Evaluación del SoC M1 de Apple para cómputo matricial
Evaluating Apple M1 SoC for Matrix Computation

Trabajo de Fin de Grado
Curso 2021–2022

Autor
Javier Piernagorda Olivé

Director
Luis Piñuel Moreno
Rafael Rodríguez Sánchez

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

Evaluación del SoC M1 de Apple para cómputo matricial
Evaluating Apple M1 SoC for Matrix Computation

Trabajo de Fin de Grado en Ingeniería de Computadores
Departamento de Arquitectura de Computadores y Automática

Autor
Javier Piernagorda Olivé

Director
Luis Piñuel Moreno
Rafael Rodríguez Sánchez

Convocatoria: *Junio 2022*
Calificación:

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

30 de Mayo de 2022

Dedicatoria

A mis padres.

Agradecimientos

A mis coordinadores Rafael Rodríguez y Luis Piñuel por su ayuda, enseñanza y disponibilidad en todo momento, y a la Universidad Complutense de Madrid por ceder temporalmente el Mac Mini M1 que ha hecho posible la realización de este proyecto.

Resumen

Evaluación del SoC M1 de Apple para cómputo matricial

El cómputo matricial está presente en multitud de ámbitos de la vida cotidiana. Una de las aplicaciones donde está siendo más utilizado en el Machine Learning, donde el tratamiento de volúmenes de datos grandes es frecuente.

En este proyecto se evalúan las ventajas que presentan las distintas librerías BLAS tanto a la hora de crear redes neuronales como a la hora de realizar operaciones algebraicas básicas, que son la base del Machine Learning.

Se han analizado diversos aspectos de estas librerías como el rendimiento, el consumo y la eficiencia de las librerías Accelerate, OpenBLAS y BLIS y se han obtenido conclusiones acerca de su viabilidad para el uso en aplicaciones de Machine Learning. Además, se analiza el rendimiento de la librería Accelerate, que es propia de Apple, con la librería MetalPerfomanceShaders, que es también propiedad de la misma compañía y permite el acceso a la GPU.

Las pruebas se han ejecutado sobre el modelo más actual de Mac Mini, que incluye el procesador Apple M1 de última generación.

Palabras clave

Machine Learning, DeepLearning, BLAS, Accelerate, Apple M1, Apple, Metal, macOS

Abstract

Evaluating Apple M1 SoC for Matrix Computation

Matrix computation is present in several daily life areas. One of the areas where its being more used is Machine Learning, since big-volume data processing is frequent.

In this project, several advantages of the BLAS libraries are analized when it comes to creating neural networks and performing basic algebraic operations, which are the base of Machine Learning.

Multiple aspects such as performance, energy consumption and efficiency have been analized from the libraries Accelerate, OpenBLAS and BLIS and conclusions have been obtained regarding their viability when it comes to using them in Machine Learning applications. Also, the performance of the Accelerate Apple-propietary library has been analized against the performance obtained with the also-Apple-propietary MetalPerformanceShaders library, which allows the GPU usage.

These tests have been performed on the latest Apple Mac Mini, which includes the latest Apple M1 chip.

Keywords

Machine Learning, DeepLearning, BLAS, Accelerate, Apple M1, Apple, