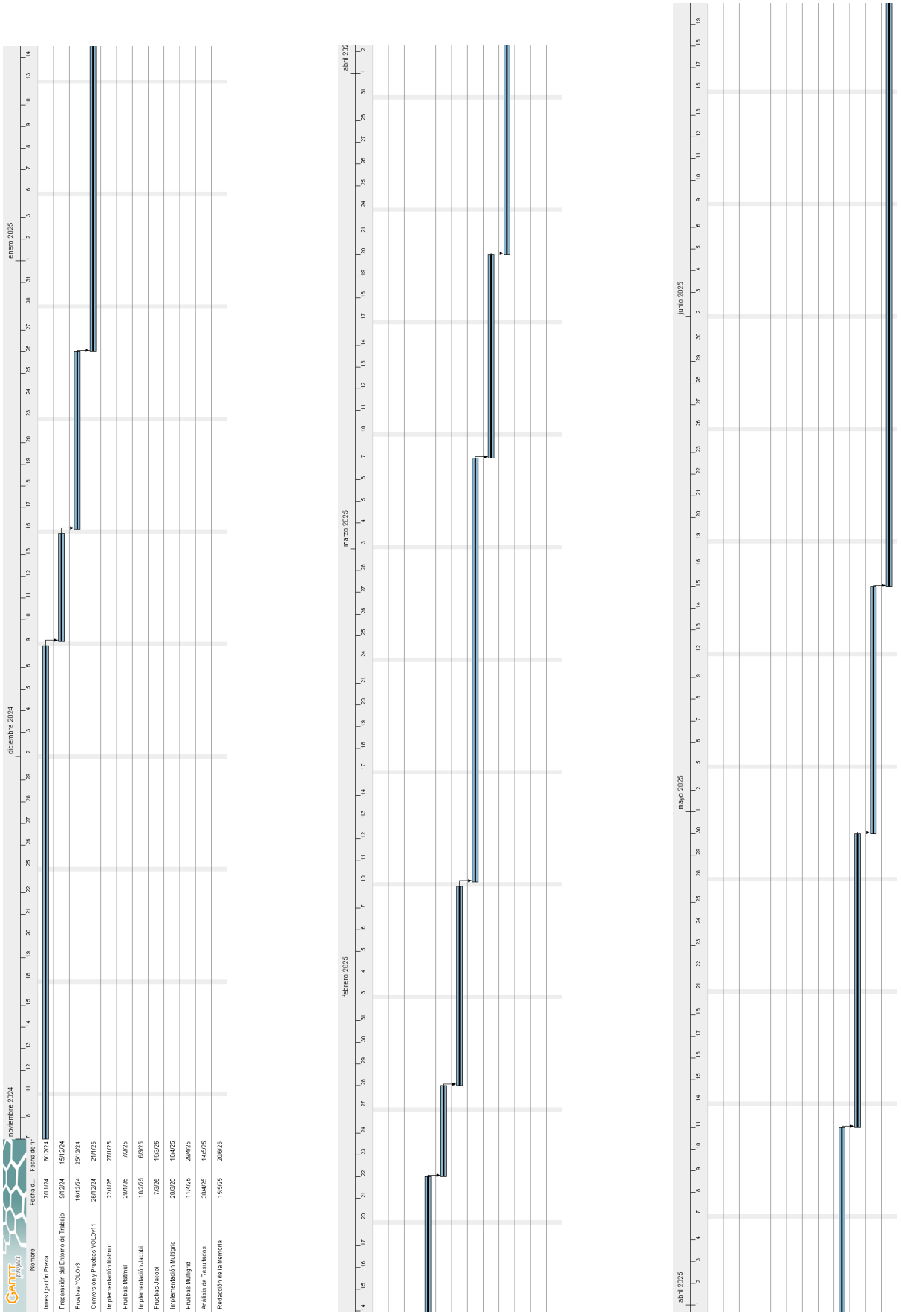
To be able to carry out all these tasks in a coordinated and clear manner, two key tasks have been performed:
The first is the creation of a GitHub repository where the project's progress is constantly uploaded, with clear commits and a well-organized structure following a correct GitFlow. This repository was created at the beginning of the project and contains a *README* file with some information about the project.
The second is the development of a Gantt chart that allows organizing the workload over time, so that certain deadlines are managed to ensure a constant and efficient work pace.

2.3.1. Gantt Chart

In this section, we find the Gantt chart used for the development of this work in figure 2.6. The diagram was created with GanttProject[9], a free and open-source program that allows the creation of this type of diagram to manage projects.

This diagram shows the entire project planning, which has been followed throughout its development.
Some tasks have been combined at times, such as repeating a test, but for simplification, given the length of the diagram, it has been kept with its initial content.
Some tasks have a longer duration because their dates coincide with holidays such as Christmas, Easter, or the exam period of the first semester.

[9] https://www.ganttproject.biz/



Part 1 of 3 Part 2 of 3 Part 3 of 3

Figura 2.6: Gantt Chart of the Project

2.3.2. Access to the GitHub Repository

The code used for this project is available on GitHub at the following link: https://github.com/Corrochano/GPNPU

In this repository, there is an Apple folder, which is divided into M1 and M4Pro subfolders, each of them with the five folders corresponding to the algorithms used: YOLOv3, YOLOv11, matmul, jacobi, and multigrid.
There's also an Accelerate folder with the code to use this framework, which allows the use of the AMX.
Each of these folders contains the files used to carry out this work, allowing for its full replicability.

2.4. Chapter Descriptions

This section will briefly explain what each of the chapters that make up this thesis will consist of.
The thesis consists of seven chapters including this introduction, which covers some of the prior knowledge needed to understand the work, its objectives, and its working methodology. The other six chapters that compose it are the following:

- Motivation: This section will review the historical evolution of microchips up to neural accelerators and their importance.
- Apple Silicon and Tools: This section will discuss in depth with what Apple Silicon is, related works that addressing it, and the tools available.
- Algorithms: Here, the different algorithms used in this work are explained.
- Methodology: This chapter explains how the project's code has been implemented, as well as all the processes that have been followed in it for its execution.
- Results: Here are the results obtained, as well as their analysis.
- Conclusions: Finally, this chapter summarizes the different conclusions of the project as well as the proposed future work.

Capítulo 3

Motivación

3.1. El Primer Microprocesador

Lo que podríamos considerar como la unidad central de procesamiento (CPU por sus siglas en inglés) es el microchip Intel 4004, lanzado en 1971, desarrollado por Marcian "Ted" Hoff, Federico Faggin y Masatoshi Shima, entre otros [23].

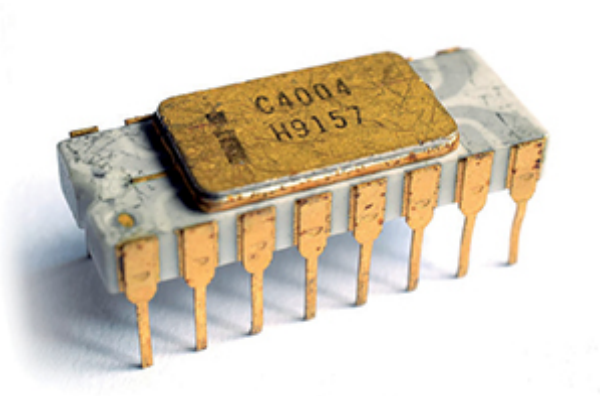


Figura 3.1: Intel 4004[1]

Hasta ese momento, los computadores se basaban en diversos chips que, en conjunto, lograban hacer funcionar el ordenador, pero las mejoras en escala hicieron posible integrar diversos elementos de un circuito computacional en un solo microchip, permitiendo así a Federico Faggin liderar el desarrollo del ya mencionado Intel 4004, aunque otras empresas como SRI, IBM o RCA tenían ideas para proyectos similares. A pesar del gran avance que suponían las CPU, no fue hasta mediados de los años 70 cuando la popularidad de las CPU se elevó de forma consistente, lo que se debió a la bajada de precio de los transistores [23].

3.2. Ley de Moore y Escalado de Dennard

Las CPU, desde su creación, han ido alcanzando hitos importantes a lo largo de la historia, en gran parte debido a la Ley de Moore y al Escalado de Dennard.

La Ley de Moore establece que cada dos años el número de transistores que caben en un circuito integrado se duplica. Por otro lado, el Escalado de Dennard (Dennard Scaling

[1] https://www.pmi.org/learning/library/es-intel-4004-12417

en inglés) dice que al reducirse el tamaño de los transistores, el rendimiento aumenta manteniendo el consumo bajo control, apoyando así la ley de Moore.

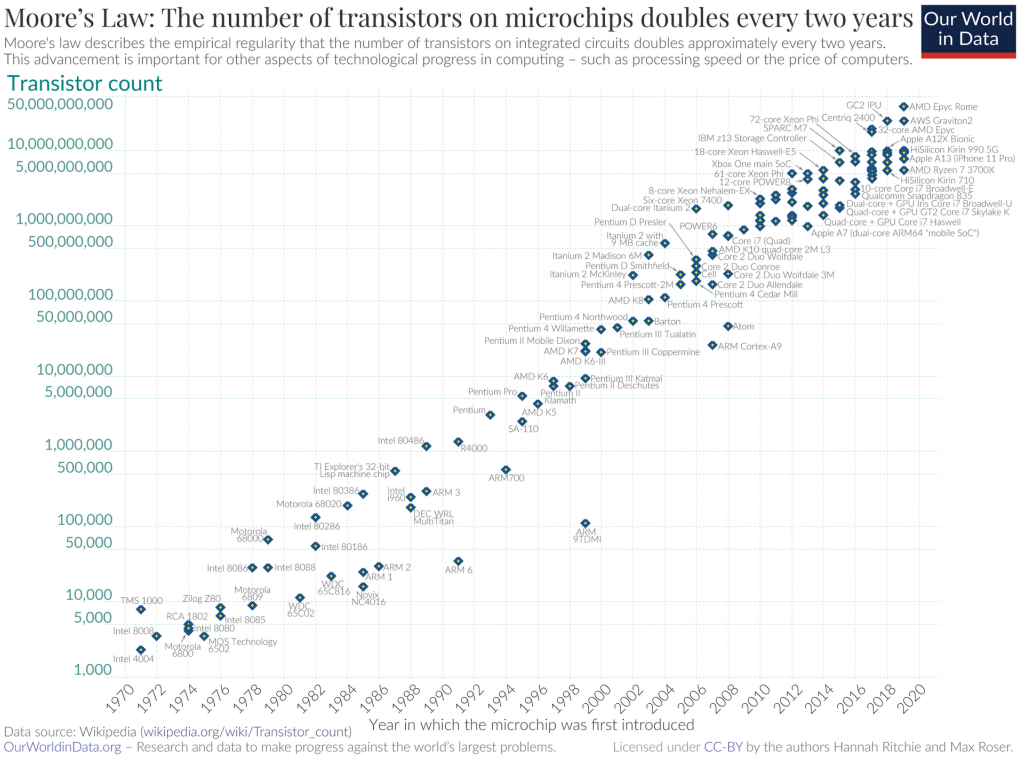


Figura 3.2: Gráfico de la Ley de Moore entre 1970 y 2020[2]

Desde el inicio de los años 2000, la Ley de Moore se ha ralentizado, lo que parece indicar que se acerca el final de la misma [24] [25]. A su vez, el Escalado de Dennard [26] parece también haber llegado a su final, lo que plantea un cambio de paradigma con el fin de seguir mejorando la potencia de cómputo [8].

3.3. Evolución de las GPU

Para poder realizar mejoras sustanciales en distintas tareas, surgieron diferentes aceleradores específicos entre los que se encuentran las unidades de procesamiento gráfico (GPU por sus siglas en inglés).

Estos aceleradores están presentes hoy en día en casi la totalidad de los computadores, con tareas tan diversas como acelerar cálculos de datos de vídeo o inferencias de modelos de aprendizaje automático. Pese a todos los usos actuales, las GPU comenzaron a existir

[2] https://sima.ai/blog/software-is-powering-the-ai-revolution/

como unidades especializadas en renderizado de gráficos tridimensionales, estando altamente optimizadas para operaciones vectoriales y matriciales.

La primera GPU que incluía todo el *pipeline* necesario de funcionamiento fue la Nvidia GeForce 256 en 1999. Este dispositivo trajo de la mano avances más allá de su mayor capacidad para gráficos tridimensionales, como el puerto para gráficos acelerados (AGP por sus siglas en inglés), lo que permitió que la industria del videojuego comenzase su verdadero despegue doméstico en aquella época [27] [28].

La enorme potencia de cómputo de las GPU las llevó a su utilización en otros propósitos para los que originalmente no estaban diseñadas, naciendo así la denominada computación de propósito general en GPU (GPGPU por sus siglas en inglés) [29].
Este paradigma se llevó a cabo gracias a diferentes *frameworks* especializados que permiten la programación manual de las GPU, comenzando con el lanzamiento de CUDA para Nvidia,[3] que en 2006 permitió programar código paralelo directamente en las GPU marcando un antes y un después en la historia de las mismas, siendo seguida por OpenCL, ampliando compatibilidad con otros fabricantes como Intel o AMD[4] o Metal para Apple[5].

Esta versatilidad en las GPU llega hasta nuestros días, manteniéndose las GPU al día en el desarrollo de modelos de lenguaje de inteligencia artificial, con soporte nativo en las principales librerías como TensorFlow[6] o PyTorch[7] gracias a la gran capacidad de ejecutar operaciones de punto flotante a gran escala. Esto ha llevado a la posibilidad de entrenar modelos que con CPU no serían viables, tal y como se demostró al desarrollarse el modelo AlexNet [30].

El avance de las GPU para su uso en aprendizaje automático ha llegado mucho más lejos, con arquitecturas específicas como los llamados *Tensor Cores* de Nvidia o sus homólogos de AMD llamados *Matrix Cores*. Estos núcleos están optimizados para operaciones de baja precisión, como FP16 o INT8 y son muy utilizados en el despliegue de modelos de

[3] https://docs.nvidia.com/cuda/cuda-c-programming-guide/
[4] https://www.khronos.org/opencl/
[5] https://developer.apple.com/metal/
[6] https://www.tensorflow.org/
[7] https://pytorch.org/

aprendizaje profundo[8].

Esta evolución constante en la inteligencia artificial ha motivado a buscar nuevas formas de aceleración, desembocando en la creación de aceleradores neuronales, cuyo propósito es la aceleración específica de modelos de inteligencia artificial.

3.4. La Llegada de los Aceleradores Neuronales

El avance de los aceleradores neuronales es un tema popular dado el gran impacto de la inteligencia artificial, especialmente con el lanzamiento de los *transformers* que han llevado a modelos como ChatGPT[9] o el reciente DeepSeek [5] [4].

En los últimos años, marcas líderes del mercado de *hardware* como AMD[10], Intel[11], Apple[12] o Samsung[13] están apostando por el desarrollo de aceleradores neuronales, lo que ha llevado a probar la eficiencia, tanto temporal como energética, de este tipo de componentes.

Por supuesto, este desarrollo está respaldado por mediciones que apuntan a un mejor desempeño de estos aceleradores en tareas específicas de la inteligencia artificial, lo que justifica y motiva su estudio [18].

Dado que las NPU están actualmente en el punto de mira, no es de extrañar que se hayan tratado de utilizar en terrenos más generales, tal y como ocurrió con las GPU. Un ejemplo es la transformación de Parrot, que consiste en seleccionar manualmente regiones de código imperativo y entrenar una red neuronal que genere el resultado, sustituyendo el código clásico. Esto permite utilizar una NPU para obtener una ganancia en rendimiento y/o coste energético, demostrando que, tal y como ocurrió con las GPU, un uso general de las NPU o al menos selectivo en algunas regiones es posible [31].

La motivación dada por esto nos impulsa a investigar sobre el acelerador neuronal de los chips Apple Silicon: el *Apple Neural Engine* (ANE).

[8] https://www.nvidia.com/en-us/data-center/h100/

[9] https://chatgpt.com/

[10] https://www.amd.com/es/products/processors/consumer/ryzen-ai.html

[11] https://www.intel.la/content/www/xl/es/products/docs/accelerator-engines/ai-engines.html

[12] https://machinelearning.apple.com/research/neural-engine-transformers

[13] https://semiconductor.samsung.com/support/tools-resources/dictionary/the-neural-processing-unit-npu-a-brainy-next-generation-semiconductor/

3.5. Apple Silicon

Apple Silicon son los sistemas en un chip (*system on a chip*, SoC en inglés) diseñados por la compañía Apple. Se empezaron a lanzar en 2020, cuando Apple dejó de incluir chips de Intel en sus ordenadores.[14]

Dada la relevancia de Apple Silicon en este trabajo, la evolución y detalles más precisos de este dispositivo se comentarán en la sección 4.1.

[14] https://support.apple.com/es-es/116943

Capítulo 4

Apple Silicon y Herramientas

4.1. Apple Silicon

Apple Silicon es la gama de sistemas en un chip (*System On Chip*, SoC) de Apple. Estos sistemas tienen la particularidad de integrar todos (o casi todos) los componentes necesarios de un computador en un solo chip. Estos sistemas han demostrado ser muy útiles en el contexto de HPC [32], con resultados que demuestran mejoras tanto en eficiencia como en consumo energético [13] [33] [12].
Algunos de los factores más importantes para esto son el hecho de tener una estructura de memoria unificada, que ahorra tiempo de transferencia de datos entre dispositivos, o la estrecha integración de componentes, que ayuda a ganar eficiencia energética.

La última y actual gama de chips de Apple es la primera diseñada para ser utilizada en ordenadores Mac y no sólo en dispositivos móviles. Esta es la gama Apple Silicon Serie-M, compuesta por: M1, M2, M3 y M4. Estos chips incluyen versiones básicas más baratas y versiones Pro, Ultra y Max, cada una más cara que la anterior, incluyendo mejoras en el chip en comparación con las otras versiones. Podemos ver la evolución de los modelos principales de los chips en la figura 4.1.

Feature	M1	M2	M3	M4
Process Technology (nm)	5	5/4	3	3
CPU Architecture	ARMv8.5-A	ARMv8.6-A	ARMv8.6-A	ARMv9.2-A
Perf./Eff. Cores	4/4	4/4	4/4	6/4
Clock Frequency (GHz)	3.2 / 2.06	3.5 / 2.42	4.05 / 2.75	4.4 / 2.85
Vector Unit (name/size)	NEON/128	NEON/128	NEON/128	NEON/128
L1 Cache (KB)	128/64	192/128	192/128	192/128
L2 Cache (MB)	12/4	16/4	16/4	16/4
AMX Characteristics	FP16,32,64	FP16,32,64/BF16	FP16,32,64/BF16	FP16,32,64/BF16
GPU Cores	7–8	8–10	10	10
GPU Clock (GHz)	1.27	1.39	1.38	1.47
FP32 TFLOPS	2.3–2.6	2.9–3.6	3.6	4.26
Neural Engine (cores)	16	16	16	16
Memory Technology	LPDDR4X	LPDDR5	LPDDR5	LPDDR5X
Max Unified Memory (GB)	8–16	8–24	24	32
Memory Bandwidth (GB/s)	67	100	100	120

Cuadro 4.1: Comparación de los distintos chips Apple Silicon Serie-M[1] [13]

En la figura 4.1 podemos ver la distribución de los distintos tipos de componentes que forman el SoC. Destaca que la mayor parte de la placa la ocupa la GPU con un amplio margen. Además, se puede observar un enfoque big.LITTLE en el procesador, con núcleos de alto rendimiento y de eficiencia por separado. La ANE (etiquetada como NPU) tiene un

[1] https://support.apple.com/es-es

tamaño pequeño, al igual que el coprocesador AMX, que ayuda a calcular multiplicaciones de matrices y es aún más pequeño.

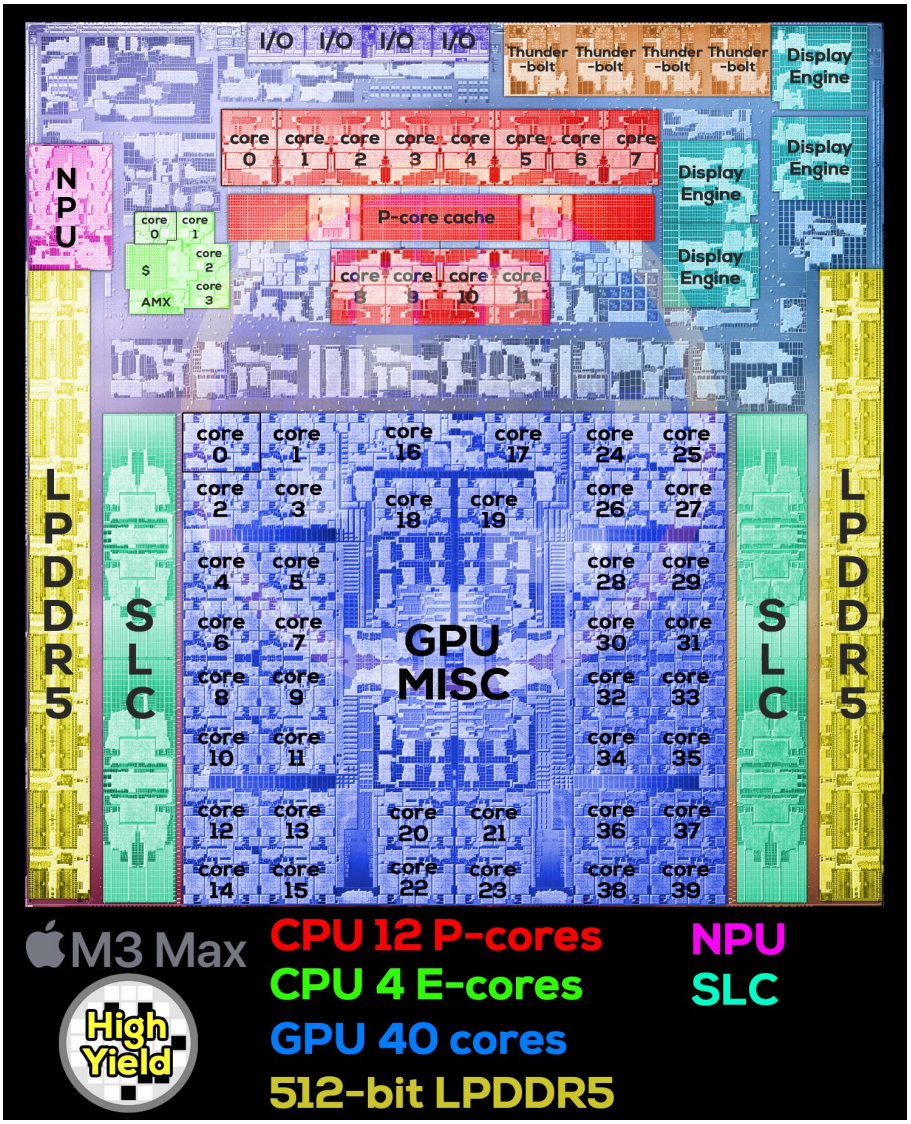


Figura 4.1: Chip M3 Pro[2]

La serie M de Apple es un gran ejemplo de arquitecturas SoC que integran multitud de componentes. Los chips, basados en arquitectura ARM, integran una CPU con núcleos tanto eficientes como de rendimiento, coprocesador para operaciones matriciales AMX, GPU integrada y un acelerador neuronal.

El procesador soporta un rango muy amplio de precisiones numéricas, incluyendo precisión doble (FP64), precisión simple (FP32) y otros formatos de menor tamaño como semiprecisión (FP16). La GPU, por su parte, soporta FP64, pero está optimizada para

[2] https://x.com/highyieldYT/status/1719306863341113349

trabajar con precisiones menores como FP32, FP16 o INT8. Por último, la ANE soporta datos de tipo FP16 únicamente[3], limitando su uso respecto a los demás dispositivos del chip [13].

El uso de los Mac series M en HPC es un terreno novedoso e interesante, dadas las especificaciones del chip de Apple. En especial, la inexplorada ANE es el terreno perfecto para realizar investigaciones.

4.1.1. CPU de Apple Silicon

La CPU de Apple Silicon está basada en ARM. Estos procesadores utilizan una aproximación big.LITTLE[13], que incorpora núcleos de dos tipos: alto rendimiento, para ganar el mayor rendimiento posible, y eficientes, para mejorar la eficiencia energética. El enfoque distribuye dinámicamente la carga de trabajo entre ambos tipos de núcleos.

Estos procesadores incluyen también capacidad para procesamiento de vectores por medio del repertorio de instrucciones NEON 128-bit y el coprocesador *Apple Matrix eXtension* (AMX), que se utiliza junto a la CPU para acelerar cálculos matriciales[4] [13]. Este coprocesador permite realizar múltiples operaciones matriciales de forma paralela siempre y cuando sus dimensiones estén fijadas independientemente de su tamaño en bytes.

El avance más significativo en el AMX viene dado en el desarrollo del chip más reciente, el M4. En este chip, la AMX hace uso de ARM SME (*Scalable Matrix Extension*) [13][12], una ampliación de ARM diseñada para acelerar los cálculos matriciales, dejando atrás la estructura propia de Apple en modelos anteriores.

Este modelo también incorpora dos clústeres de núcleos de alto rendimiento, lo que implica dos AMX que pueden trabajar con dos hilos paralelos[5].

[3] https://apple.github.io/coremltools/docs-guides/source/typed-execution.html
[4] https://research.meekolab.com/the-elusive-apple-matrix-coprocessor-amx
[5] https://eclecticlight.co/2024/11/11/inside-m4-chips-p-cores/

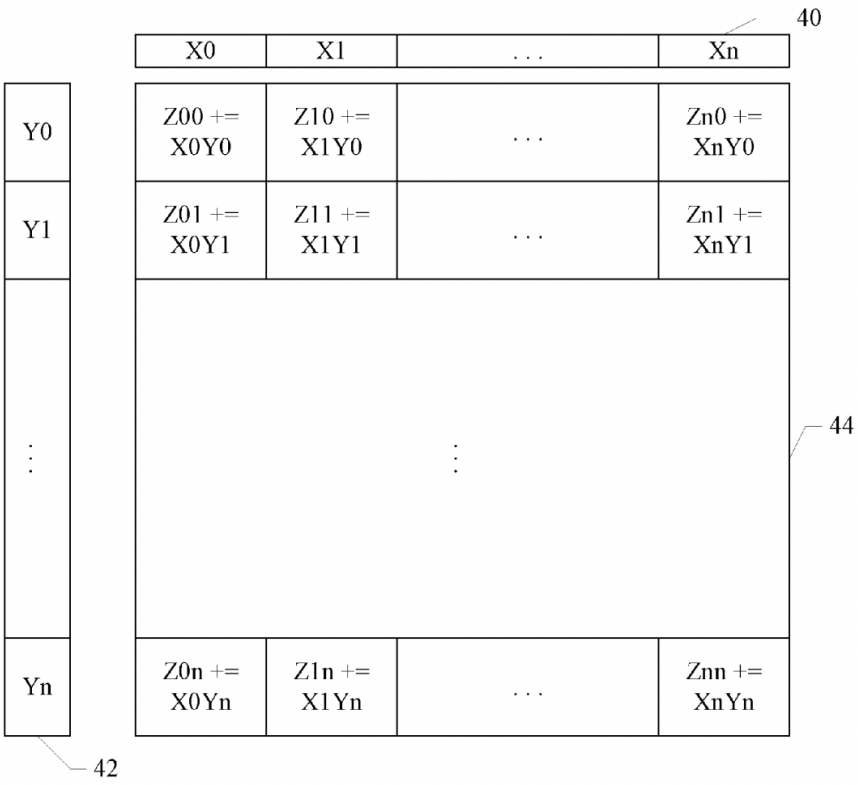


Figura 4.2: Imagen Explicativa Sobre el Funcionamiento del AMX[6]

La figura 4.2 muestra una tabla que podemos interpretar como una tabla de multiplicar, donde en cada posición sumamos al valor existente el resultado de multiplicar la fila X y la columna Y correspondientes. Por ejemplo, la posición Z0 de la tabla sumará el valor de X0*Y0. Este cálculo se realiza a la vez en toda la tabla en lugar de ir valor a valor, acelerando enormemente la ejecución.

El AMX permite también controlar su malla de 32x32 unidades de cómputo agrupándolas de diferentes maneras según la precisión deseada:

- Para precisión de 16-bit, cada una de las 1024 unidades realiza operaciones de forma independiente, tal y como se ha explicado anteriormente.
- Para precisión de 32-bit, las unidades se agrupan en submallas de 2x2, actuando cada bloque 2x2 como una sola unidad.
- Para precisión de 64-bit, se sigue una lógica similar a la de 32-bit, agrupando esta vez en 4x4 unidades.

[6] https://zhen8838.github.io/2024/04/23/mac-amx_en/9

Programación

La programación específica de la CPU se rige por las herramientas Clang/LLVM, que ofrecen soporte para ARM y que permiten la escritura de código en lenguajes como C, Objective-C, C++ y Swift. Estos compiladores optimizan el uso de los núcleos de alto rendimiento y eficiencia de la CPU. Existe también soporte para intrínsecas de ARM para escribir operaciones SIMD (*Single Instruction Multiple Data*) explícitamente [13].

Por otro lado, la AMX se puede utilizar por medio del *framework* Accelerate[7] de Apple, que aprovecha las capacidades del coprocesador para ganar rendimiento en aplicaciones como procesamiento de señales o álgebra lineal, aunque a un nivel de programación más alto. Existen tutoriales realizados por usuarios para usar de forma eficiente este *framework* para explotar las capacidades del AMX[8]. A día de hoy, no existe documentación oficial para programar la AMX a bajo nivel.
Por último, con un nivel de abstracción aún más alto, se puede utilizar el *framework* CoreML, que gestiona todo automáticamente por nosotros [13].

4.1.2. GPU de Apple Silicon

La GPU de los chips Serie-M emplea un renderizado diferido (*tile-based deferred rendering*, TBDR), consistente en dividir la escena a visualizar en fragmentos para renderizar los cálculos fragmento a fragmento. El primer paso es dividir la pantalla en fragmentos, posponiendo los cálculos de sombreado hasta determinar la visibilidad de los píxeles para optimizar el consumo energético al minimizar el número de píxeles ocultos renderizados. La GPU, además, incluye memoria caché en el chip, integrando memoria compartida y por núcleo [13].

Programación

La interfaz de programación principal de estas GPU es Metal API[9], un *framework* de bajo nivel que permite un control a grano muy fino de los recursos y la ejecución de la GPU. Metal soporta desde tareas de propósito general como la multiplicación de matrices hasta

[7] https://developer.apple.com/documentation/accelerate
[8] https://zhen8838.github.io/2024/04/23/mac-amx_en/
[9] https://developer.apple.com/metal/

tareas de aprendizaje automático. El código se puede programar mediante el lenguaje *Metal Shading Language* (MSL)[10], similar a CUDA o C++ [13]. El uso del lenguaje Julia también es posible para programar la GPU[11].

Existen también *frameworks* de alto nivel como *Metal Performance Shaders* (MPS) o el ya mencionado CoreML, que permiten la ejecución en la GPU con tan solo especificarlo, sin ofrecer control al usuario sobre la ejecución.

4.1.3. Acelerador Neuronal de Apple Silicon

El acelerador neuronal de estos chips, llamado *Apple Neural Engine* (ANE) es un acelerador *hardware* optimizado para aprendizaje automático introducido en el chip M1. Este dispositivo es compatible con el formato de datos FP16 según la documentación oficial de la librería CoreML[12] y opera como un dispositivo independiente de manera similar a la CPU y la GPU, a diferencia del AMX, que actúa como un coprocesador [13].

Programación

Es posible utilizar la ANE por medio de CoreML, que gestiona automáticamente todo sin opción de control al usuario. Además, CoreML no garantiza el uso de la ANE, pudiendo utilizar la CPU en su lugar si alguna operación no es compatible. No existe documentación a día de hoy sobre estas compatibilidades ni sobre las instrucciones internas del acelerador, por lo que una investigación mediante ingeniería inversa es interesante en este campo [13]. Existen manuales y guías no oficiales sobre cómo crear redes compatibles con la ANE y el uso de la misma [34]. Además, la documentación de CoreML indica cómo ejecutar en la misma. De esto se hablará en mayor profundidad en el capítulo 6.

4.1.4. Trabajos Relacionados con Apple Silicon

La principal referencia para este trabajo ha sido el artículo de Hübner et al.[13], que investiga el potencial y las características de los distintos chips de Apple Silicon y documenta en un trabajo exhaustivo el uso en HPC con datos de tipo FP32.

El estudio utiliza C++ para programar la CPU y Metal para la GPU, utilizando además

[10] https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf
[11] https://github.com/JuliaGPU/Metal.jl
[12] https://apple.github.io/coremltools/docs-guides/source/typed-execution.html

el *framework* Accelerate[13] en la CPU para programar el dispositivo AMX. Para medir la energía, se hace uso de la herramienta Powermetrics[14], una instrucción de consola presente en MacOS.

El trabajo señala la dificultad para programar en la ANE, poco documentada por parte de Apple, aunque con algunos manuales como el escrito por Matthijs Hollemans [34], con consejos e información sacada mediante ingeniería inversa sobre operaciones aceptadas en la ANE.
Finalmente, el trabajo de Hübner et al. [13] concluye que los chips han experimentado un gran desarrollo a lo largo de su seguimiento, pasando la GPU de 1.36 FP32 TFLOPS en el M1 a 2.9 FP32 TFLOPS en el M4. Se destaca también cómo la GPU supera a la CPU con una gran ventaja, especialmente a partir del M2, demostrando una importancia especialmente relevante en estos dispositivos.
Sobre consumo energético, se observa un bajo consumo de los chips que podría ser muy relevante en el ámbito de HPC.

El trabajo de Kenyon et al. [35] es otra investigación relevante para estos chips, ya que compara las GPUs de los modelos M1 y M1 Ultra con las de otros fabricantes (Nvidia y AMD), concluyendo que las GPUs de los chips de Apple pueden superar a las de las otras marcas en rendimiento y coste por GFLOP en operaciones como GEMM.
Resultados similares se han encontrado en otros trabajos, como el estudio de Struniawski et al. [36] que compara los chips M1 y M2 con una GPU Nvidia 3090 y un portátil de gama media, demostrando que el chip M2 tiene mejor rendimiento y eficiencia energética que la GPU de Nvidia en algoritmos como KNN o SVN.

Apple demuestra con estos chips una arquitectura robusta y fácil de programar en GPU, con estudios como el de Gebraad et al. [37], que elogia la facilidad de implementación que ofrece Metal gracias a la memoria unificada de Apple, que elimina la necesidad de gestionar y copiar datos entre la CPU y la GPU.
Sobre la ANE destaca la investigación de Kasperek et al. [38], que evalúa el rendimiento en *benchmarks* de aprendizaje automático en los tres modelos de chip M1 y portátiles

[13] https://developer.apple.com/documentation/accelerate
[14] https://ss64.com/mac/powermetrics.html

con Intel i5, con resultados muy superiores en los ordenadores de Apple gracias a su acelerador neuronal.

4.2. Entornos de Trabajo

Se han utilizado dos ordenadores Mac disponibles en el grupo de investigación ArTeCS, perteneciente al Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid[15].

Estos son un Mac Mini M1 (Mac M1 abreviado) y un Mac Mini M4 Pro (Mac M4 Pro abreviado).

Ambos han sido utilizados programando en Python con el framework de CoreML, ya que, entre otras ventajas, permite utilizar los tres dispositivos desde el mismo.

4.2.1. Mac M1

El Mac Mini M1 fue lanzado al mercado en 2020 por Apple. Es un ordenador *System On a Chip* (SoC), es decir, con todos sus componentes en un solo chip. Fue el primero en llegar entre los ordenadores de la gama Silicon de Apple y, entre sus características, destaca que posee un acelerador neuronal de dieciséis núcleos llamado *Apple Neural Engine* (ANE).

La CPU del modelo tiene ocho núcleos, cuatro de eficiencia y cuatro de rendimiento, lo que ayuda a tener un buen rendimiento y una buena eficiencia energética, además de una GPU con ocho núcleos[16].

[15] https://artecs.dacya.ucm.es/

[16] https://support.apple.com/es-es/111894

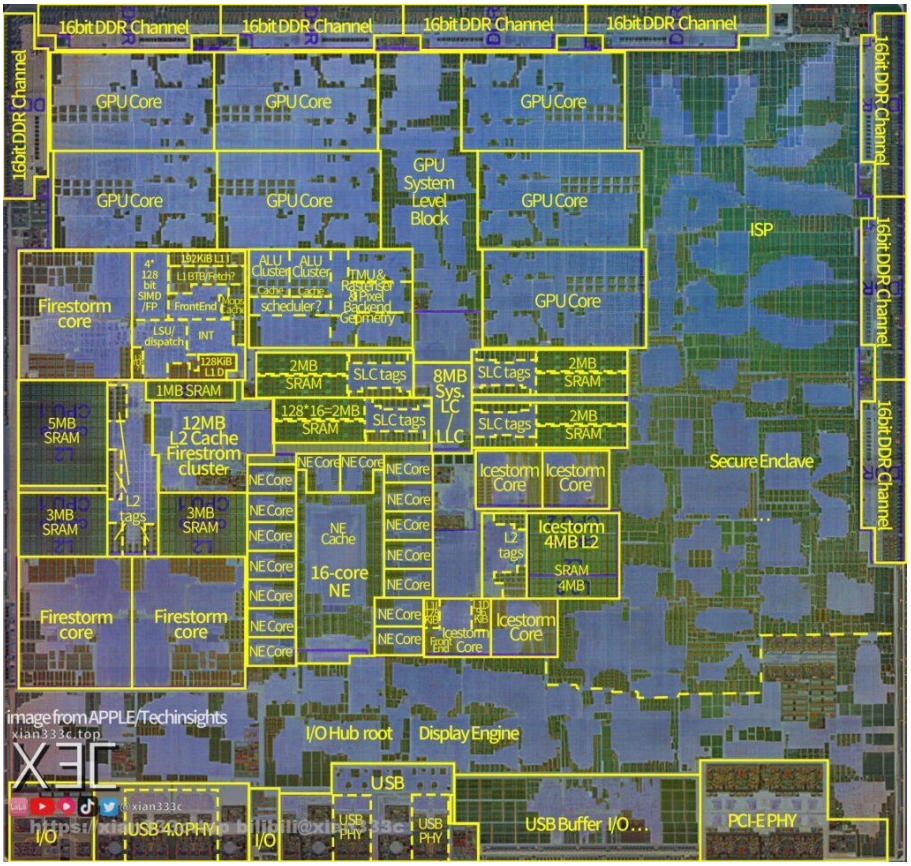


Figura 4.3: Chip Apple Silicon M1[17]

Por la investigación de trabajos anteriores, conocemos de antemano que este modelo cuenta también con un acelerador matricial (AMX) del que se hace uso mediante la CPU [11] [13].

4.2.2. Mac M4 Pro

El Mac Mini M4 Pro es un ordenador SoC perteneciente a la última gama de modelos de gama Silicon de Apple. Fue lanzado en 2024 e integra un chip Apple M4 mejorado, el cual cuenta con una CPU con doce núcleos en total: ocho de rendimiento y cuatro de eficiencia. Esto ayuda a gestionar mayor carga de trabajo en núcleos de rendimiento si lo comparamos con el Mac M1, el otro dispositivo del que disponemos.

La GPU también se ha mejorado, disponiendo ahora de dieciséis núcleos, el doble que el M1.

La ANE, por su parte, se mantiene en dieciséis núcleos, aunque en la sección de resultados veremos que dispone de mayor capacidad, lo que le permite computar datos de mayor

[17] https://xian333c.top/archives/479

tamaño[18].

Gracias a otras investigaciones, conocemos de antemano que este modelo cuenta también con dos aceleradores matriciales (AMX), uno más que el Mac Mini M1[19].

4.2.3. Tipos de Datos Compatibles con cada Dispositivo

La librería de Apple para inteligencia artificial, CoreMLTools, especifica que la ejecución se debe realizar con una de dos precisiones: FP16 o FP32[20].

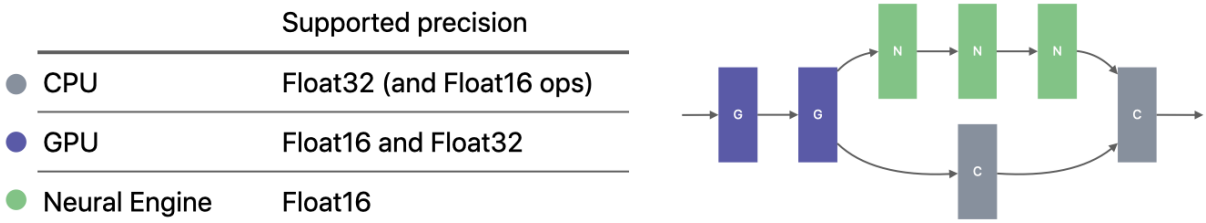


Figura 4.4: Ilustración Oficial de los Tipos de Datos soportados en CoreML[21]

Esta limitación ha llevado a probar en los dispositivos disponibles si son capaces de ejecutar ambos tipos de datos o no, algo que se ha comprobado midiendo su gasto energético.

	Apple M1			Apple M4 Pro		
	CPU	GPU	ANE	CPU	GPU	ANE
FP16	X	X	X	X	X	X
FP32	X	X	-	X	X	-

Cuadro 4.2: Compatibilidad de tipos de datos en dispositivos Apple

Como se observa en la tabla 4.2, en ambos ordenadores la compatibilidad de datos es exactamente igual. La CPU y la GPU son capaces de manejar ambos tipos de datos, mientras que la ANE tan solo FP16.

[18] https://support.apple.com/es-es/121555
[19] https://eclecticlight.co/2024/11/11/inside-m4-chips-p-cores/
[20] https://apple.github.io/coremltools/docs-guides/source/typed-execution.html
[21] https://apple.github.io/coremltools/docs-guides/_images/ml-program-runtime.png

Este tipo de datos es obligatorio para la precisión de ejecución, pero la de almacenamiento se podría usar en FP32. Para medir el tiempo de más en la compilación, este proyecto utilizará el mismo tipo de datos en ejecución y compilación.

4.3. Frameworks y herramientas

En esta sección se tratarán los distintos *frameworks* y herramientas que se han utilizado a lo largo del proyecto.

4.3.1. Python

El principal lenguaje de programación utilizado ha sido Python[22], ya que es el lenguaje líder en cuestiones de aprendizaje automático y uno de los documentados por la librería de Apple de dicho tema. Esta librería se llama CoreMLTools y se ha elegido implementar los algoritmos en ella dado que puede ejecutar código en los tres dispositivos de los ordenadores sin necesidad de crear código adicional.
La versión de Python utilizada ha sido Python *3.9.6* mediante la creación de un entorno virtual (PyEnv).

PyTorch

PyTorch[23][39][40] es una de las principales librerías utilizadas en el ámbito del aprendizaje automático.
La librería está basada en Torch y permite la diferenciación automática. Funciona gracias a grafos computacionales, representando cada operación mediante un nodo que almacena información sobre el gradiente de la función.
Asimismo, una de las principales características de esta librería es la de implementar la clase tensor, la cual funciona de forma similar a un vector pero con la característica de ser operada en GPUs. Los tensores son utilizados en todas las librerías de aprendizaje automático y son uno de los pilares de dicho campo.
Esta librería es primordial en este proyecto, ya que se han recreado los algoritmos de propósito general seleccionados en la misma para poder convertirse posteriormente al

[22] https://www.python.org/about/
[23] https://pytorch.org/

formato propio de Apple con el objetivo de ser ejecutados en la ANE.

Este trabajo ha utilizado la versión *2.3.0* de la librería.

Torchvision es un *framework*[24] perteneciente a la librería PyTorch, el cual incluye *datasets* populares, arquitecturas de modelos, utilidades para visión por computadora y algunas operaciones.
En este proyecto se ha utilizado la versión *0.18.0* de torchvision.

Numpy

Numpy[25] [41] es una librería de Python que permite la creación de matrices multidimensionales y vectores, y la utilización de operaciones matemáticas de alto nivel.
Este proyecto ha utilizado la versión *2.0.2* de la librería.

4.3.2. CoreMLTools

CoreMLTools[26] [27] es un *framework* desarrollado por Apple, el cual permite integrar modelos de aprendizaje automático en sus sistemas, pudiendo utilizar todos los dispositivos del mismo: CPU, GPU y ANE.

Esta herramienta permite convertir modelos desde las principales librerías de inteligencia artificial como PyTorch o TensorFlow al formato propio de Apple[28].

[24] https://docs.pytorch.org/vision/stable/index.html
[25] https://numpy.org/
[26] https://github.com/apple/coremltools
[27] https://apple.github.io/coremltools/docs-guides/
[28] https://coremltools.readme.io/v6.3/docs/unified-conversion-api

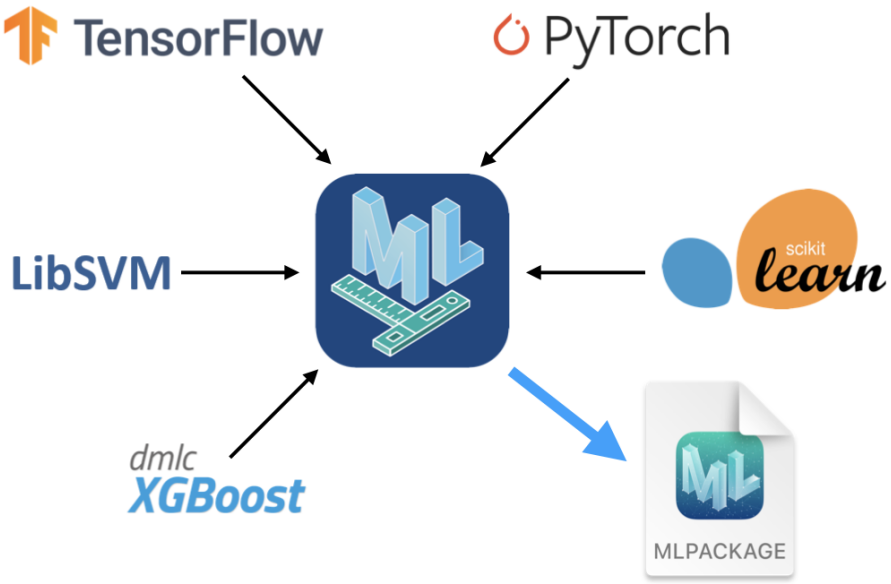


Figura 4.5: Ilustración sobre cómo CoreML es capaz de transformar modelos de las principales librerías de aprendizaje automático[29]

El *framework* también ofrece diversas utilidades, como definir operadores personalizados que se utilicen en otros *framework*[30], definir parámetros de entrada de longitud variable[31] o aplicar compresión de pesos a las redes, pudiendo cuantizar a FP16 o INT8.[32]

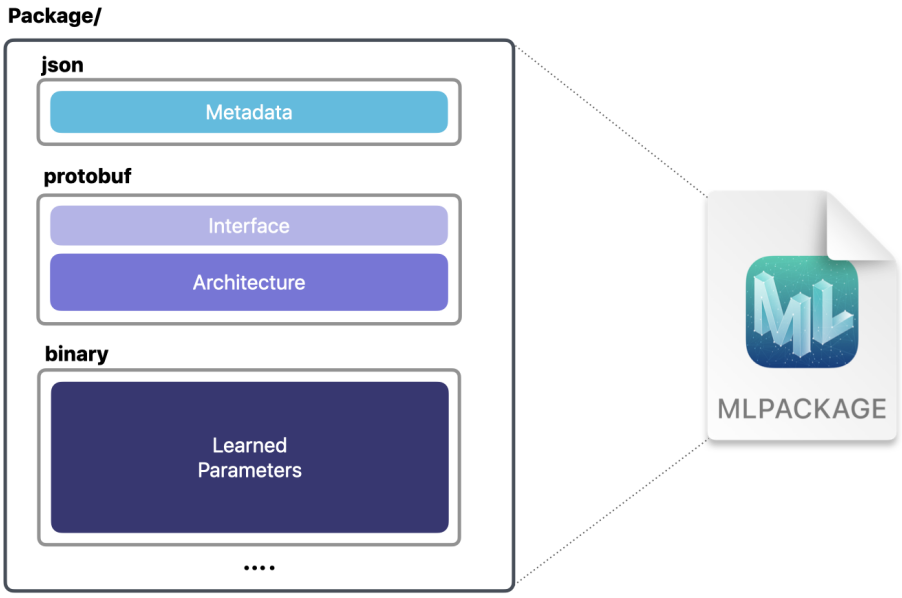


Figura 4.6: Ilustración sobre la composición de un archivo MLPackage[33]

[29] https://coremltools.readme.io/v6.3/docs/what-are-coreml-tools
[30] https://coremltools.readme.io/v6.3/docs/custom-operators
[31] https://coremltools.readme.io/v6.3/docs/flexible-inputs
[32] https://coremltools.readme.io/v6.3/docs/quantization
[33] https://coremltools.readme.io/v6.3/docs/new-in-coremltools#save-a-core-ml-model-package

Los archivos creados con este trabajo son del formato *.mlpackage*, un formato propio de Apple.

Este formato es un paquete de MacOS que organiza los diferentes componentes de un modelo de aprendizaje automático, tal y como podemos observar en la figura 4.6[34] [35] .

Dichos componentes son los siguientes:

- Archivo *Manifest* (JSON): archivo en el que se documentan todos los archivos archivos que deben aparecer y su localización.
- Metadatos (JSON): se almacenan metadatos sobre el modelo, como podrían ser el autor, la licencia o una breve descripción del mismo[36].
- Estructura del modelo (.mlmodel): la estructura, configuración y métodos de predicción del modelo vienen juntos en un archivo .mlmodel[37] [38].

  Este archivo funciona internamente con la especificación Model.proto que Apple documentada en su GitHub[39]. Gracias a la especificación *protobuf*, el modelo se divide en descripción, parámetros y metadatos.
- Parámetros aprendidos (.bin): MLPackage aísla los parámetros aprendidos como pesos y sesgos en sus propios archivos. Esta característica se realiza para tener una mayor flexibilidad a la hora de gestionar estos datos[40] [41].

El programa a su vez puede funcionar en tres modos dependiendo de qué dispositivos queramos utilizar:

- *ct.ComputeUnit.ALL*: Ejecución entre los tres dispositivos: CPU, GPU y ANE.
- *ct.ComputeUnit.CPU_ONLY*: Ejecución sólo en CPU.
- *ct.ComputeUnit.CPU_AND_GPU*: Ejecución en CPU y GPU.
- *ct.ComputeUnit.CPU_AND_ANE*: Ejecución en CPU y ANE.

El propio *framework* gestiona en cuáles de los dispositivos seleccionados se ejecuta cada

[34] https://developer.apple.com/videos/play/wwdc2021/10038/
[35] https://coremltools.readme.io/v6.3/docs/new-in-coremltools#save-a-core-ml-model-package
[36] https://apple.github.io/coremltools/docs-guides/source/mlmodel.html
[37] https://apple.github.io/coremltools/docs-guides/source/mlmodel.html
[38] https://coremltools.readme.io/v6.3/docs/ml-programs
[39] https://github.com/apple/coremltools/blob/main/mlmodel/format/Model.proto
[40] https://apple.github.io/coremltools/docs-guides/source/convert-to-ml-program.html
[41] https://coremltools.readme.io/v6.3/docs/ml-programs

una de las operaciones, sin opciones por parte del desarrollador de poder elegirlo manualmente.

CoreMLTools se ha utilizado en Python para ejecutar en la ANE todos los modelos probados en este trabajo, dado que permite ejecutar en todos los dispositivos sin necesidad de generar código adicional. Además, es la única forma oficial de ejecutar código en la ANE.

La versión utilizada ha sido la versión *8.0b2*.

4.3.3. Powermetrics

PowerMetrics[42] es una instrucción de consola presente en MacOS. Esta instrucción permite saber el consumo energético de los distintos dispositivos presentes en la máquina y se ha utilizado para realizar mediciones precisas de consumo.

Powermetrics se ha utilizado ejecutando el siguiente comando:

```
sudo powermetrics -i 100 --samplers cpu_power -a --hide-cpu-duty-cycle
--show-usage-summary --show-extra-power-info --show-process-energy
```

Este comando devuelve cada 100 milisegundos, el consumo total del ordenador y el consumo individual de cada dispositivo.

4.3.4. Asitop

Asitop [43] es una herramienta de monitorización de rendimiento *OpenSource* para ordenadores Apple Silicon. Esta herramienta permite ver en tiempo real el uso de los diferentes componentes de la máquina por medio de una interfaz en consola.

[42] https://ss64.com/mac/powermetrics.html
[43] https://github.com/tlkh/asitop

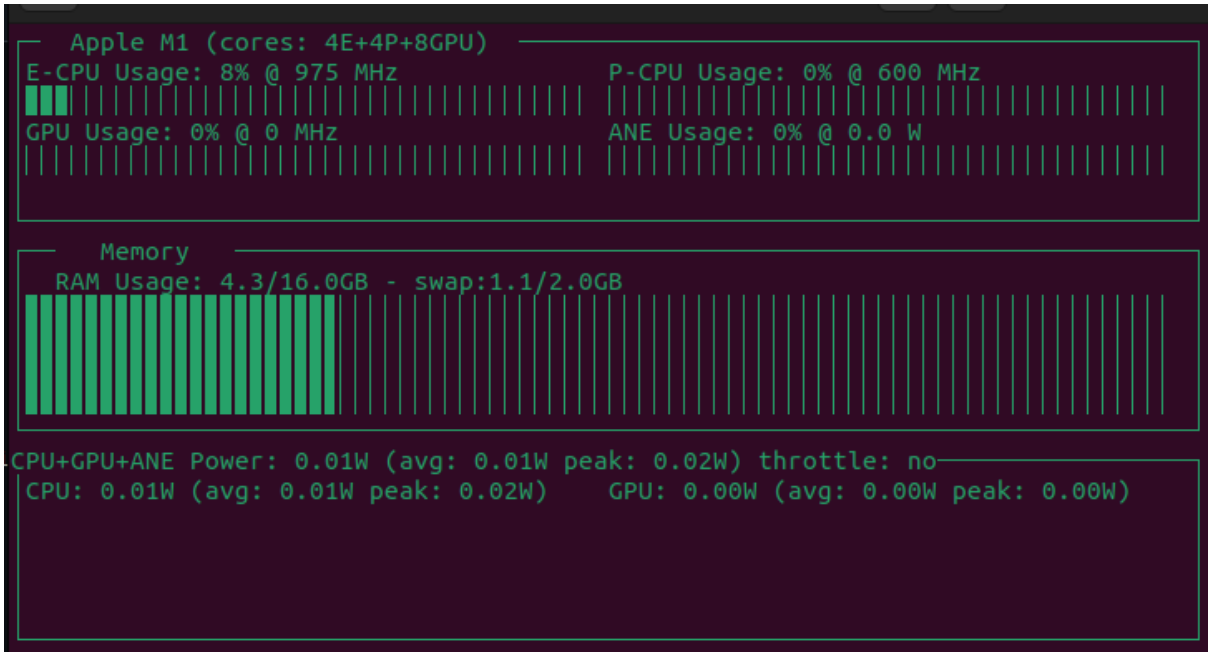


Figura 4.7: Ejemplo de uso de Asitop en el Mac M1 Utilizado

Para asegurar que se está usando la ANE, al convertir los modelos se realizan varias ejecuciones de pruebas que se monitorizan en otra consola con la herramienta Asitop, que se invoca con el siguiente comando:

Listing 4.1: Comando para utilizar Asitop

```
sudo python3 -m asitop.asitop
```

4.3.5. XCode

XCode[44] es un entorno de desarrollo para MacOS que, entre sus múltiples funciones, permite visualizar información sobre modelos de la librería CoreML.

[44] https://developer.apple.com/xcode/

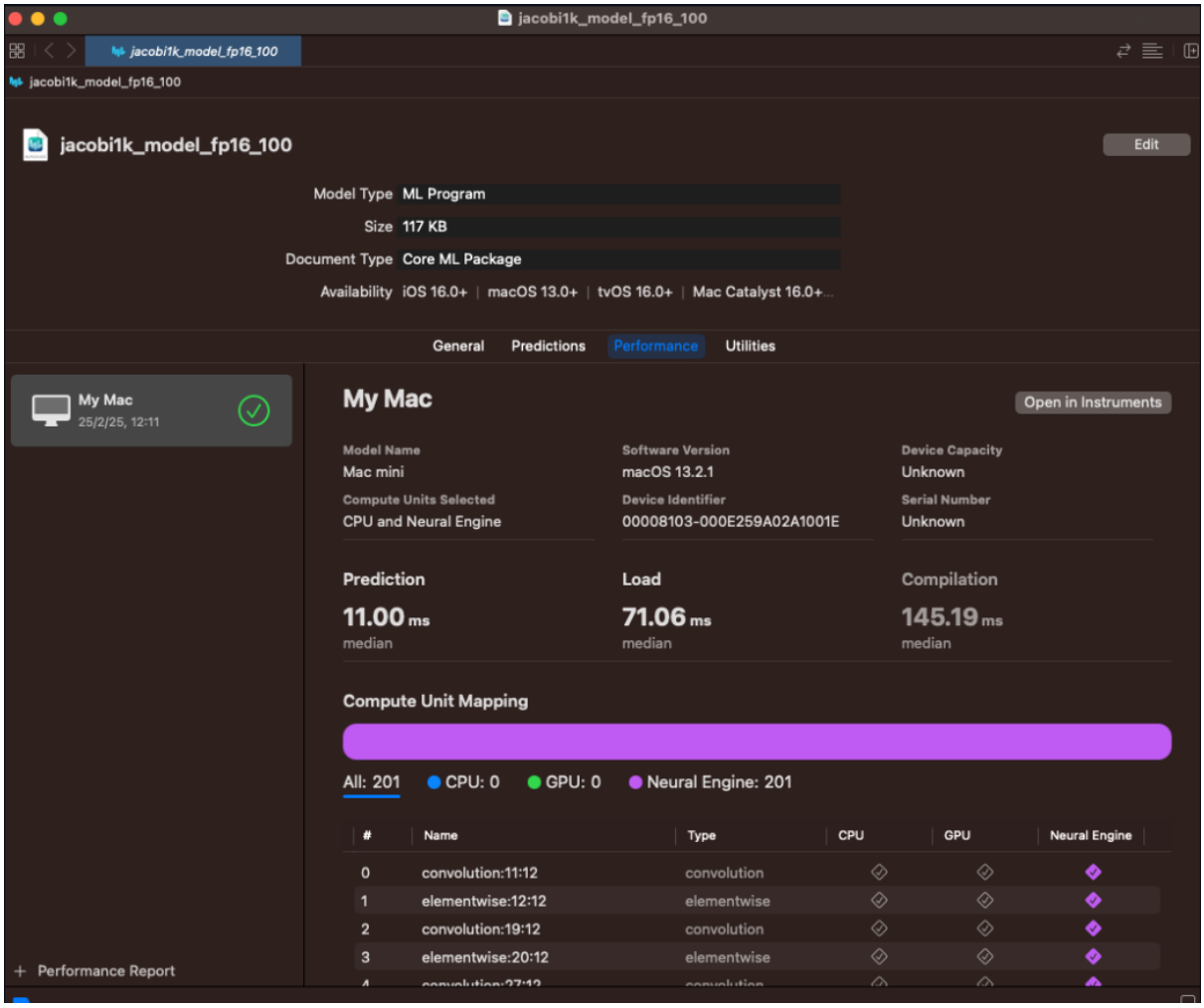


Figura 4.8: Ejemplo de visualización de un modelo Jacobi con XCode

En la figura 4.8 podemos ver esta visualización de los modelos en su pestaña de *Performance*, que nos deja ejecutar en *Performance Report* una prueba en el modo deseado (CPU, CPU y GPU, CPU y ANE o todos) que mide diferentes *benchmark* y, lo más relevante para este trabajo, permite visualizar en qué dispositivo se ha lanzado cada una de las instrucciones del modelo.

Otra pestaña interesante es la pestaña *General*, que nos indica con qué tipos de datos computa el modelo, así como cuáles son sus datos de almacenamiento y qué instrucciones a nivel de máquina utiliza.

Una vez se ha comprobado el uso de la ANE al observar consumo energético con la herramienta Asitop, se ha comprobado que el 100 % del modelo lo utilice por medio de XCode, abriendo el modelo con la herramienta. MacOS por defecto, abre los archivos MLProgram y MLPackage con este entorno.

4.3.6. Shell

Shell es el intérprete de comandos de los sistemas basados en Unix. Gracias a este sistema, se puede crear lo que se denomina *Script* de Shell, que es un programa ejecutable que ejecuta diferentes comandos que el sistema debe ejecutar.
En este proyecto se han creado diferentes *scripts* de Shell para poder automatizar las pruebas.

4.3.7. LaTeX y Overleaf

LaTeX[45] es un sistema de composición de textos enfocado a la confección de artículos, documentos y libros científicos de alta calidad tipográfica.
Su principal ventaja de uso es el hecho de que permite dividir el contenido del documento de forma clara, ajustar el formato de manera muy precisa, su ya comentada alta calidad tipográfica y las facilidades para crear índice y bibliografía.
Esta memoria ha sido escrita utilizando LaTeX.

Overleaf[46] es un editor de documentos LaTeX colaborativo basado en la nube. Permite la compilación automática del código, de forma que el documento resultante pueda observarse de manera simultánea y ofrece opciones de comentarios e historial.
Overleaf ha sido utilizado para confeccionar esta memoria, permitiendo en todo momento que el director tenga acceso al documento en su totalidad y pueda hacer comentarios en las secciones que considerara pertinentes.

4.3.8. GitHub

GitHub[47] es una plataforma de desarrollo colaborativo en la que se alojan proyectos controlando las versiones mediante el sistema Git[48].
GitHub se ha utilizado durante todo el proyecto para mantener un repositorio con el proyecto siguiendo un flujo de trabajo Git adecuado. El repositorio es accesible y se puede encontrar en el apartado 2.3.2.

[45] https://www.latex-project.org
[46] https://www.overleaf.com/about
[47] https://github.com/
[48] https://git-scm.com/

Capítulo 5

Algoritmos

5.1. Algoritmos Utilizados

Para el desarrollo del estudio se comenzó el análisis tomando como punto de partida un modelo YOLOv3 descargado de la página oficial de Apple para poder medir la auténtica eficiencia de la ANE.
Una vez comprobado, el primer punto de partida fue analizar una simple multiplicación de matrices, para posteriormente adaptar el algoritmo de Jacobi y un algoritmo Multigrid, ambos para resolver la ecuación del calor.

En esta sección se explica cómo funcionan los algoritmos en un contexto general. La implementación específica de los algoritmos generados se encuentra en la sección 6.

5.2. YOLOv3

El primer paso de la investigación fue probar un modelo de inteligencia artificial.

YOLOv3 es un modelo de detección de objetos capaz de clasificar ochenta clases diferentes. Fue descargado de la página oficial de Apple, donde hay varios modelos de inteligencia artificial de los más comunes publicados[1].
YOLO es uno de los modelos más utilizados de inteligencia artificial, lo que ha sido determinante en su elección como modelo a probar, además de la experiencia previa del autor de este trabajo con estos modelos.

Para ejecutar este algoritmo, requieren imágenes de un tamaño de 416x416 píxeles, por lo que se ha creado un *script* donde se crea una matriz aleatoria de ese tamaño, ejecutando cien veces el modelo con ella para obtener el mejor tiempo posible de entre esas ejecuciones y su consumo energético medio.

[1] https://developer.apple.com/machine-learning/models/

5.3. YOLOv11

Utilizando el tutorial oficial de Apple sobre conversión de modelos a MLProgram[2], se han podido convertir todos los modelos YOLOv11. Los modelos fueron descargados del repositorio oficial en formato ".pt"(Pytorch)[3]
YOLO 11 es un modelo de inteligencia artificial que se puede utilizar para diferentes tareas, como detección de objetos, segmentación, clasificación, detección de pose y creación de cajas contenedoras orientadas. Para este ejemplo, el modelo se utilizará en el modo detección de objetos.

Para pasar el modelo a CoreML se han tenido que seguir varios pasos, ya que en primer lugar es necesario utilizar la librería de YOLO para pasar el modelo a TorchScript[4] para que luego se pueda cargar dicho archivo con la librería PyTorch y convertir con CoreML a MLProgram siguiendo el tutorial oficial[5].

5.4. GEMM: *General matrix multiply*

El primer algoritmo de propósito general analizado fue la Multiplicación de Matrices, un algoritmo que se suponía de fácil adaptación, al ser una operación ya implementada en PyTorch y en CoreML que se suele utilizar en *Machine Learning* muy a menudo.
La operación de multiplicación de matrices (abreviada GEMM en inglés) consiste en multiplicar dos matrices de tamaño compatible. Esto significa que el número de columnas en la primera matriz debe ser el mismo que el número de filas en la segunda matriz. La matriz resultante tiene el mismo número de filas que la primera matriz y el mismo número de columnas que la segunda matriz.
Esta operación se ha utilizado como *benchmark* para medir el rendimiento de ambas arquitecturas debido a su enorme popularidad en el ámbito científico. Esta popularidad ha desembocado en la creación de librerías especializadas para explorar los múltiples niveles de paralelismo de la arquitectura y optimizar la jerarquía de memoria. La especificación BLAS (*Basic Linear Algebra Subprograms)* incorpora la operación GEMM en el nivel 3.[6]

[2] https://apple.github.io/coremltools/docs-guides/source/convert-to-ml-program.html
[3] https://github.com/ultralytics/ultralytics
[4] https://docs.ultralytics.com/es/integrations/torchscript/
[5] https://apple.github.io/coremltools/docs-guides/source/convert-to-ml-program.html
[6] https://www.netlib.org/blas/

La operación ha sido implementada en multitud de implementaciones propietarias como oneMKL para arquitecturas Intel[7] o cuBLAS de Nvidia[8]. Existen también multitud de alternativas de código libre como ATLAS[9], OpenBLAS[10] o BLIS[11].

$$A \cdot B = C \tag{5.1}$$

Tamaños	M1	M4 Pro
256x256	X	X
512x512	X	X
1024x1024	X	X
2048x2048	X	X
4096x4096	X	X
8192x8192	X	X
12288x12288	-	X
14336x14336	-	X
16384x16384	-	X

Cuadro 5.1: Tabla de Tamaños Probados en la Multiplicación de Matrices

En la tabla 5.1 encontramos los tamaños probados en la multiplicación de matrices, donde se aprecia que 16384x16384 no llega a probarse en el M1 dado que la ANE tiene menor capacidad.

Al ser tal y como se ha comentado una operación muy común en algoritmos de *Machine Learning*, la multiplicación de matrices se adapta fácilmente mediante un simple *torch.matmul(A, B)* en el *forward* de un modelo.

5.5. Jacobi

El método de Jacobi es un algoritmo iterativo utilizado para resolver sistemas de ecuaciones diferenciales estrictamente diagonalmente dominantes.

[7] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
[8] https://developer.nvidia.com/cublas
[9] https://math-atlas.sourceforge.net/
[10] http://www.openmathlib.org/OpenBLAS/
[11] https://github.com/flame/blis

En este trabajo, se ha utilizado para resolver la ecuación del calor, una famosa ecuación diferencial que define cómo se distribuye el calor en una región a lo largo del tiempo. Su ecuación representativa se puede ver a continuación.

$$\frac{\partial u}{\partial t} = \left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2} \right) \tag{5.2}$$

Para resolver el problema iterativamente con el método Jacobi, se deben seguir los siguientes pasos:

- Inicializar la malla con la temperatura inicial de cada punto.
- En cada iteración, calcularemos la temperatura de cada punto en función de la temperatura de los vecinos en la iteración anterior.
- Repetimos el proceso hasta que pase el tiempo deseado o se cumpla una condición de convergencia.

Este tipo de problemas suelen tener también lo que se conoce como condiciones de frontera. Los puntos de frontera son puntos cuyo valor no cambia nunca. Por ejemplo, para la ecuación del calor podrían ser los bordes de una malla metálica, que al no cambiar su temperatura, guían a los vecinos interiores a actualizarse correctamente.

El pseudocódigo puede describirse de la siguiente manera:

Algorithm 1: Pseudocódigo de Jacobi

```
// Paso 1: Inicializar la malla
k ← 0;
while k <maxIter do
    // Paso 2: Actualizar vecinos
    for i ← 1 to n do
        σ ← 0;
        for j ← 1 to n do
            if j ≠ i then
                // Se actualiza cada punto
                σ ← σ + a_ij · x_j^(k);
        // Actualizamos la malla con los nuevos valores
        x_i^(k+1) ← (b_i − σ) / a_ii;
        // Paso 3: Mantener las condiciones de frontera
        if mask_i = 1 then
            x_i^(k+1) ← x_i^(0);
    // Paso 4: Comprobamos condiciones de convergencia
    if ‖x^(k+1) − x^(k)‖ < ε then
        break // Solución encontrada
    Increment k;
```

En la tabla 5.2 encontramos los tamaños probados en el método de Jacobi. Se probaron menos en el M1 dado que en la ANE de este computador el tamaño máximo de datos es menor.

Tamaños	M1	M4 Pro
250x250	X	X
500x500	X	X
750x750	X	X
1000x1000	X	X
2000x2000	X	X
3000x3000	X	X
5000x5000	X	X
7000x7000	X	X
8000x8000	X	X
10000x10000	X	X
15000x15000	X	X
17500x17500	-	X
20000x20000	-	X

Cuadro 5.2: Tabla de Tamaños Probados en el Método de Jacobi

Este algoritmo requiere una adaptación mucho mayor que la multiplicación de matrices, dado que existen diversas operaciones que no son posibles de ejecutar en la ANE.

5.6. Multigrid

El método Multigrid es otro método de resolución de ecuaciones diferenciales. Este método aplica distintas discretizaciones, de forma que el problema se haga más sencillo y pueda resolverse más rápido.

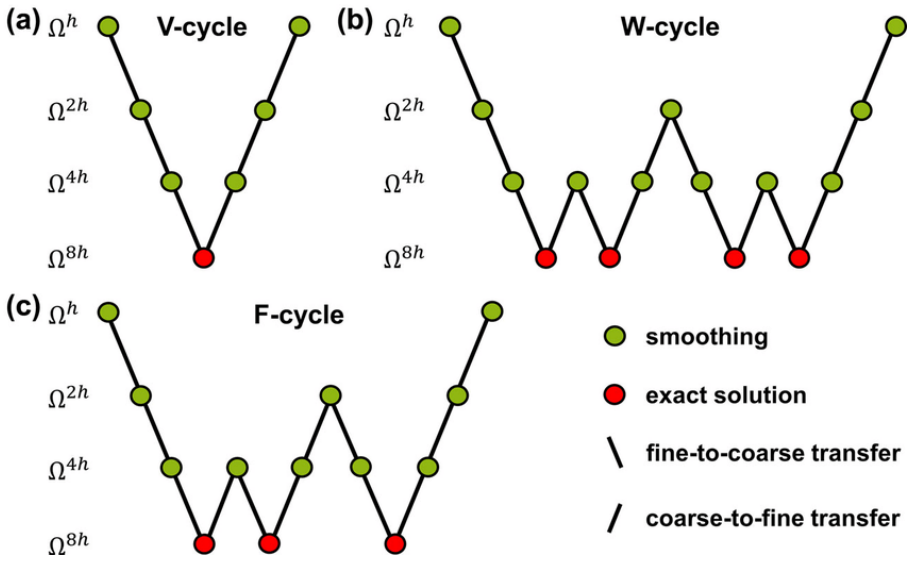


Figura 5.1: Diferentes Aproximaciones del Método Multigrid[12]

Se ha utilizado el Multigrid *V-Cycle*, la versión más sencilla de este algoritmo. Esto se debe a que restaura el tamaño una única vez y, al ser la misma operación, si se adapta a la ANE, podría usarse una o más veces, lo que la convierte en la más sencilla de probar en este trabajo.

La resolución presentada es la de la ecuación del calor, al igual que el método Jacobi explicado en la sección 5.5.

El algoritmo consiste en los siguientes pasos:

- Pre-suavizado (*Smoothing*): Se aplican varias iteraciones de un método iterativo (en el caso de este trabajo, Jacobi) en la malla fina (la malla más grande).
- Cálculo del Residuo: se calcula el error residual de la solución. El residuo es la representación aproximada de cuán lejos está la solución actual de la solución verdadera.
- Restricción (*Restriction*): El residuo se transfiere a una malla más gruesa (más pequeña), reduciendo así el tamaño del problema.
- Resolución en la Malla Gruesa: Se resuelve el problema en la malla más gruesa de todas.
- Prolongación y Corrección (*Prolongation & Correction*): La solución obtenida en la malla gruesa se interpola y se agranda a una malla más fina. Una vez interpolada, se suma a la solución que teníamos en la malla fina para corregirla.
- Post-suavizado (*Smoothing*): Finalmente, se aplican de nuevo iteraciones de Jacobi para eliminar errores que hayan podido ser producidos durante el paso de prolongación.

Podemos dividir el proceso en tres fases:

- Bajada: es cuando «bajamos»en la «V». En esta etapa se realiza el pre-suavizado, el cálculo del residuo y la restricción.
- Fondo: es el punto bajo de la «V», donde se realiza la resolución en la malla gruesa.
- Subida: es cuando «subimos»en la «V». En esta etapa se hace la prolongación y

[12] https://www.researchgate.net/figure/Structure-of-one-multigrid-cycle-a-V-cycle-b-W-cycle-and-c-F-cycle-The-finest_fig2_308388169

corrección y el post-suavizado.

El pseudocódigo del algoritmo es el siguiente:

Algorithm 2: *V-Cycle* Multigrid

Dado: ϕ, f, h ;
$\phi \leftarrow \text{smoothing}(\phi, f, h)$; // Pre-suavizado
$r \leftarrow f - \nabla^2\phi$; // Residuos
$\text{rhs} \leftarrow \mathcal{R}(r)$; // Restricción
$\varepsilon \leftarrow 0$ en la malla gruesa;
$\varepsilon \leftarrow \text{resolver\_grueso}(\text{rhs}, 2h)$;
$\phi \leftarrow \phi + \mathcal{P}(\varepsilon)$; // Prolongación
$\phi \leftarrow \text{smoothing}(\phi, f, h)$; // Post-suavizado
return ϕ

El método Multigrid ha sido probado con los tamaños mostrados en la tabla 5.3. Como es habitual, en el Mac M1 se ha probado hasta un tamaño menor debido a la menor capacidad de la ANE.

Tamaños	M1	M4 Pro
512x512	X	X
1024x1024	X	X
2048x2048	X	X
4096x4096	X	X
8192x8192	X	X
10240x10240	X	X
12288x12288	X	X
16384x16384	X	X
32768x32768	-	X

Cuadro 5.3: Tabla de Tamaños Probados en el Método Multigrid

La complejidad para adaptar este algoritmo a CoreML es mucho mayor que la del método Jacobi, dado que se presentan más operaciones que se deben reestructurar de manera diferente. Es por ello que se ha realizado una comprobación adicional sobre si el inicio y resultado son los mismos en una resolución típica de la ecuación del calor y este método.

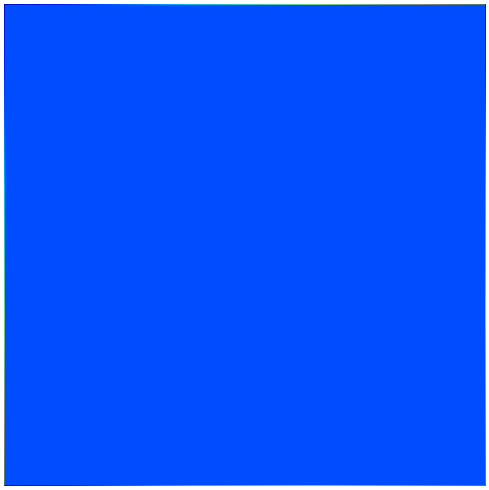
5.6.1. Comprobación del Método Multigrid

Al ser el algoritmo Multigrid tan complejo de implementar debido a la gran cantidad de cambios necesarios para su funcionamiento en la ANE, se han realizado pruebas para poder asegurar su correcto funcionamiento.

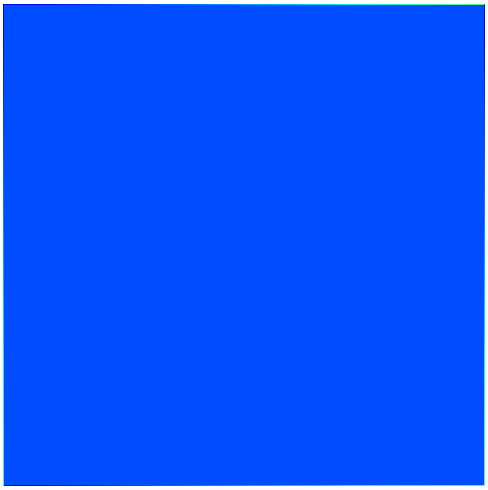
Para ello, se ha programado un algoritmo que realice la ecuación del calor para poder comparar numéricamente si el resultado es similar basándose en repositorios de internet [13]. Se han guardado imágenes de su estado inicial y su estado final para poder visualizar esta característica. Se ha hecho lo mismo para la aproximación realizada con Multigrid con el fin de comparar si se obtiene un resultado similar. Esta prueba se realizó en el Mac M4 Pro dadas sus mayores prestaciones frente al M1.

En la figura 5.2 podemos ver cómo los resultados obtenidos para ambas ejecuciones son similares, por lo que podemos determinar que la aproximación Multigrid funciona correctamente.

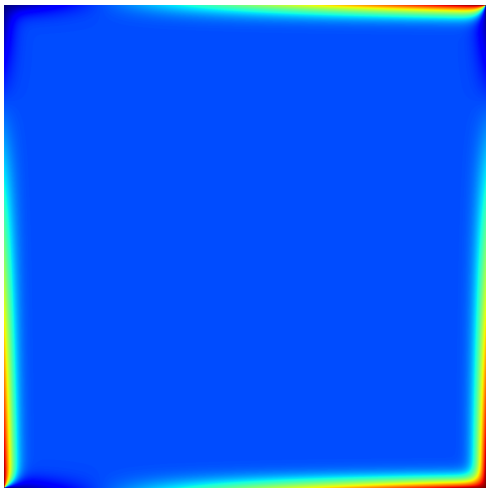
[13] https://github.com/Younes-Toumi/Youtube-Channel/



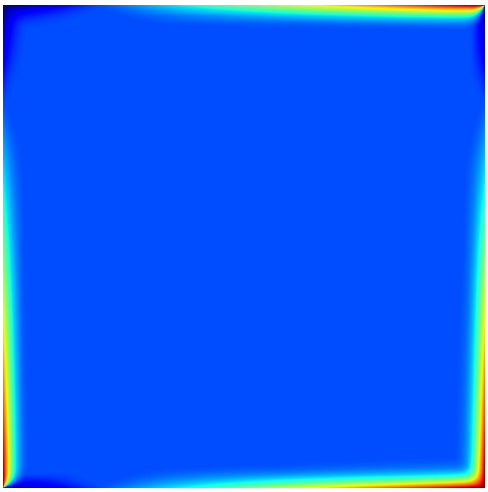
(a) Estado Inicial de la malla en la Ecuación del Calor Clásica



(b) Estado Inicial de la malla en el Método Multigrid



(c) Estado Final de la malla en la Ecuación del Calor Clásica



(d) Estado Final de la malla en el Método Multigrid

Figura 5.2: Resolución de una malla con la Ecuación del Clásica y el Método Multigrid

Capítulo 6

Metodología

6.1. Flujo de trabajo

En esta sección se explica en qué consiste y cómo se realiza el flujo de trabajo utilizado para cada uno de los modelos.

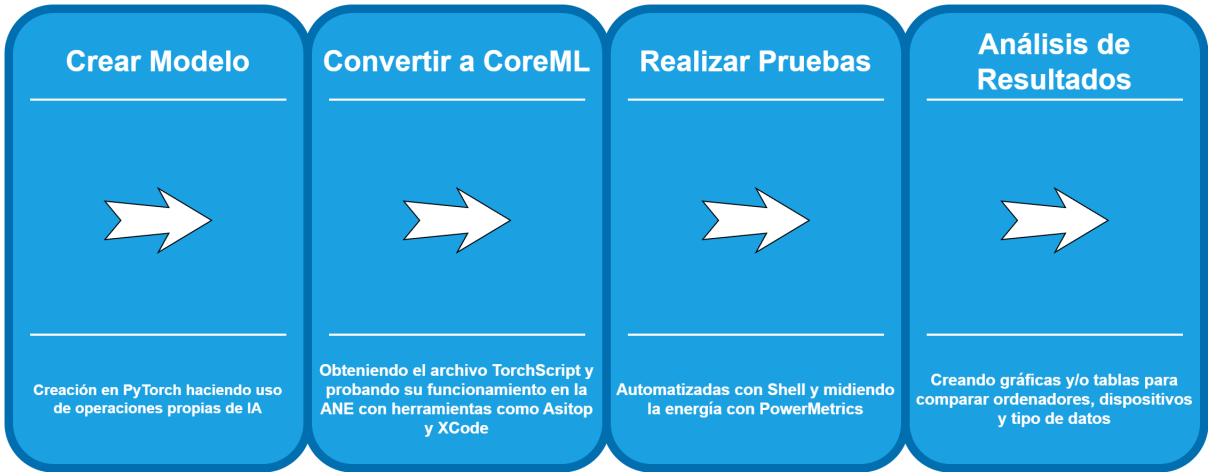


Figura 6.1: Flujo de Trabajo utilizado con cada Modelo

Para cada modelo, este se crea previamente en PyTorch, asegurando que el modelo funcione correctamente.
Posteriormente, el modelo se convierte a CoreML asegurando su correcto uso en la ANE.
Una vez completada la conversión, se realizan las pruebas correspondientes al modelo y se pasa a analizar sus resultados.

Esto se repite para cada uno de los modelos probados, a excepción de los YOLOs, que ya estaban creados. El YOLOv3, además, se ha descargado directamente en formato CoreML, por lo que tampoco ha sido necesaria su conversión.

6.2. Creación de Modelos

Para poder ejecutar programas en la ANE, estos deben ser modelos de *machine learning* en un formato que se pueda ejecutar en el dispositivo, ya que es la única forma oficial que Apple documenta para ello [13] [34].
Para crear los modelos, se ha utilizado PyTorch, utilizando Asitop durante una prueba previa para poder saber si la ANE se estaba utilizando y posteriormente asegurándose

con XCode[1] [2], ya que la herramienta permite visualizar dónde se está ejecutando cada una de las instrucciones del modelo.

El proceso para crear un modelo es el mismo que para crear cualquier otro modelo de PyTorch, salvo por la complejidad de no saber exactamente qué operaciones están soportadas y cuáles no, lo cual se ha ido descubriendo tanto por un proceso de prueba propio como por ayuda de algunos usuarios de internet, como la página de GitHub de Matthijs Hollemans[3] y su libro «Core ML Survival Guide» [34].

Además de esta documentación adicional, se ha revisado la API oficial de Apple[4].

6.2.1. Consideraciones para crear Modelos

Para implementar y ejecutar los modelos diseñados en este trabajo, se han tenido en cuenta una serie de requisitos a cumplir derivados de la arquitectura de la ANE y las herramientas utilizadas. Estas consideraciones han influido directamente en el diseño de cada arquitectura presentada en las siguientes secciones (Jacobi, GEMM y Multigrid).

A continuación se describen puntos que aseguran la compatibilidad y eficiencia del despliegue:

- Compatibilidad con ANE: se han evitado operaciones no soportadas como *scatter*.
- Estructura del modelo: todos los modelos están encapsulados en una clase *nn.Module* con un método *forward*, compatible con *torch.jit.trace()* para su conversión posterior a *TorchScript*.
- Conversión a MLProgram: se ha seguido un flujo de conversión mediante *torch.jit.trace* a *CoreMLTools* a modelo final con *backend MLProgram*.
- Pruebas progresivas: cada modelo ha sido probado primero en PyTorch, luego convertido, y finalmente ejecutado en MLProgram, realizando comprobaciones con Asitop, Powermetrics y XCode para validar su ejecución sobre ANE.

Estas consideraciones se han tenido en cuenta para todos los modelos implementados, los cuales se detallan a continuación.

[1] https://fritz.ai/does-my-core-ml-model-run-on-apples-neural-engine/
[2] https://developer.apple.com/videos/play/wwdc2024/10161/
[3] https://github.com/hollance/neural-engine
[4] apple.github.io/coremltools/index.html

6.2.2. Multiplicación de Matrices

La multiplicación de matrices se ha implementado definiendo como la función *forward* del modelo de inteligencia artificial una simple multiplicación de matrices de PyTorch.

```
1 class MyMachine(nn.Module):
2     def __init__(self):
3         super(MyMachine, self).__init__()
4 
5     def forward(self, A, B):
6         x = torch.matmul(A, B)
7         return x
```

Listing 6.1: Código de la Multiplicación de Matrices

Con este código se puede crear el modelo de la multiplicación de matrices.

6.2.3. Jacobi

El modelo Jacobi para resolver la ecuación del calor es un modelo con una complejidad mayor de adaptar con respecto a la multiplicación de matrices.

El punto principal que debe adaptarse es el cálculo del nuevo valor de cada punto a partir de los vecinos en cada una de las iteraciones. Esto se ve sustituido por una convolución 2D que, con ayuda de una máscara precalculada fuera del modelo, calcula correctamente el nuevo valor de la malla.

En concreto, se crea un *kernel* para calcular para cada punto de la malla su nuevo valor a partir del promedio de cuatro vecinos: arriba, abajo, izquierda, derecha. Esto se logra estableciendo en una matriz 3x3 con 0.25 como valor en esas cuatro posiciones y 0 en el resto.

Con este *kernel*, se realiza una convolución 2D que utiliza el *kernel* como un filtro y posteriormente suma los valores obtenidos.

El parámetro *padding=1* añade una nueva columna y fila extra de ceros a la malla durante la ejecución con el objetivo de que se pueda calcular el valor para todas las posiciones. Si no se hiciera esto, la malla se haría más pequeña y no conservaría su tamaño original.

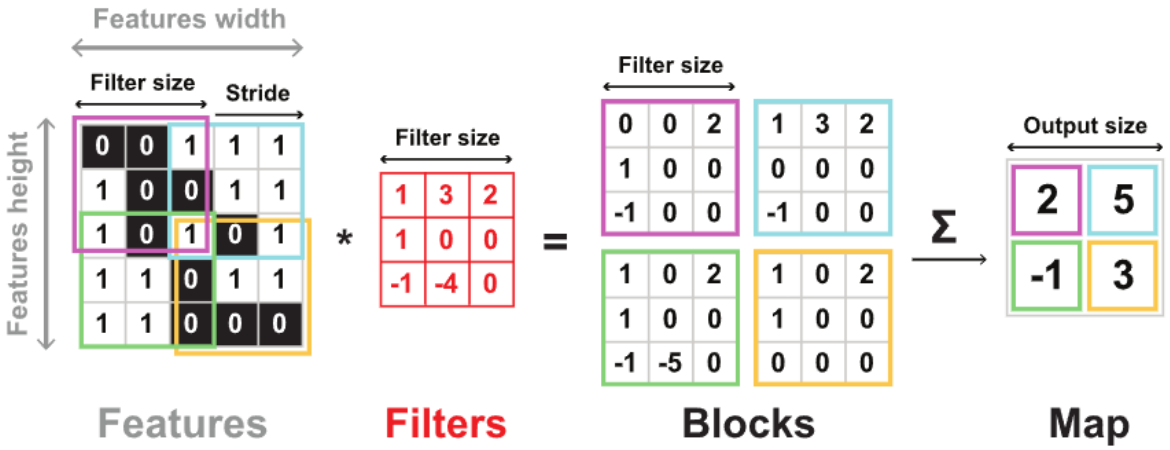


Figura 6.2: Ejemplo de Cómo funciona una Convolución 2D[5]

Complementariamente a la convolución, se usa una máscara con el propósito de mantener las condiciones de frontera tras realizar la convolución, multiplicando el resultado de la convolución por esta máscara. La máscara contiene ceros en los bordes de la malla y unos en el interior para realizar esta tarea.

La máscara es calculada fuera dado que las operaciones de *scatter* (modificación de valores concretos en un tensor) no se pueden calcular en la ANE. La creación de las máscaras genera operaciones de ese tipo, según indica XCode.

```
class JacobiMachine(nn.Module):
    def __init__(self, nt=1000, datatype=torch.float32):
        super(JacobiMachine, self).__init__()
        self.datatype=datatype
        self.nt = torch.tensor(nt, dtype=self.datatype)

    def forward(self, X, X_prev, Mask):
      x = X
      x_prev = X_prev
      mask = Mask
      kernel = torch.tensor([[0.0, 0.25, 0.0],
                             [0.25, 0.0, 0.25],
                             [0.0, 0.25, 0.0]], dtype=self.datatype).view
```

[5] https://epynn.net/_images/convolution0-01.svg

```
                                (1, 1, 3, 3)
14
15      i = torch.tensor(0, dtype=self.datatype)
16
17      while torch.ne(i, self.nt):
18          x_prev = x.clone()
19          x_next = F.conv2d(x_prev, kernel, padding=1)
20          x = x_next * mask
21          diff = torch.max(torch.abs(x - x_prev))
22          i = torch.add(i, 1)
23      return x
```

Listing 6.2: Código de Jacobi

El código visto en este apartado es el utilizado para la creación del modelo Jacobi.

6.2.4. Multigrid

La implementación del método Multigrid para la resolución de la ecuación del calor presenta un mayor desafío.

El modelo implementado sigue la estructura de un ciclo en V (*V-Cycle*), que está compuesto de tres fases principales: una fase descendente de restricción, la resolución en la malla más gruesa, y una fase ascendente de prolongación y corrección. A continuación, se detalla cómo se ha implementado cada parte del algoritmo.

Suavizado (*Smoothing*)

El suavizado se realiza aplicando iteraciones de Jacobi, el cual se ha definido de manera idéntica a como se hizo para implementar el método de Jacobi explicado en la sección 6.2.3.

Esta implementación utiliza una convolución 2D con un *kernel* que promedia los valores de cuatro vecinos (arriba, abajo, izquierda y derecha).

```
1 def jacobi(self, Z, mask):
2     X = z
3     x_prev = x.clone()
```

```
    kernel = torch.tensor([[0.0, 0.25, 0.0],
                           [0.25, 0.0, 0.25],
                           [0.0, 0.25, 0.0]], dtype=self.datatype).view
                               (1, 1, 3, 3)
    i = torch.tensor(0, dtype=self.datatype)
    while torch.ne(i, self.nt):
        x_prev = x.clone()
        x_next = F.conv2d(x_prev, kernel, padding=1)
        x = x_next * mask
        i = torch.add(i, 1)
    return X
```

Listing 6.3: Código de Jacobi usado en Multigrid

Cálculo del Residuo

Tras el suavizado, se calcula el error residual de la solución.
El residuo representa una aproximación de cuánto se aleja la solución actual de la solución real y se calcula como la siguiente ecuación:

$$r = A\phi \tag{6.1}$$

Dónde $A\phi$ es el valor de la malla calculado para cada punto a partir de sus vecinos.
En la implementación de este trabajo, este cálculo $A\phi$ se realiza mediante una técnica que separa el tratamiento de los puntos interiores de la malla y los puntos de la frontera, que tienen condiciones fijas. El proceso es el siguiente:

- Un tensor *masked_input*: que contiene únicamente los valores de los puntos interiores (los bordes se ponen a cero).
- Un tensor *unmasked_input*: que contiene únicamente los valores de los puntos de la frontera (el interior se pone a cero).

Se calcula la influencia de cada uno de estos tensores por separado por medio de una convolución 2D con el mismo *kernel* que el método Jacobi.
Una vez calculadas ambas influencias, el residuo se termina de calcular con la suma de

estos dos resultados.

```
# ...
# Fase de bajada
for level in range(1, num_levels):
    # ... (operaciones de enmascaramiento)
    masked_output = F.conv2d(masked_input, kernel, padding=1)
    unmasked_output = F.conv2d(unmasked_input, kernel, padding=1)
    residual = masked_output + unmasked_output
# ...
```

Listing 6.4: Cálculo del residuo usado en Multigrid

Este resultado combina la influencia de los puntos interiores y de frontera, obteniendo así un residuo de todo el sistema, el cual será transferido posteriormente a la malla más gruesa de la parte de restricción.

Restricción (*Restriction*)

En la fase descendente, el algoritmo se desplaza de la malla más fina a la más gruesa. Para ello, se debe reducir el tamaño de la malla en cada nivel por medio de la restricción. La aproximación implementada se ha realizado sustituyendo una operación de promediado y submuestreo tradicional por una capa *Average Pooling*.

Esta capa calcula el promedio de los valores de una ventana, en el caso de este trabajo de tamaño 2x2, reduciéndola a un cuarto y, por consiguiente, el tamaño total de la malla también a un cuarto.

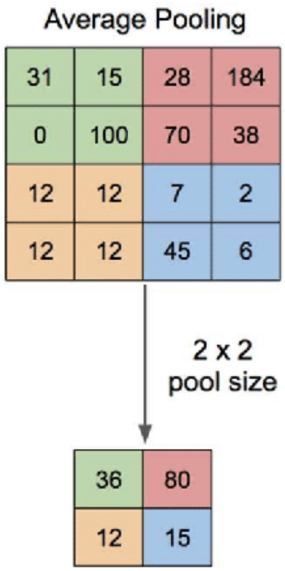


Figura 6.3: Ejemplo de Cómo funciona una Average Pooling 2D[6]

```
def restriction(self, residual):
    return nn.AvgPool2d(kernel_size=2)(residual)
```

Listing 6.5: Código de restricción usado en Multigrid

El residuo se pasa a la función de restricción definida y el resultado se añade a la lista de mallas.

```
# ... (dentro del bucle de la fase descendente)
coarse_residual = self.restriction(residual)
grids.append(coarse_residual)
```

Listing 6.6: Aplicación de restricción en Multigrid

Resolución en la malla gruesa

Una vez se ha alcanzado la malla más gruesa, el problema se resuelve directamente utilizando el método Jacobi:

```
coarse_solution = self.jacobi(coarse_solution, Mask9)
grids[-1] = coarse_solution
```

Listing 6.7: Resolución en Malla Gruesa

[6] https://paperswithcode.com/method/average-pooling

Con esto obtenemos una solución aproximada en esta malla, que al ser muy pequeña, se calcula muy rápido.

Prolongación y Corrección

En la fase ascendente, la solución de la malla gruesa se vuelve a agrandar a mallas más finas.

Esta prolongación se realiza mediante una interpolación bilineal con la función *interpolate* de PyTorch. Esta función calcula el valor de un punto basándose en los cuatro puntos más cercanos conocidos que lo rodean.

Esta función aumenta el tamaño de la malla, escalando la solución del nivel más grueso para que coincida con las dimensiones del siguiente nivel más fino.

```
def interpolate(self, f, target_size):
    return F.interpolate(f, size=target_size, mode='bilinear',
        align_corners=False)
```

Listing 6.8: Interpolación en el método Multigrid

A la solución interpolada se le debe aplicar una corrección mediante la suma del residuo calculado en la bajada en ese mismo nivel.

```
# Upward phase
for level in range(num_levels - 2, -1, -1):
    target_size = grids[level].shape[-2:]

    # Interpolate to finer grid
    fine_solution = self.interpolate(grids[level + 1], target_size=
        target_size)

    # Add correction to finer grid
    fine_solution = torch.add(fine_solution, grids[level])

    # ... (post-smoothing)
```

Listing 6.9: Corrección en el método Multigrid

Finalmente, se aplica un suavizado para refinar el resultado por medio de Jacobi, tal y como se ha explicado en la sección 6.2.4.

Manejo de condiciones de frontera

Las condiciones de frontera se manejan de forma similar a Jacobi, con el inconveniente de que ahora se necesita una máscara para cada nivel del método de Jacobi.
El problema de esto es que no se ha conseguido pasar todas las máscaras en una lista como parámetro de entrada, por lo que se han tenido que pasar individualmente cada una de ellas. Esto es necesario porque para crear las máscaras se usan operaciones de tipo *scatter*, que modifican valores concretos en un tensor y no son compatibles con la ANE.

6.3. Conversión a MLProgram

Una vez creado un modelo, tan solo es necesario convertirlo a MLProgram, un proceso ligeramente más complejo, pero que Apple documenta con bastante más detalle en sus tutoriales oficiales[7].
Se puede dividir este proceso en varias partes: obtener el TorchScript y convertir el modelo a MLProgram.

El primer paso para transformar un modelo a MLProgram es conseguir su grafo de PyTorch, algo que se puede lograr de dos maneras diferentes: por medio de TorchScript, tal y como se ha hecho, o por medio de *exportedProgram*.
El problema del segundo método es que no se ha logrado ejecutar correctamente en la ANE todas las operaciones, algo que no ha ocurrido al convertir los modelos con TorchScript. Esto puede deberse a que TorchScript es más antiguo y, por tanto, está más establecido en los flujos de conversión de Apple.

```
traced_model = torch.jit.trace(jacobiModel, (x, x_prev, mask))
```

Listing 6.10: Conversión de Jacobi Model a TorchScript

[7] https://apple.github.io/coremltools/docs-guides/source/convert-pytorch-workflow.html

TorchScript traza el grafo interno del modelo gracias a una entrada de prueba. Para realizarlo, es necesario crear dicha imagen y utilizar la operación de PyTorch correspondiente, tal y como se ve en el *listing* 6.10.

El siguiente paso es convertir de TorchScript a MLProgram, para tener ya nuestro modelo listo para ejecutar en la ANE.

```
npfloat = np.float16
ctfloat = ct.precision.FLOAT16
jacobi_from_trace = ct.convert(
    traced_model,
    inputs=[
        ct.TensorType(shape=x.shape, dtype=npfloat),
        ct.TensorType(shape=x_prev.shape, dtype=npfloat),
        ct.TensorType(shape=mask.shape, dtype=npfloat)
    ],
    outputs=[ct.TensorType(dtype=npfloat)],
    minimum_deployment_target=ct.target.macOS13,
    compute_precision=ctfloat
)
```

Listing 6.11: Conversión de Jacobi Model a MLProgram

En el *listing* 6.11, podemos observar un ejemplo de cómo se realiza la conversión a MLProgram para el método Jacobi a partir de su archivo TorchScript, teniendo que indicar todos los parámetros de entrada y cómo es la salida, indicando también qué tipo de datos son.

Además de esta información, se indica en qué precisión se van a realizar las operaciones del modelo (*ct.precision.FLOAT16* para FP16 o *ct.precision.FLOAT32* para FP32) y para qué sistemas Apple va a ser compatible. Esto último es importante, dado que por defecto el sistema lo hará para el mínimo posible para la máquina donde se está ejecutando [8]. Esto podría darnos problemas para ejecutar en la ANE en el futuro, por lo que se ha acabado indicando como mínimo MacOS 13.

Una vez realizadas estas operaciones, tendremos nuestro modelo convertido a MLProgram.

[8] https://apple.github.io/coremltools/source/coremltools.converters.convert.html

Para los modelos YOLOv11 se ha recurrido a otra forma alternativa al resto de algoritmos propios, dado que este modelo tiene capas dinámicas que no son sencillas de trazar. Estos modelos se deben convertir en TorchScript por medio de la librería propia de YOLO[9], que indica que hay que utilizar la operación *model.export(format="torchscript")* tras cargar el modelo. Una vez se tiene el archivo TorchScript, continuamos al igual que con los otros modelos, convirtiéndolo a CoreML mediante *ct.convert*.

Si queremos guardar nuestro modelo para poder usarlo más adelante o en otro ordenador, podemos hacerlo mediante el siguiente código:

```
jacobi_from_trace.save(f"jacobi_model.mlpackage")
```

Listing 6.12: Código para guardar un MLProgam

Con esto, tendremos nuestro modelo guardado y listo para ser cargado en cualquier momento.

6.4. Ejecución

Para ejecutar un modelo, el primer paso es cargarlo a través del *framework* de CoreML.

```
mlmodel = ct.models.MLModel(f"jacobi_model.mlpackage", compute_units=ct.
    ComputeUnit.ALL)
```

Listing 6.13: Código para cargar un MLProgam

Al cargar el modelo, es necesario especificar en qué dispositivos queremos ejecutarlo, siendo las opciones disponibles en CoreML [10] las siguientes:

- *ct.ComputeUnit.ALL*: la librería optimiza la ejecución entre los tres dispositivos: CPU, GPU y ANE.
- *ct.ComputeUnit.CPU_ONLY*: la librería ejecuta todo el modelo en CPU.

[9] docs.ultralytics.com/es/integrations/torchscript/#what-is-ultralytics-yolo11-model-export-to-torchscript

[10] apple.github.io/coremltools/index.html

- *ct.ComputeUnit.CPU_AND_GPU*: la librería optimiza la ejecución entre CPU y GPU.
- *ct.ComputeUnit.CPU_AND_ANE*: la librería optimiza la ejecución entre CPU y ANE.

Una vez el modelo está cargado, lo ejecutamos con el siguiente código:

```
result = mlmodel.predict(input_dict)
```

Listing 6.14: Código para ejecutar un MLProgam

Este código devolverá el resultado obtenido tras ejecutar el programa con los parámetros de entrada deseados.

Capítulo 7

Resultados

7.1. Resultados

En esta sección se analizan los diferentes resultados obtenidos en los ordenadores Mac M1 y Mac M4 Pro para los tres algoritmos seleccionados: GEMM, Jacobi y Multigrid. Asimismo, se analizará el rendimiento de los tres aceleradores que incorporan ambos Mac: CPU, GPU y ANE. Todos estos dispositivos se analizarán con datos en FP16 y FP32, tal como la librería MLTools de Mac permite en sus modelos, a excepción de la ANE que es incompatible con datos en FP32 y, en ese caso, los envía a la CPU tal y como se ha explicado previamente.

Por último, destacar también que la ANE permite realizar operaciones hasta un tamaño menor al que permiten la CPU y la GPU, lo que ocasiona un error si se intentan realizar cálculos con tamaños demasiado grandes.

Los resultados comprenden un análisis de rendimiento y eficiencia energética, comparando en gráficas el rendimiento en GFLOPS y la eficiencia energética en GFLOPS/Wattio, a excepción del algoritmo Multigrid, dado que no era factible calcular los GFLOPS para dicho algoritmo dada su complicada naturaleza multinivel, por lo que se mostrarán medidas de tiempo de ejecución en segundos y consumo medio del algoritmo en W.

7.2. Evaluación

Para evaluar los modelos, se han obtenido los siguientes datos:

- Tiempo medio de ejecución (s)
- Mejor tiempo de ejecución (s)
- Eficiencia media (GFLOPS)
- Mejor eficiencia (GFLOPS)
- Consumo energético medio total y por dispositivo (mW)
- Mejor consumo energético total y por dispositivo (mW)

Con estos datos se han evaluado los resultados, tal y como se puede apreciar en la sección 7.1.

El análisis se ha realizado con la mejor eficiencia y el consumo medio en el caso de GEMM

y Jacobi, y con el mejor tiempo de ejecución y el consumo medio en el caso de Multigrid, dada la complejidad que presenta el algoritmo.

En el caso de los modelos YOLO, se han utilizado las mismas métricas que en el caso de Multigrid, dado que estos son modelos de inteligencia artificial.
Los datos de energía se han calculado gracias a Powermetrics por medio del siguiente comando:

```
sudo powermetrics -i 100 --samplers cpu_power -a --hide-cpu-duty-cycle
--show-usage-summary --show-extra-power-info --show-process-energy
```

Este comando devuelve cada 100 milisegundos, entre otra información, el consumo total del ordenador y el consumo de cada dispositivo. La salida del comando se guarda en un archivo de texto que posteriormente es procesada para quedarnos únicamente con esta información.
Las medias de consumo se han tomado midiendo la totalidad del consumo durante las ejecuciones, que se divide entre el número de ejecuciones que se han realizado.

Estas pruebas se han automatizado gracias a *scripts* de Shell, guardando en archivos *csv* los resultados para que estos puedan ser almacenados después.

Para realizar cada una de las pruebas, se ha ejecutado repetidamente cada uno de los modelos (100 veces los modelos YOLO, Jacobi y Multigrid y 1000 veces el modelo GEMM), calculando con dichas ejecuciones los datos obtenidos que se pueden encontrar en este capítulo.

7.3. YOLOv3

Los resultados de tiempo medio obtenidos en la ejecución de un modelo YOLOv3, descargado de la página oficial de Apple [1], se muestran en la tabla 7.1. Para realizar la tabla, se ha ejecutado el modelo cien veces, de forma que el mejor resultado es el que se ha apuntado. Los datos ejecutados son de tipo FP16.

[1] https://developer.apple.com/machine-learning/models/

Como podemos observar, entre ambos ordenadores la ANE aporta una diferencia muy significativa, lo que valida su eficacia con respecto a la CPU y la GPU. Además, observamos cómo el M4 Pro es más rápido que el M1, lo que ya nos muestra el avance en *hardware* que existe entre ambos.

Ordenador	T. Ejecución CPU (ms)	T. Ejecución GPU (ms)	T. Ejecución ANE (ms)
M1	14.6	8.8	5.4
M4 Pro	6.9	6.3	3.2

Cuadro 7.1: Tabla de Comparación de Mejor Tiempo de Ejecución en YOLOv3

Por otro lado, se ha medido también el consumo energético medio durante las cien ejecuciones. Este se puede consultar en la tabla 7.2.

Podemos observar que en el M1 el consumo medio es menor en la GPU, estando la ANE en un punto medio entre la CPU y la GPU. En el M4 Pro, en cambio, la CPU y la GPU han aumentado mucho su consumo respecto al M1, al contrario que la ANE, que ha mantenido un consumo muy similar, lo que indica que este dispositivo es muy eficiente energéticamente en el M4 Pro, ya que es el de menor consumo. En el M1 sigue siendo una buena opción, aunque no tan destacada, ya que es el dispositivo que mejor tiempo consigue, con un consumo no demasiado grande respecto a los otros dispositivos.

Ordenador	C. Medio CPU (mW)	C. Medio GPU (mW)	C. Medio ANE (mW)
M1	6237.57	4013.45	5148.86
M4 Pro	7213.29	5523.06	5147.58

Cuadro 7.2: Tabla de Comparación de Consumo Medio en YOLOv3

Con estos datos podemos concluir que la ANE efectivamente mejora respecto a los otros dos dispositivos tanto en tiempo como en el caso del M4 Pro, en consumo.

Estas pruebas nos ayudan a caracterizar este trabajo, ya que podemos ver que el acelerador de Apple logra una mejora sustancial con respecto a modelos de aprendizaje automático, con lo que merece la pena si se pueden obtener mejores resultados también en algoritmos de propósito general.

7.4. YOLOv11

En esta sección se analizarán todas las versiones de YOLOv11 oficiales en ambos dispositivos Mac.

Para obtener los resultados, se ha analizado el mejor tiempo obtenido en cien ejecuciones de cada modelo, así como su consumo medio.

En primer lugar, en la tabla 7.3 podemos encontrar los datos sobre el tiempo de ejecución de cada uno de los modelos YOLOv11.

Ordenador	Modelo	Precisión	T. Ejecución CPU (s)	T. Ejecución GPU (s)	T. Ejecución ANE (s)
M1	YOLOv11x	FP16	0.6125	0.1210	0.0327
		FP32	0.8153	0.1383	-
	YOLOv11l	FP16	0.3157	0.0581	0.0163
		FP32	0.3777	0.0635	-
	YOLOv11m	FP16	0.2760	0.0473	0.0139
		FP32	0.3351	0.0518	-
	YOLOv11s	FP16	0.0999	0.0181	0.0055
		FP32	0.1041	0.0188	-
	YOLOv11n	FP16	0.0438	0.0082	0.0033
		FP32	0.0412	0.0085	-
M4 Pro	YOLOv11x	FP16	0.0697	0.0390	0.0222
		FP32	0.1800	0.0416	-
	YOLOv11l	FP16	0.0463	0.0193	0.0109
		FP32	0.0861	0.0205	-
	YOLOv11m	FP16	0.0348	0.0155	0.0092
		FP32	0.0709	0.0161	-
	YOLOv11s	FP16	0.0190	0.0063	0.0031
		FP32	0.0282	0.0066	-
	YOLOv11n	FP16	0.0095	0.0032	0.0016
		FP32	0.0132	0.0032	-

Cuadro 7.3: Comparación de Tiempos de Ejecución para todos los modelos YOLOv11.

En la tabla observamos como, cuanto más pequeño es el modelo, menos apreciable es la diferencia de tiempo tanto entre FP16 y FP32, y entre ambos equipos Mac.

A pesar de ello, en todos los modelos se mantiene una estructura general donde la ANE es el dispositivo más rápido, con una diferencia enorme especialmente en el modelo x. No obstante, la diferencia es mucho más notoria en el M1 que en el M4 Pro, pasando de tener un *Speedup* con respecto a la CPU de 1873 % a uno de 314 % en la versión x, mientras que la GPU pasa de un 500 % a un 179 %. Esto muestra cómo la ANE ha mejorado menos que los otros dos dispositivos (CPU, GPU) de un ordenador al otro, ya que en todos los modelos ocurre una situación similar.

Esa diferencia hace que en el M4 Pro presente un tiempo muy competitivo en los tres

dispositivos, mientras que la CPU del M1 y, en menor medida, la GPU obtienen tiempos considerablemente peores.

La diferencia de tiempo entre tipos de datos es muy baja en ambos dispositivos, por lo que no parece que este factor sea determinante más allá de si queremos usar la ANE en términos de rendimiento.

Por otro lado, en la tabla 7.4 encontramos los resultados para el consumo medio de todos los modelos YOLOv11.

Ordenador	Modelo	Precisión	C. Medio CPU (mW)	C. Medio GPU (mW)	C. Medio ANE (mW)
M1	YOLOv11x	FP16	5682.03	8353.86	5884.40
		FP32	5647.42	8060.65	-
	YOLOv11l	FP16	5693.44	7399.11	5706.76
		FP32	5913.78	7600.92	-
	YOLOv11m	FP16	5694.54	7379.86	5735.29
		FP32	5660.16	7692.03	-
	YOLOv11s	FP16	5949.13	6135.68	5276.77
		FP32	6111.55	6703.50	-
	YOLOv11n	FP16	6078.40	5380.35	5669.67
		FP32	5982.78	5924.13	-
M4 Pro	YOLOv11x	FP16	7657.90	11095.23	8452.22
		FP32	7332.38	12715.25	-
	YOLOv11l	FP16	7684.67	10871.15	8769.38
		FP32	7879.98	16868.31	-
	YOLOv11m	FP16	7846.11	12405.33	9644.23
		FP32	7892.85	13474.40	-
	YOLOv11s	FP16	7740.40	10444.30	6993.56
		FP32	8294.97	14246.92	-
	YOLOv11n	FP16	8414.38	10578.93	10415.67
		FP32	9313.65	10339.00	-

Cuadro 7.4: Comparación de Consumo Medio (mW) para todos los modelos YOLOv11.

Estos resultados muestran siempre un consumo superior de la GPU en ambos dispositivos y tipos de datos, mientras que la ANE tiene un consumo similar o algo superior al de la CPU.

Este factor es clave, pues indica que la GPU es menos eficiente energéticamente que la ANE por una amplia diferencia, ya que la ANE tiene mejores resultados de tiempo.

Los tres dispositivos parecen haber aumentado una cantidad similar de consumo de un ordenador a otro, lo que podría significar que lo que más ha aumentado es el consumo general del ordenador por motivos ajenos a la ejecución de los modelos. Por último, el consumo entre los dos tipos de datos no varía demasiado, con lo que podemos concluir que la elección de usar uno u otro no debe depender tampoco de la eficiencia energética.

7.5. GEMM

En la multiplicación de matrices encontramos resultados distintos a lo esperado debido al uso del dispositivo AMX con el que cuentan los Macs por parte de la CPU. Esto es especialmente notable en el modelo M4 Pro, dadas sus mejoras.

Para comprobar esto, se ha ejecutado código en C++ que hace uso del *framework* Accelerate y los modelos creados en este trabajo con el objetivo de observar si se obtiene el mismo rendimiento. Los datos pueden consultarse en la tabla 7.5.

Tamaño de Matriz (k)	**Dispositivo**	**Rend. Accelerate (1 hilo)** (GFLOPS/s)	**Rend. Accelerate (2 hilos)** (GFLOPS/s)	**Rend. CoreML** (GFLOPS/s)
256	M1	328.29	-	369.39
	M4 Pro	564.33	462.87	611.90
512	M1	745.05	-	715.77
	M4 Pro	1120.62	819.23	1376.41
1024	M1	980.57	-	606.63
	M4 Pro	1257.37	1378.69	1356.71
2048	M1	843.57	-	811.93
	M4 Pro	1600.02	1599.76	1599.15
4096	M1	923.31	-	1032.54
	M4 Pro	1584.81	2546.39	1528.89
8192	M1	870.99	-	835.55
	M4 Pro	1542.35	3016.10	1609.28
12288	M1	-	-	-
	M4 Pro	1536.88	3041.88	1648.19
14336	M1	-	-	-
	M4 Pro	1534.89	3039.61	1589.69
16384	M1	-	-	-
	M4 Pro	1529.61	3069.36	1591.63

Cuadro 7.5: Comparativa de Rendimiento con GEMM en CPU M1 y M4 Pro (GFLOPS/s)

Los datos muestran tres columnas con los datos de CoreML y Accelerate con uno y dos hilos, ya que al disponer el chip M4 Pro dos AMX, puede usar dos hilos para aumentar la velocidad.

Los datos claramente indican un rendimiento prácticamente idéntico en todos los casos entre CoreML y Accelerate con un hilo en ambos ordenadores, por lo que podemos afirmar con seguridad que CoreML utiliza el dispositivo AMX, aunque no explota todo su potencial en el caso del M4 Pro.

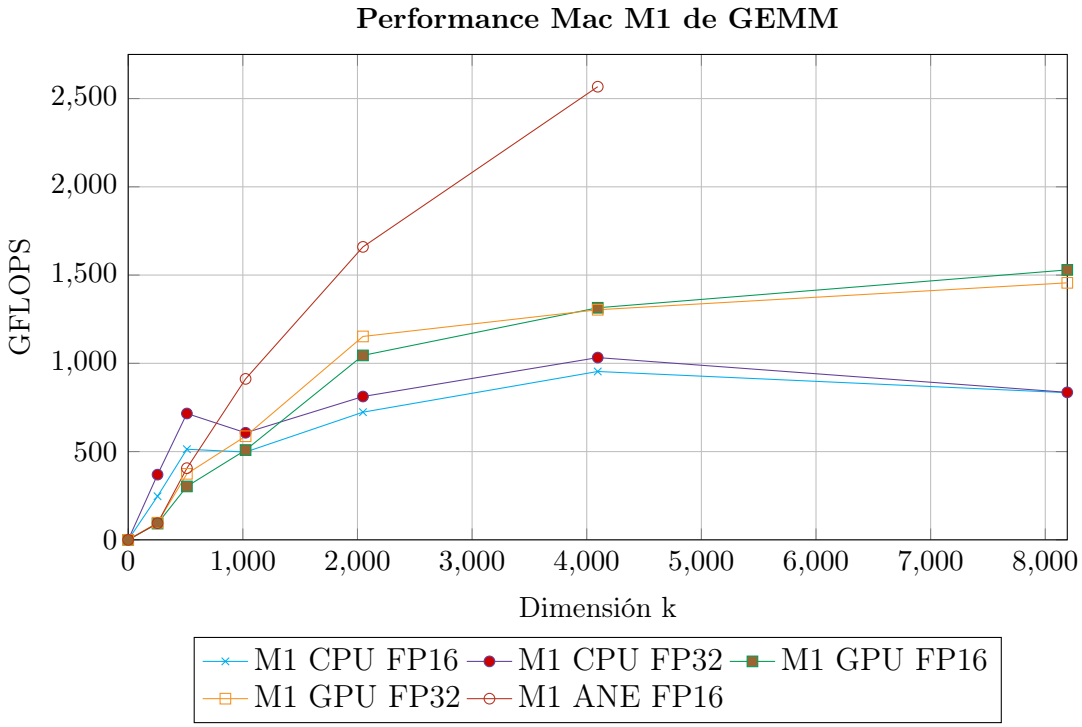


Figura 7.1: Performance Mac M1 de GEMM

En la figura 7.1 podemos apreciar el rendimiento que tienen en el Mac M1 los tres dispositivos que lo componen.
Destaca notablemente la ANE, especialmente para tamaños de matriz iguales o superiores a 1024, a pesar de no estar pensada concretamente para este tipo de operaciones.

Inicialmente, la CPU logra un desempeño mejor que la GPU, lo cual es debido al uso del AMX, demostrado anteriormente.
Por otro lado, la CPU en FP32 muestra un rendimiento superior al de FP16, lo que podría indicar una compatibilidad nativa con datos FP32 que requiere una conversión previa para FP16.
Por último, la GPU no supera a la CPU en rendimiento FP32 hasta el tamaño 4096, lo que también podría deberse a consideraciones de compatibilidad de datos

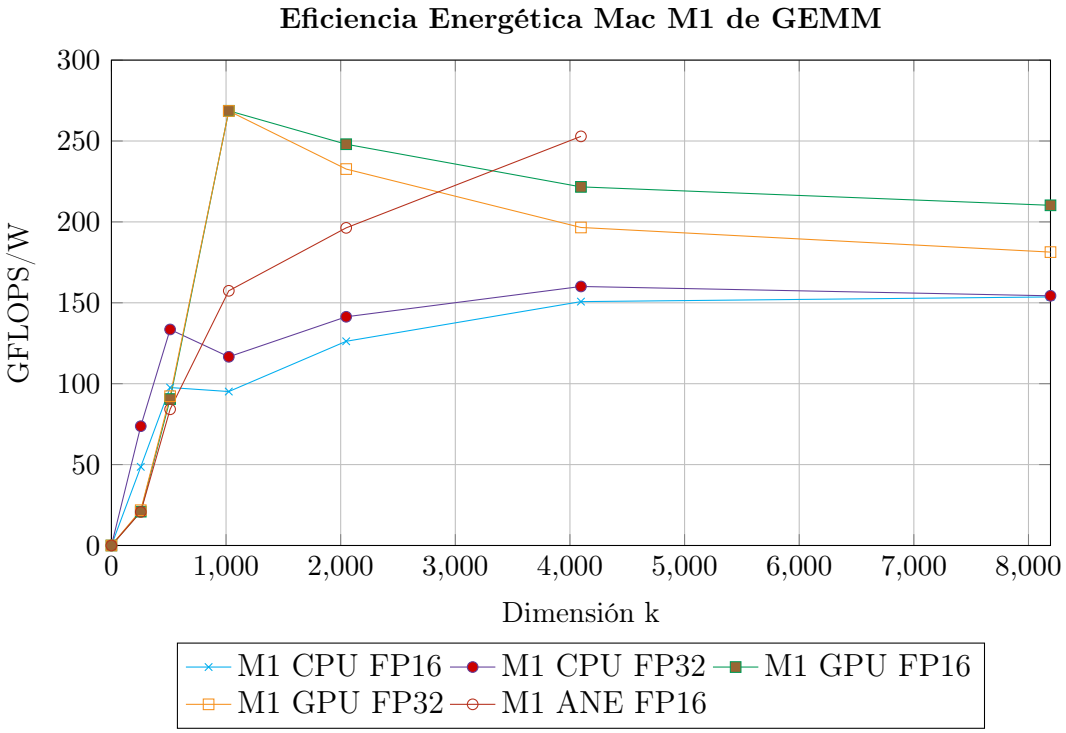


Figura 7.2: Eficiencia Energética Mac M1 de GEMM

En términos de eficiencia energética en el Mac M1, disponible en la figura 7.2, la ANE es más eficiente para el mayor tamaño que puede alcanzar, 4096. La GPU, generalmente más eficiente en FP16, es el dispositivo con mayor eficiencia energética en la mayoría de casos. Esto indica un menor gasto energético respecto a FP32 a pesar de que este en ocasiones logre un mejor rendimiento.

Por su parte, en la CPU, el AMX parece ser el responsable de que los datos en FP32 mantengan una mayor eficiencia que en FP16, probablemente debido a su posible compatibilidad nativa con FP32.

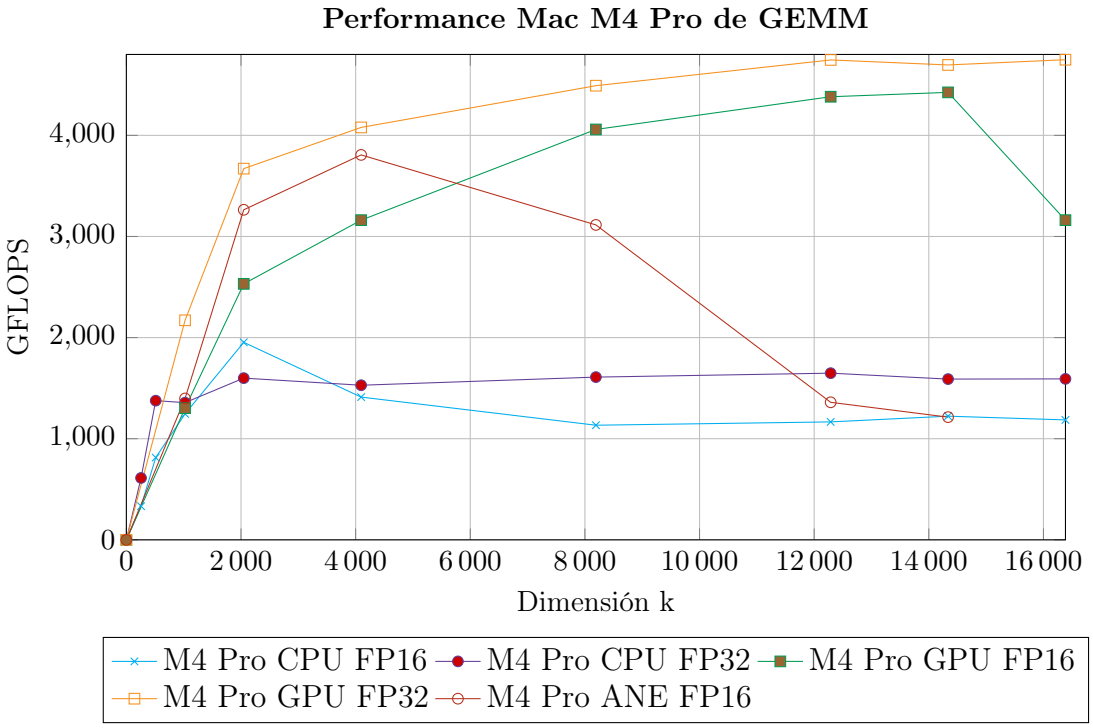


Figura 7.3: Performance Mac M4 Pro de GEMM

En el Mac M4 Pro (Figura 7.3), la GPU en FP32 lidera consistentemente en rendimiento a lo largo de toda la gráfica.

La mejora en el rendimiento de la GPU en el M4 Pro es significativamente mayor en comparación con el M1, superando las mejoras observadas en la ANE y la CPU. La ANE, por su parte, ofrece un rendimiento competitivo, situándose entre la GPU FP32 y la GPU FP16.

Este dispositivo además baja en rendimiento a partir de un tamaño de 4096 debido a que las capas utilizadas en inteligencia artificial son más pequeñas, por lo que para tamaños de malla grande el dispositivo debe operar sin poder ejecutar todo el problema a la vez al no caber entero en el acelerador.

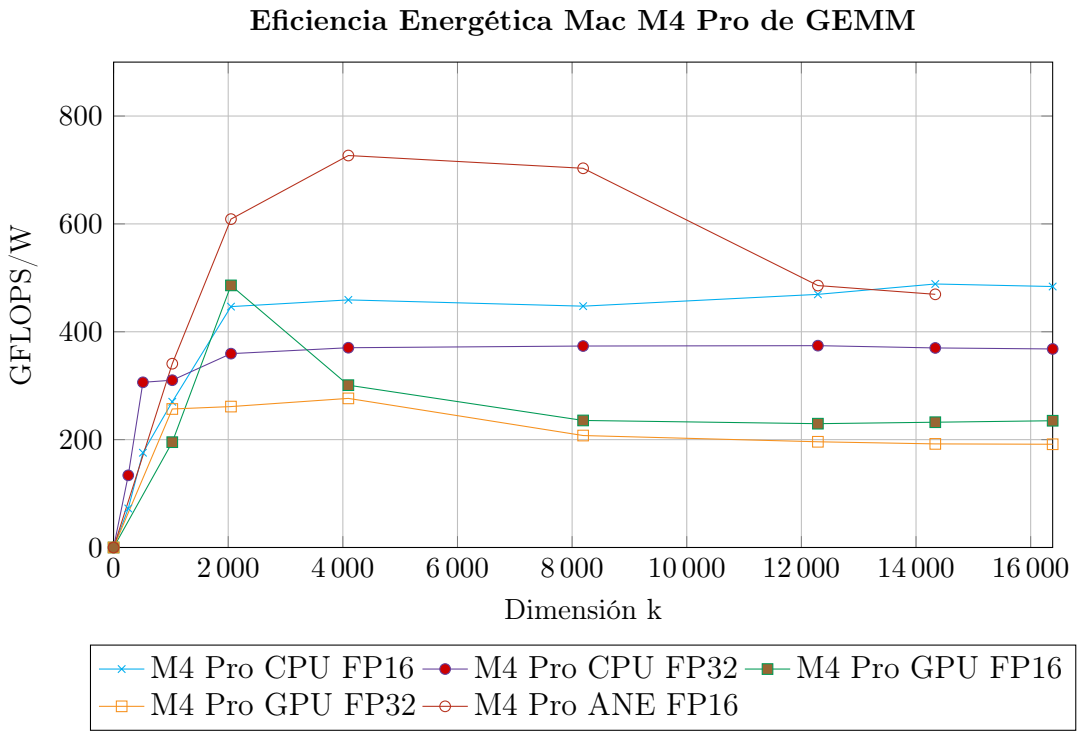


Figura 7.4: Eficiencia Energética Mac M4 Pro de GEMM

En cuanto a la eficiencia energética en el M4 Pro, los datos de la figura 7.4 denotan que la ANE se destaca como la mejor opción en la práctica totalidad de los escenarios. Aunque la GPU ha avanzado enormemente en cuanto a rendimiento, también ha aumentado su consumo, lo que otorga a la ANE una ventaja significativa.

La GPU en FP32, en algunos casos, supera a la GPU en FP16, pero por lo general, la GPU presenta una eficiencia energética inferior no solo a la ANE, sino también a la CPU. Esto es relevante para aplicaciones con limitaciones de potencia, como dispositivos *IoT*.

Por último, la CPU en FP16 supera a la CPU en FP32 a partir de un tamaño determinado en el M4 Pro, a diferencia del M1, lo que podría atribuirse al uso de ARM SME.

Size	SpeedUp	FP16	FP32
1024	CPU	2.5294	2.1875
	GPU	2.625	3.7
	ANE	1.6	–
2048	CPU	2.7045	1.9813
	GPU	2.4118	3.1702
	ANE	1.9623	–
4096	CPU	1.4795	1.4805
	GPU	2.4023	3.1276
	ANE	1.4820	–

Cuadro 7.6: SpeedUp de M1 a M4 Pro en GEMM para distintos tamaños y precisiones

En la tabla 7.6 se ha calculado el *SpeedUp* delM4 Pro con respecto al M1 de la multiplicación de matrices.

Con el *SpeedUp* se observa una mayor mejora en GPU de un ordenador a otro. Además de esto, la CPU parece tener también una mejora más pronunciada que la ANE en tamaños intermedios, lo que apunta a un avance más limitado en la ANE entre ambos equipos, tal y como ya concluían las pruebas realizadas en YOLO.

En resumen, los resultados de la multiplicación de matrices indican que la ANE posee un gran potencial para ejecutar algoritmos de propósito general, especialmente en términos de eficiencia energética.

Además, se ha demostrado que CoreML utiliza el dispositivo AMX en ejecuciones en CPU, aumentando en gran medida su eficiencia en operaciones matriciales. Pese a ello, CoreML no logra alcanzar todo el rendimiento que podría en el M4 Pro, al no ejecutar en el AMX con dos hilos de ejecución.

Por último, GEMM parece indicar, al igual que YOLO, un avance más pronunciado en CPU y, sobre todo, GPU entre un chip y otro, que la ANE.

7.6. Resultados Jacobi

El método de Jacobi es un algoritmo iterativo utilizado para resolver sistemas de ecuaciones diferenciales, en este estudio aplicado a la ecuación del calor. Los resultados muestran un comportamiento diferente al de la multiplicación de matrices, especialmente en relación con el rendimiento de la ANE y la GPU.

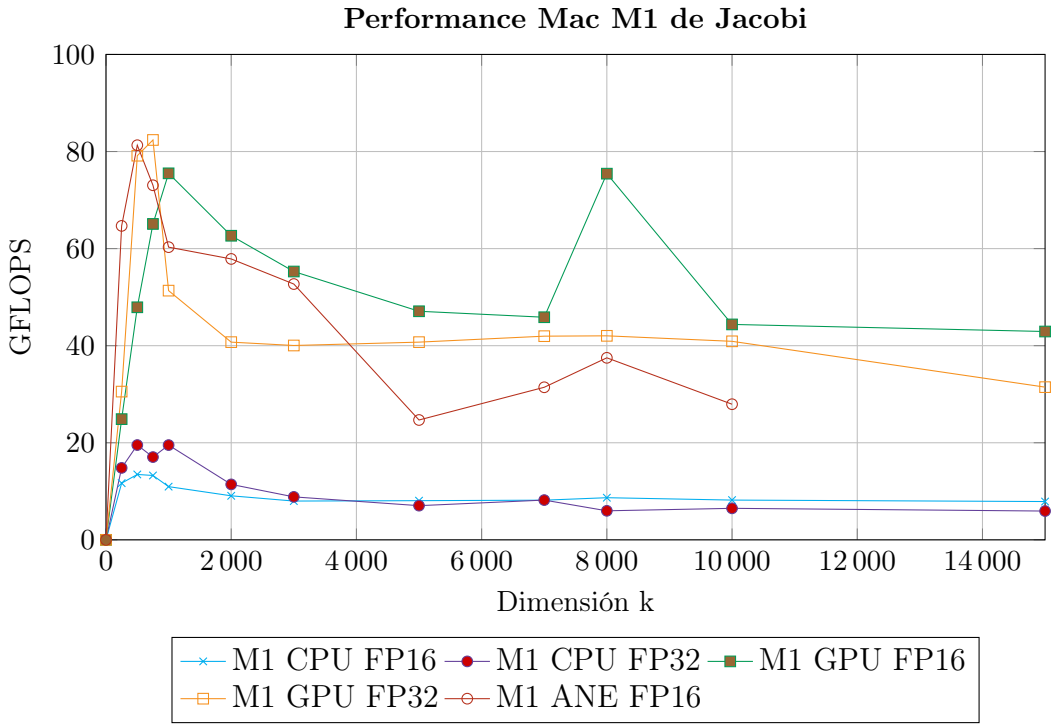


Figura 7.5: Performance Mac M1 de Jacobi

A diferencia de GEMM, en la figura 7.5 podemos observar que la ANE no siempre es el dispositivo que mejor rendimiento logra en el Mac M1, a excepción de tamaños "pequeños"(inferiores a 1000). A partir de ese punto, la GPU con datos FP16, es el dispositivo que mejor rendimiento obtiene. La ANE es también superada por la GPU con datos FP32 a partir de un tamaño de 3000.

La CPU, por su parte, muestra un rendimiento similar con ambos tipos de datos, indicando que la eficiencia es prácticamente la misma independientemente de esto.

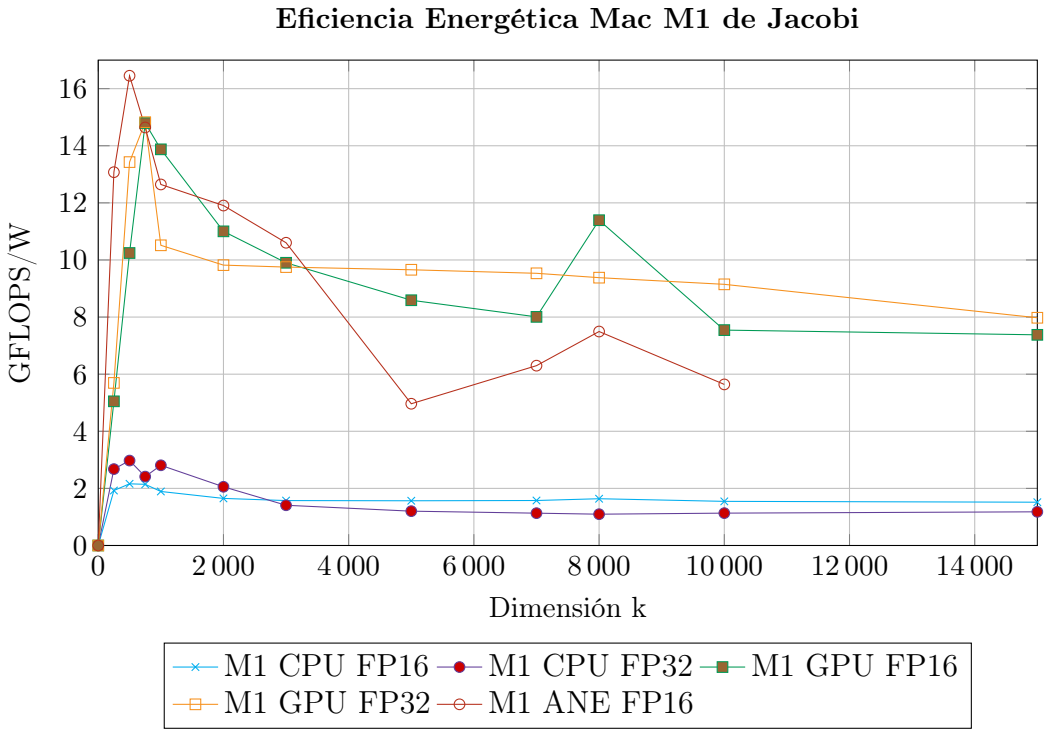


Figura 7.6: Eficiencia Energética Mac M1 de Jacobi

La eficiencia energética del algoritmo de Jacobi en el Mac M1 se encuentra en la figura 7.6.

Esta gráfica revela que la ANE destaca, al igual que en rendimiento, en tamaños pequeños. Sin embargo, es superada por la GPU con ambos tipos de datos a medida que aumenta el tamaño de la malla.

Es notable que los datos FP32 en GPU muestran una mayor eficiencia energética en casi todos los puntos, lo que probablemente se deba a la compatibilidad nativa del dispositivo. La CPU mantiene un patrón similar al rendimiento, con FP32 por encima de FP16 hasta cierto valor, lo que podría deberse a compatibilidad nativa de datos.

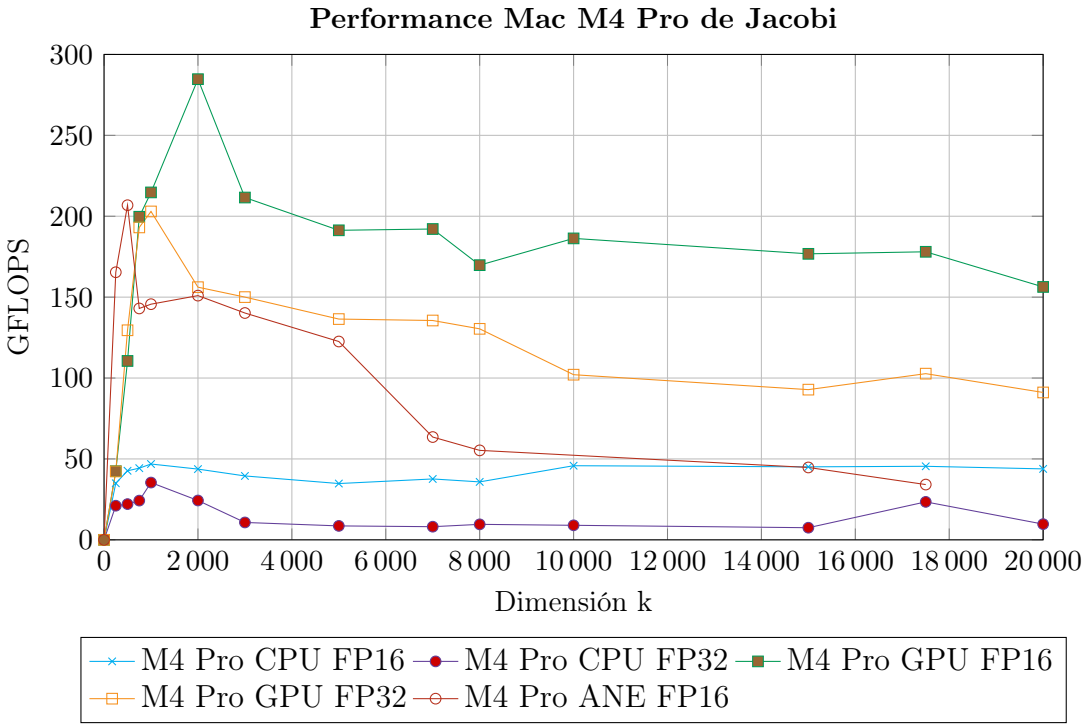


Figura 7.7: Performance Mac M4 Pro de Jacobi

El rendimiento en el Mac M4 Pro (disponible en la figura 7.7) alcanza valores mucho más altos que los del M1.

La ANE continúa mostrando un mayor desempeño para tamaños pequeños de nuevo, aunque la GPU la supera en otros tamaños, especialmente con datos FP16, y con una ventaja mucho más pronunciada que en el M1. Para tamaños grandes, la ANE incluso puede rendir peor que la CPU con datos FP16.

La CPU, a diferencia de GEMM, muestra mejores resultados con datos FP16 sobre FP32 a lo largo de todo el rango de tamaños. Esto se debe a la mejora general de la CPU del M1 al M4 Pro.

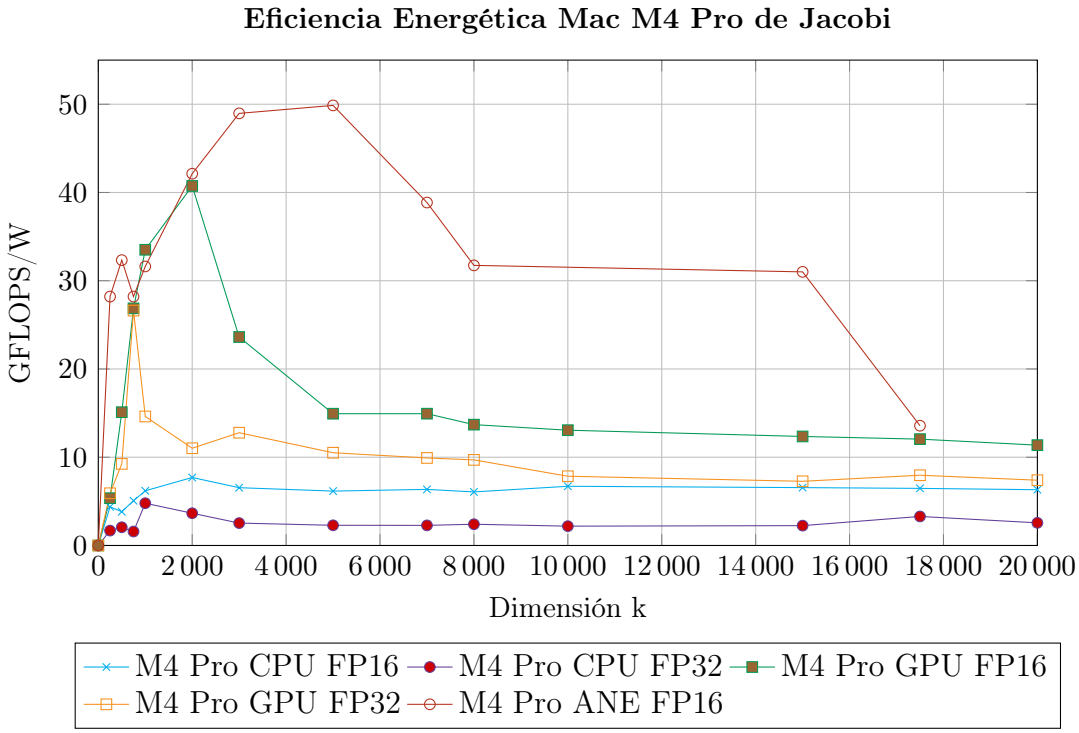


Figura 7.8: Eficiencia Energética Mac M4 Pro de Jacobi

La figura 7.8 muestra la eficiencia energética del método Jacobi en el dispositivo M4 Pro. En esta gráfica es muy favorable para la ANE en la práctica totalidad de los casos. Esto se atribuye en parte al elevado consumo energético de las GPU, lo que convierte a la ANE en el dispositivo más eficiente para este algoritmo.

Aunque la GPU en FP16 puede igualar a la ANE en algunos tamaños (como 2000), no puede competir en la mayoría de los casos. Los datos en FP16 para la GPU obtienen mejores resultados que los FP32, lo cual es esperable. De manera similar, la CPU con FP16 es más eficiente que con FP32.

Size	SpeedUp	FP16	FP32
250	CPU	2.9444	1.4
	GPU	1.6667	1.3333
	ANE	2.5	–
500	CPU	3.1525	1.1327
	GPU	2.2609	1.6842
	ANE	2.5833	–
750	CPU	3.3386	1.4224
	GPU	3.0714	2.3448
	ANE	1.9744	–
1000	CPU	4.2864	1.8156
	GPU	2.8085	3.9759
	ANE	2.4058	–
2000	CPU	4.8197	2.1276
	GPU	4.5571	3.8359
	ANE	2.6075	–
3000	CPU	4.9434	1.2125
	GPU	3.8306	3.7450
	ANE	2.6604	–
5000	CPU	4.3127	1.2242
	GPU	4.06197	3.3499
	ANE	4.9667	–
7000	CPU	4.6068	1.2439
	GPU	4.1886	3.2302
	ANE	2.0214	–
8000	CPU	4.1278	1.6049
	GPU	2.2718	3.1023
	ANE	1.4742	–

Cuadro 7.7: SpeedUp de M1 y M4 Pro en Jacobi para distintos tamaños y precisiones

Se ha analizado el *SpeedUp* del M4 Pro con respecto al M1 en la tabla 7.7.
Estos datos corroboran lo observado en GEMM y YOLO: la ANE ha sido el dispositivo que menos ha mejorado entre generaciones. La CPU es el dispositivo que más ha mejorado proporcionalmente, aunque la GPU obtiene un *SpeedUp* similar en muchos casos.

Los resultados de Jacobi reafirman que la ANE puede ser utilizada en algoritmos de propósito general para obtener mejores resultados, especialmente en el Mac M1 para tamaños pequeños, tanto en eficiencia energética como en rendimiento. En el Mac M4 Pro, el rendimiento es mejor para valores pequeños y, además, la eficiencia energética es superior en casi todos los casos, lo que confirma el potencial de los aceleradores neuronales en algoritmos de propósito general.

7.7. Resultados Multigrid

La resolución Multigrid tiene el problema añadido de que calcular su rendimiento ya no es una cuestión trivial en este algoritmo debido a su naturaleza iterativa y multinivel, donde las operaciones varían significativamente entre los tamaños de malla. Por este motivo, en esta sección analizaremos el tiempo medio de ejecución (en segundos) y su consumo medio (en vatios).

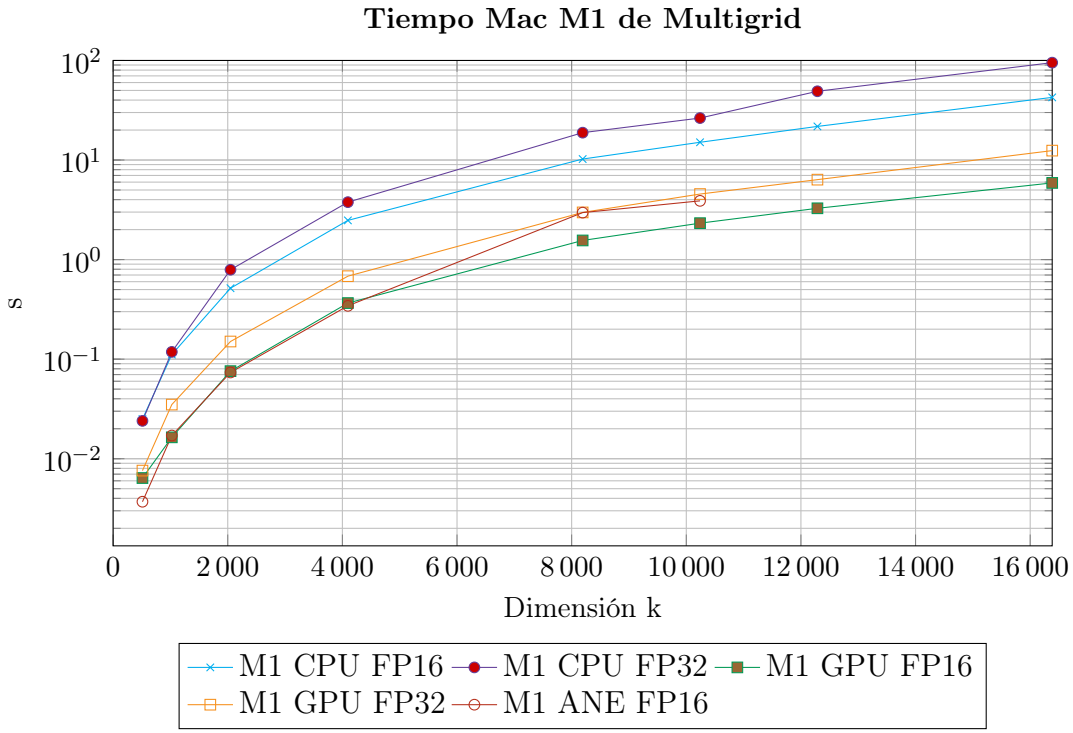


Figura 7.9: Tiempo Mac M1 de Multigrid

En la figura 7.9 se muestra el tiempo medio de ejecución en el Mac M1, usando una escala logarítmica en el eje Y para poder realizar mejor el análisis dada la gran diferencia de tiempo entre algunos dispositivos.

Los resultados indican que, a partir de cierto tamaño (4096), la GPU con datos FP16 tarda menos en ejecutar que la ANE. En cuanto a la CPU, es el dispositivo más lento con mucha diferencia y es más rápida con datos FP16 que con FP32.

Analizando con métricas, la GPU FP16 llega a ser un 90 % más rápida que la ANE en el mejor de los casos, pero están igualadas la mayor parte del tiempo. A su vez, la ANE es más rápida que la GPU FP32 en todos los casos, siendo casi aproximadamente dos veces más rápida en el valor de malla más pequeño, y hasta un 17 % más rápida en el

más grande. Estos resultados son un buen indicio para la ANE, ya que si su consumo energético es lo suficientemente eficiente, podría ser una buena alternativa a la GPU como ocurre con los otros algoritmos analizados.

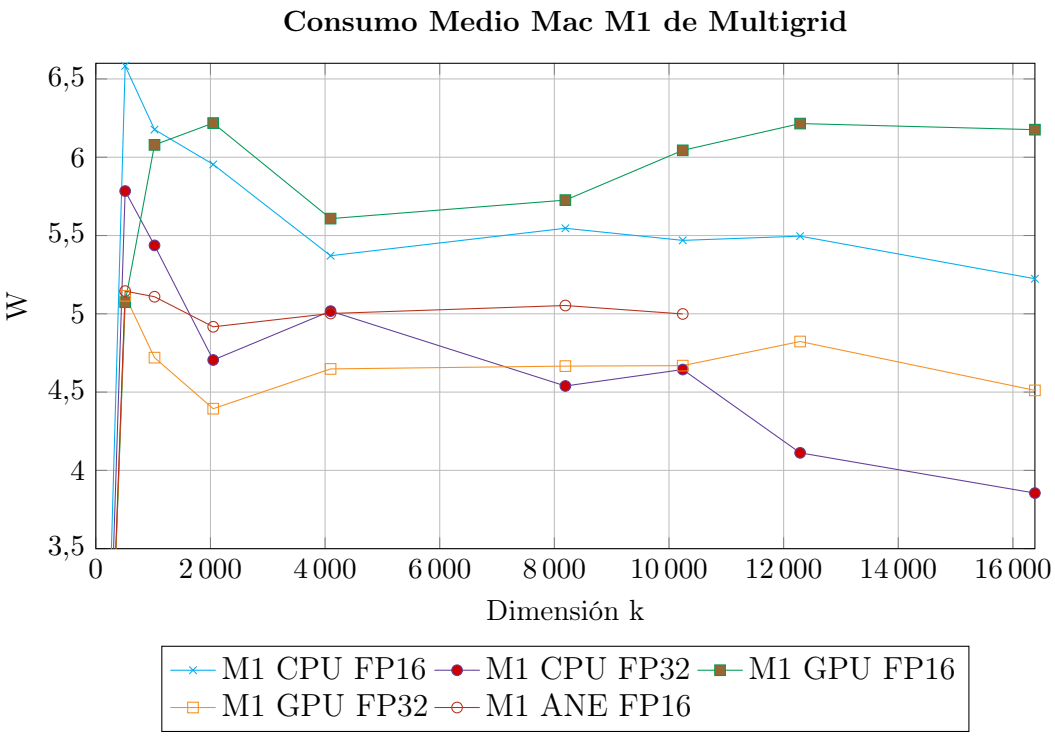


Figura 7.10: Consumo Medio Mac M1 de Multigrid

Los datos sobre consumo medio plasmados en la figura 7.10, reflejan un mayor consumo en general en GPU con datos FP16, con diferencias de hasta aproximadamente 1W. Sorprende el enorme consumo de la CPU con datos FP16 y el bajo consumo de la GPU con datos FP32. Esta última parece indicarse como más eficiente que la ANE en este dispositivo Mac, probablemente debido a que la naturaleza del algoritmo Multigrid no es la más indicada para la ANE al consumir gran parte del tiempo de ejecución en la malla más fina (la malla más grande), donde la GPU tendría una mayor ventaja.

Con estos datos podemos concluir que en el M1 la ANE consume de media un 14 % menos de energía que la GPU FP16, mientras que la GPU FP32 consume aproximadamente un 7.3 % menos de energía que la ANE.

La conclusión a la que se llega con este análisis es que en el M1 la ANE es más efi-

ciente energéticamente que la GPU FP16 para tamaños pequeños. Sin embargo, para tamaños grandes, aunque la GPU FP16 es más rápida, su mayor consumo implica que la ventaja de eficiencia de la ANE no es tan clara.

Además, la GPU32 consume menos que la ANE en todos los tamaños, con lo que, pese a ser esta más rápida, todo parece indicar que no es más eficiente energéticamente.

En resumen, la ANE se posiciona bien frente a la GPU FP16 en eficiencia para ciertos tamaños, pero la GPU FP32 destaca por su bajo consumo, lo que la convierte en una fuerte contendiente en eficiencia energética, pese a su menor velocidad.

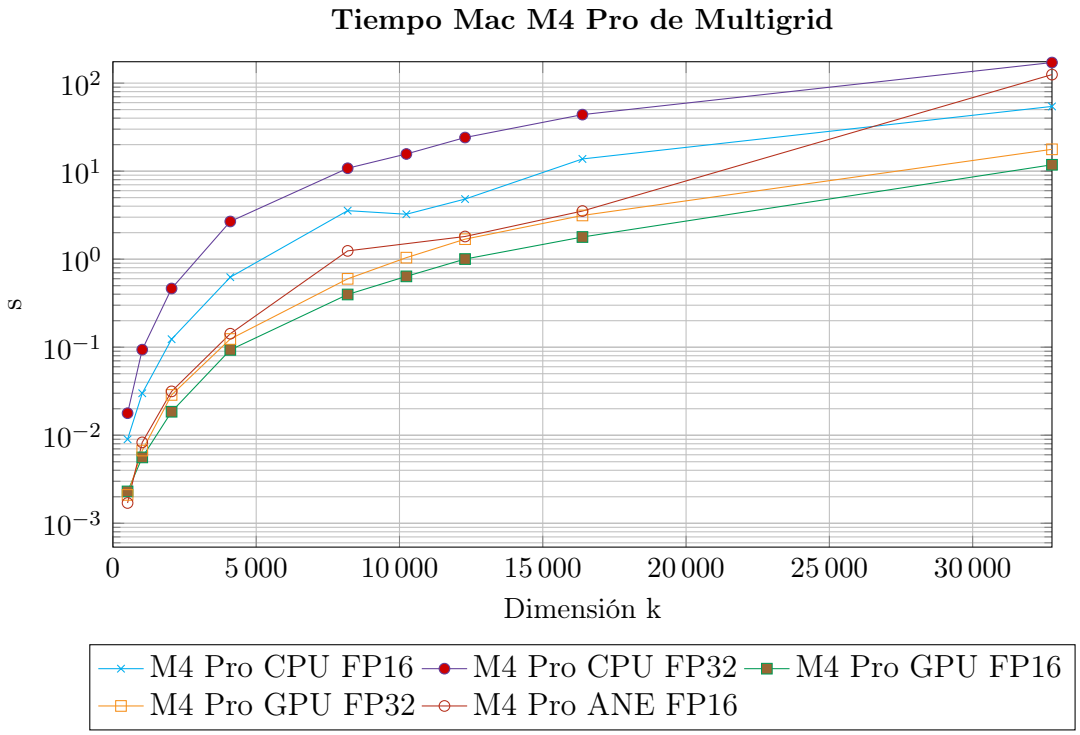


Figura 7.11: Tiempo Mac M4 Pro de Multigrid

La figura 7.11, al igual que la gráfica del Mac M1, presenta una escala logarítmica para una mejor visualización de los datos.

Se observa que la GPU con FP32 supera en tiempo a la ANE, lo que representa una desventaja para la ANE en comparación con el M1 en este aspecto.

En los demás dispositivos, los resultados son los esperados, con la CPU aún más rezagada respecto a la GPU y la ANE. La GPU en FP16, por su parte, en la mayoría de los casos ya no presenta tanta ventaja con respecto a los datos FP32 y la ANE.

En este punto, la ANE sigue siendo relevante en datos muy pequeños, siendo un 35 % más rápida que la GPU con FP16 con mallas de tamaño 512x512. Sin embargo, cuanto

mayor es el tamaño, más eficiencia saca la GPU FP16 al acelerador neuronal.

La GPU FP32 es también consistentemente más rápida que la ANE en el M4 Pro, siendo aproximadamente 1.24 veces más rápida para mallas de 512x512, y esta diferencia se acentúa en tamaños mayores.

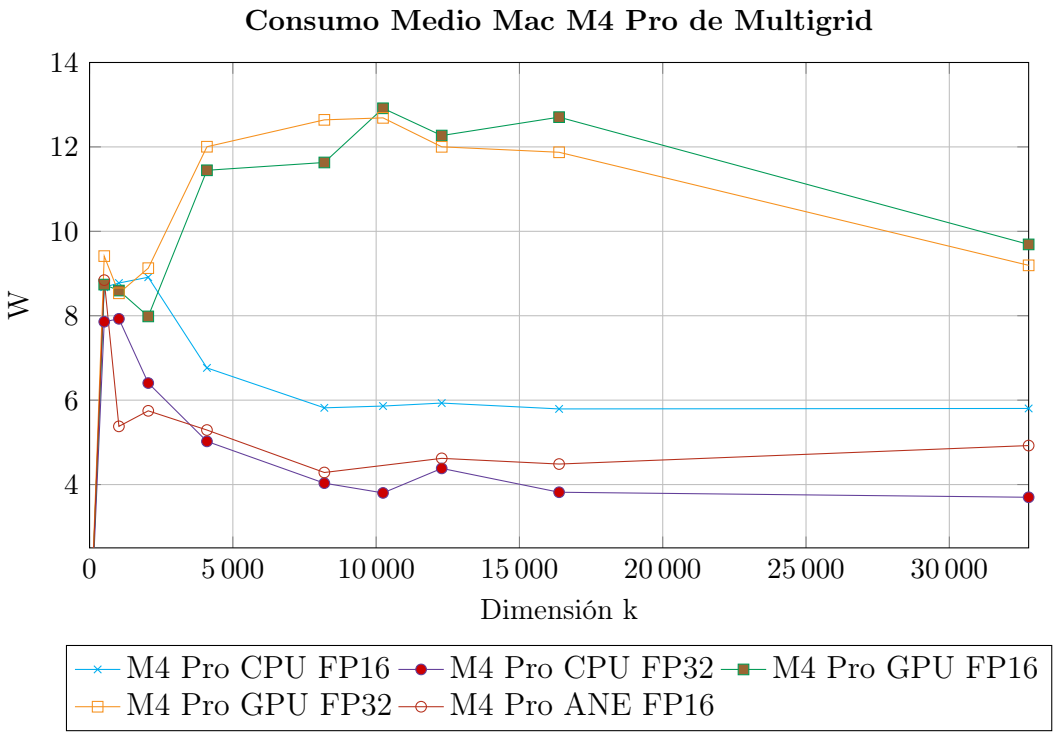


Figura 7.12: Consumo Medio Mac M4 Pro de Multigrid

Por último, analizando el consumo medio disponible en la figura 7.12, los datos son muy favorables para la ANE en comparación con la GPU en ambos tipos de datos, de forma similar a lo observado en el método Jacobi y la multiplicación de matrices. El bajo consumo de la ANE, incluso menor que el de la CPU con datos FP16, indica una eficiencia energética muy buena por parte de la ANE frente a la GPU. La GPU, que es el dispositivo con mayor rendimiento, en este caso presenta un consumo bastante similar en ambos tipos de datos y muy por encima de la CPU y la ANE.

Estos datos favorables para la ANE tienen valores como, por ejemplo, en 1024x1024. En este tamaño la ANE consume aproximadamente un 37 % menos que la GPU FP16. Esta tendencia de menor consumo de la ANE se mantiene para casi todos los tamaños de malla. Con los datos GPU FP32 encontramos unos datos muy similares.

Estos datos nos permiten posicionar a la ANE como un dispositivo relevante en tema de eficiencia energética a pesar del declive en su tiempo de ejecución.

Size	SpeedUp	FP16	FP32
512	CPU	2.78	1.35
	GPU	2.78	3.62
	ANE	2.18	–
1024	CPU	3.71	1.26
	GPU	2.93	5.22
	ANE	2.06	–
2048	CPU	4.19	1.70
	GPU	4.10	5.24
	ANE	2.33	–
4096	CPU	3.96	1.41
	GPU	3.92	5.50
	ANE	2.41	–
8192	CPU	2.87	1.75
	GPU	3.92	4.98
	ANE	2.39	–

Cuadro 7.8: *SpeedUp* de M4 Pro respecto a M1 en Multigrid para distintos tamaños y precisiones

El *SpeedUp* del chip M4 Pro con respecto al M1 se encuentra en la tabla 7.8. En ella, observamos la misma tendencia que en la multiplicación de matrices y Jacobi, con una menor mejora en la ANE que en la GPU y la CPU.
Esto reafirma que la ANE ha sufrido una mejora menos pronunciada del M1 al M4 Pro que los otros dos dispositivos.

Los datos obtenidos en Multigrid, aunque no favorables para la ANE en el Mac M1, indican una eficiencia energética muy buena en el M4 Pro, lo que posiciona a la ANE como un dispositivo muy factible para situaciones donde el consumo energético sea vital. Cabe mencionar que los resultados son acordes a lo visualizado en la sección 7.6, ya que la ANE es eficiente principalmente en mallas de pequeño tamaño; por ello, es lógico que la GPU sea más rápida en Multigrid, dado que alrededor de la mitad del tiempo de ejecución se consume en las mallas más grandes (mallas finas).

Capítulo 8

Conclusiones

8.1. Conclusiones

Analizando los resultados obtenidos, se concluye que el acelerador neuronal de Apple (ANE) puede alcanzar un buen rendimiento en algoritmos de propósito general, como la multiplicación de matrices en el chip M1.
Sin embargo, en otros algoritmos como Jacobi, Multigrid e incluso la misma multiplicación de matrices en el chip M4 Pro, el rendimiento es inferior al de la GPU, excepto para valores de malla pequeños.
En algoritmos de inteligencia artificial, para los que está diseñado, el desempeño de la ANE es superior al de los demás dispositivos, especialmente en el Mac M1 y en modelos grandes, como el YOLOv11x.

Por otro lado, la ANE presenta una excelente eficiencia energética tanto en el Mac Mini M1 como en el M4 Pro, especialmente en este último. Esta eficiencia podría ser especialmente relevante en ámbitos de computación en el borde (*Edge Computing* en inglés) o en otras aplicaciones donde el consumo energético sea un factor crítico.
Sin embargo, la ANE presenta limitaciones de tamaño notables que provocan errores de ejecución si se le envían datos de gran tamaño, lo cual es un aspecto muy negativo de la misma si se quiere tratar con este tipo de datos.

Asimismo, se ha observado que, del M1 al M4 Pro, la GPU y la CPU han experimentado mejoras significativas en rendimiento, mientras que la ANE ha mostrado un avance limitado.
Esto se refleja en los cálculos de *SpeedUp*, que indican un desarrollo general menor para la ANE en comparación con los otros dispositivos.
La CPU cobra especial relevancia en este aspecto, ya que se ha demostrado que en la multiplicación de matrices ha mejorado su rendimiento gracias a las mejoras en el coprocesador AMX. Aunque CoreML limita su uso a un solo hilo, las instrucciones ARM SME contribuyen a una reducción del tiempo de ejecución.
Las mejoras de los tres dispositivos vienen acompañadas de un mayor consumo energético, particularmente en la GPU, lo que posiciona a la ANE como líder en eficiencia energética.

En conclusión, la ANE demuestra ser un dispositivo competente en el uso de algorit-

mos de propósito general. Su desempeño es particularmente notable en el procesamiento de datos de pequeño tamaño, ya que en ellos logra un rendimiento y una eficiencia energética especialmente destacada. Esta ventaja se acentúa aún más en el chip M4 Pro, dado que compensa el alto consumo energético de la GPU, optimizando así el balance entre rendimiento y eficiencia.

8.2. Trabajo Futuro

Como trabajo futuro, se han planteado dos ideas: dividir el método Multigrid en varios modelos para ejecutar en diferentes dispositivos y probar a ejecutar en la ANE algoritmos más largos y complejos, como podrían ser herramientas reales de *benchmarking*.
Por supuesto, otros algoritmos podrían ser utilizados, pero las dos ideas mencionadas anteriormente y que se desarrollarán a continuación son las más relevantes debido a la capa extra de complejidad que representan.

8.2.1. División en Modelos de Multigrid

El método Multigrid ha demostrado un rendimiento especialmente destacado en tamaños de malla pequeños. Dado que aproximadamente el 50 % del tiempo se consume en la malla más fina (la de mayor tamaño), y que la ANE ha mostrado mejor desempeño en tamaños de malla más pequeños, resulta conveniente dividir el algoritmo entre la GPU y la ANE, asignando las partes según su rendimiento óptimo.

El principal desafío que encontramos es que, una vez que el modelo se ejecuta en un modo (GPU o ANE), no es posible cambiar a otro modo durante la ejecución. Por ello, proponemos como trabajo futuro dividir el modelo en tres submodelos:

- Un primer modelo que se ejecute en la GPU y que realice la "bajada" del algoritmo hasta un determinado nivel de malla.
- Un segundo modelo que se ejecute en la ANE, encargado de completar la "bajada" restante, realizar las iteraciones en la malla más gruesa, y ejecutar la "subida" hasta el mismo nivel en que comenzó el primer modelo.
- Un tercer modelo que se ejecute nuevamente en la GPU y complete la parte restante

del algoritmo.

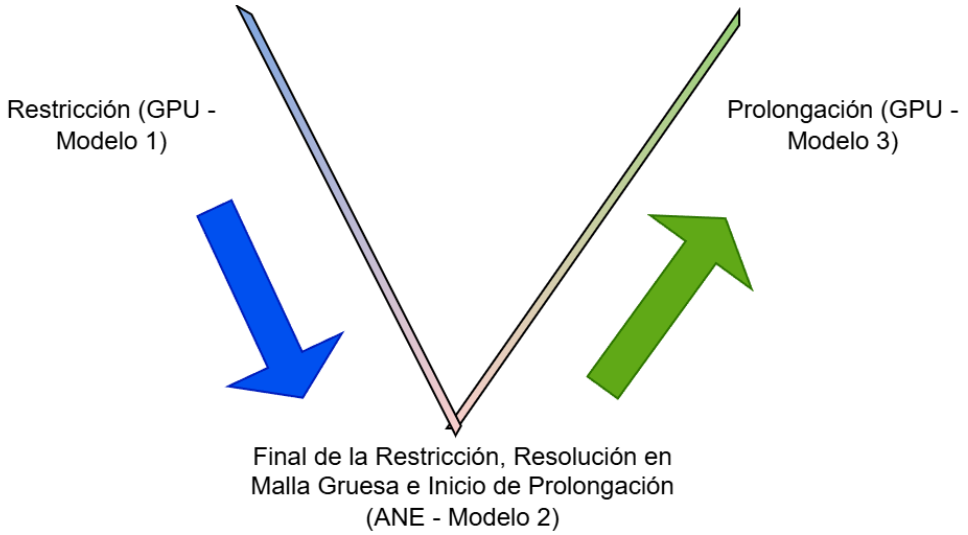


Figura 8.1: Propuesta de División de Multigrid en Tres Modelos

La figura 8.1 ilustra esta propuesta.

Para implementar esta división, se podría partir del modelo actual de Multigrid (disponible en el repositorio de GitHub mencionado en el apartado 4.3.8), y crear tres submodelos independientes. Cada submodelo tendría almacenada una variable que indique la altura de malla donde finaliza su ejecución y transferiría el flujo de trabajo al siguiente modelo a través de su salida. Así, cada uno ejecutaría únicamente el fragmento del proceso *forward* que le corresponde.

8.2.2. Herramientas de *Benchmarking*

El uso de algoritmos más complejos en la ANE sería la forma de trabajo futuro más sencilla de pensar, aunque en lugar de utilizar otros algoritmos famosos, adaptar una herramienta de prueba utilizada en entornos reales podría ser una opción más interesante y que arroje resultados más concisos.

Existen varias herramientas utilizadas en *benchmarking* de *hardware* como AutoDock, capaz de simular moléculas para descubrimientos farmacéuticos[1] o LAMMPS, un simu-

[1] https://github.com/ccsb-scripps/AutoDock-GPU

lador de dinámicas moleculares[2].

A pesar de lo interesante que sería la implementación en la ANE de todas estas herramientas, la herramienta que se ha elegido como principal candidata es MiniBUDE[3] [4] [5].

MiniBUDE es una mini aplicación derivada de la famosa herramienta de *benchmarking* para HPC *Bristol University Docking Engine* (BUDE). La herramienta simula distintos procesos moleculares, principalmente relacionados con la proteína NDM-1, para evaluar el *hardware*.

Se ha elegido MiniBUDE dado que es una herramienta ampliamente usada en algoritmos de paralelismo para tecnologías muy utilizadas en la actualidad [42] [43], como CUDA y OpenMP, lo que la convierte en una excelente candidata para comparar la ANE con otros dispositivos de alto rendimiento actuales.

[2] https://www.lammps.org/
[3] https://github.com/UoB-HPC/miniBUDE
[4] https://openbenchmarking.org/test/pts/minibude?
[5] https://uob-hpc.github.io/assets/ISC-2021-miniBUDE-slides.pdf

Capítulo 9

Conclusions

9.1. Conclusions

Analysing the results obtained, it is concluded that the Apple Neural Engine (ANE) can achieve good performance in general-purpose algorithms, such as matrix multiplication on the M1 chip.
However, in other algorithms such as Jacobi, Multigrid, and even matrix multiplication itself on the M4 Pro chip, the performance is lower than that of the GPU, except for small grid sizes.
In artificial intelligence algorithms, for which it is designed, the ANE's performance is superior to that of other devices, especially on the Mac M1 and with large models, such as YOLOv11x.

On the other hand, the ANE shows excellent energy efficiency on both the Mac Mini M1 and the M4 Pro, especially on the latter. This efficiency could be particularly relevant in Edge Computing or in other applications where energy consumption is a critical factor.
However, the ANE has notable size limitations that cause runtime errors if large amounts of data are sent to it, which is a very negative aspect if one wants to deal with this type of data.

Likewise, it has been observed that, from the M1 to the M4 Pro, the GPU and CPU have experienced significant performance improvements, while the ANE has shown limited advancement.
This is reflected in the SpeedUp calculations, which indicate a lower overall development for the ANE compared to the other devices.
The CPU takes on special relevance in this aspect, as it has been shown that its performance in matrix multiplication has improved thanks to enhancements in the AMX coprocessor. Although CoreML limits its use to a single thread, the ARM SME instructions contribute to a reduction in execution time.
The improvements in all three devices are accompanied by higher energy consumption, particularly in the GPU, which positions the ANE as the leader in energy efficiency.

In conclusion, the ANE proves to be a competent device for use in general-purpose al-

gorithms. Its performance is particularly notable in processing small-sized data, as it is here that it achieves especially outstanding performance and energy efficiency. This advantage is even more pronounced on the M4 Pro chip, as it compensates for the high energy consumption of the GPU, thus optimizing the balance between performance and efficiency.

9.2. Future Work

As future work, two ideas have been proposed: dividing the multigrid method into several models to run on different devices, and testing the execution of longer and more complex algorithms on the ANE, such as real benchmarking tools.
Of course, other algorithms could be used, but the two ideas described above and developed below are the most relevant due to the extra layer of complexity they represent.

9.2.1. Division of Multigrid into Models

The Multigrid method has shown particularly outstanding performance with small grid sizes. Since approximately 50 % of the time is spent on the finest grid (the largest one), and the ANE has shown better performance with smaller grid sizes, it is convenient to divide the algorithm between the GPU and the ANE, assigning parts according to their optimal performance.

The main challenge we face is that once the model is running on one device (GPU or ANE), it is not possible to switch to another during execution. Therefore, we propose as future work to divide the model into three sub-models:

- A first model that runs on the GPU and performs the "descent"phase of the algorithm down to a certain grid level.
- A second model that runs on the ANE, responsible for completing the remaining "descent", performing the iterations on the coarsest grid, and executing the .ªscent"phase up to the same level where the first model began.
- A third model that runs again on the GPU and completes the remaining part of the algorithm.

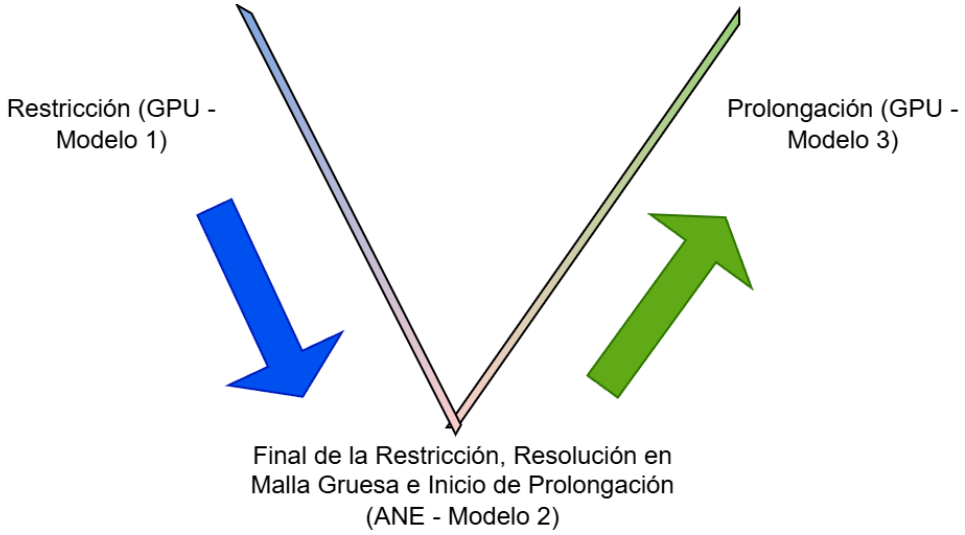


Figura 9.1: Proposal for Dividing Multigrid into Three Models

Figure 9.1 illustrates this proposal.

To implement this division, one could split the current multigrid model (available in the GitHub repository mentioned in section 4.3.8), and create three independent sub-models. Each sub-model would store a variable indicating the grid level where its execution ends and would transfer the workflow to the next model through its output. Thus, each would only execute its corresponding fragment of the forward process.

9.2.2. Benchmarking Tools

Using more complex algorithms on the ANE would be the most straightforward line of future work, although instead of using other famous algorithms, adapting a testing tool used in real-world environments could be a more interesting option that would yield more concise results.

There are several tools used in hardware benchmarking such as AutoDock, capable of simulating molecules for pharmaceutical discoveries[1], or LAMMPS, a molecular dynamics simulator[2].

Despite the interest in implementing all these tools on the ANE, the tool chosen as the

[1] https://github.com/ccsb-scripps/AutoDock-GPU

[2] https://www.lammps.org/

main candidate is MiniBUDE[3] [4] [5].

MiniBUDE is a mini-application derived from the famous HPC benchmarking tool Bristol University Docking Engine (BUDE). The tool simulates various molecular processes, mainly related to the NDM-1 protein, to evaluate hardware.

MiniBUDE has been chosen because it is a widely used tool in parallelism algorithms for technologies in high demand today [42] [43], such as CUDA and OpenMP, making it an excellent candidate for comparing the ANE with other current high-performance devices.

[3] https://github.com/UoB-HPC/miniBUDE

[4] https://openbenchmarking.org/test/pts/minibude?

[5] https://uob-hpc.github.io/assets/ISC-2021-miniBUDE-slides.pdf

Parte C

Índices

Índice de figuras

Índice de cuadros

Parte D

Bibliografía

Bibliografía

[1] S. Bavikadi, A. Dhavlle, A. Ganguly, A. Haridass, H. Hendy, C. Merkel, V. J. Reddi, P. R. Sutradhar, A. Joseph, and S. M. Pudukotai Dinakarrao, "A survey on machine learning accelerators and evolutionary hardware platforms," *IEEE Design Test*, vol. 39, no. 3, pp. 91–116, 2022.

[2] D. Reed, D. Gannon, and J. Dongarra, "Reinventing high performance computing: Challenges and opportunities," 2022.

[3] S. Susnjara and I. Smalley, "What is high-performance computing (hpc)?." https://www.ibm.com/think/topics/hpc.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[5] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Ding, H. Xin, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Wang, J. Chen, J. Yuan, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, S. Ye, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Zhao, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu,

X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025.

[6] P. Powell and I. Smalley, "What is a central processing unit (cpu)?." https://www.ibm.com/think/topics/central-processing-unit.

[7] W. Stallings, *Computer Organization and Architecture: Designing for Performance*. USA: Prentice Hall Press, 8th ed., 2009.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 6th ed., 2017.

[9] N. Nassif, A. O. Munch, C. L. Molnar, G. Pasdast, S. V. Lyer, Z. Yang, O. Mendoza, M. Huddart, S. Venkataraman, S. Kandula, R. Marom, A. M. Kern, B. Bowhill, D. R. Mulvihill, S. Nimmagadda, V. Kalidindi, J. Krause, M. M. Haq, R. Sharma, and K. Duda, "Sapphire rapids: The next-generation intel xeon scalable processor," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, pp. 44–46, 2022.

[10] F. Wilkinson and S. McIntosh-Smith, "An initial evaluation of arm's scalable matrix extension," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 135–140, Nov. 2022.

[11] E. Engheim, "The secret apple m1 coprocessor." https://medium.com/swlh/apples-m1-secret-coprocessor-6599492fc1e1.

[12] S. Remke and A. Breuer, "Hello sme! generating fast matrix multiplication kernels using the scalable matrix extension," 2024.

[13] P. Hübner, A. Hu, I. Peng, and S. Markidis, "Apple vs. oranges: Evaluating the apple silicon m-series socs for hpc performance and efficiency," 2025.

[14] M. Flinders, S. Susnjara, and I. Smalley, "What is a graphics processing unit (gpu)?." https://www.ibm.com/think/topics/gpu.

[15] "Cpu o gpu: ¿cuál es la diferencia?." https://www.intel.la/content/www/xl/es/products/docs/processors/cpu-vs-gpu.html.

[16] H. Reyes, "Cpu vs gpu for model training: Understanding the differences." https://medium.com/@reyes83/cpu-vs-gpu-for-model-training-understanding-the-differences-a82c8d016293.

[17] E. BUBER and B. DIRI, "Performance analysis and cpu vs gpu comparison for deep learning," in *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, pp. 1–6, 2018.

[18] R. Jayanth, N. Gupta, and V. Prasanna, "Benchmarking edge ai platforms for high-performance ml inference," 2024.

[19] "What is an ai accelerator?." https://www.ibm.com/think/topics/ai-accelerator.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2nd ed., 2001.

[21] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadia-

ni, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," 2017.

[23] W. Aspray, "The intel 4004 microprocessor: what constituted invention?," *IEEE Annals of the History of Computing*, vol. 19, no. 3, pp. 4–15, 1997.

[24] N. Zhang, "Moore's law is dead, long live moore's law!," 2022.

[25] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[26] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[27] C. McClanahan, "History and evolution of gpu architecture a paper survey," 2011.

[28] W. J. Dally, S. W. Keckler, and D. B. Kirk, "Evolution of the graphics processing unit (gpu)," *IEEE Micro*, vol. 41, no. 6, pp. 42–51, 2021.

[29] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[31] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, 2012.

[32] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramirez, "Energy efficient hpc on embedded socs: Optimization techniques for mali gpu," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 123–132, 2014.

[33] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with commodity cpus: Are mobile socs ready for hpc?," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2013.

[34] M. Hollemans, *Core ML Survival Guide: More than you ever wanted to know about mlmodel files and the Core ML and Vision APIs.* Leanpub, 2020. Published on 2020-10-09. © 2018–2020 M.I. Hollemans.

[35] C. Kenyon and C. Capano, "Apple silicon performance in scientific computing," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–10, 2022.

[36] K. Struniawski, A. Konopka, and R. Kozera, *Exploring Apple Silicon's Potential from Simulation and Optimization Perspective*, pp. 35–42. 06 2024.

[37] L. Gebraad and A. Fichtner, "Seamless gpu acceleration for c++-based physics with the metal shading language on apple's m series unified chips," *Seismological Research Letters*, Feb. 2023.

[38] D. Kasperek, M. Podpora, and A. Kawala-Sterniuk, "Comparison of the usability of apple m1 processors for various machine learning tasks," *Sensors*, vol. 22, no. 20, 2022.

[39] S. Imambi, K. B. Prakash, and G. Kanagachidambaresan, "Pytorch," in *Programming with TensorFlow*, pp. 87–104, Springer, 2021.

[40] E. Stevens, L. Antiga, and T. Viehmann, *Deep learning with PyTorch.* Manning Publications, 2020.

[41] T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[42] A. Poenaru, W.-C. Lin, and S. McIntosh-Smith, "A performance analysis of modern parallel programming models using a compute-bound application," in *High Performance Computing* (B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, eds.), (Cham), pp. 332–350, Springer International Publishing, 2021.

[43] W.-C. Lin and S. McIntosh-Smith, "Comparing julia to performance portable parallel programming models for hpc," in *2021 International Workshop on Performance*

Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 94–105, 2021.

Álvaro Corrochano López

acorroch@ucm.es

Junio 2025

Últ. actualización 7 de julio de 2025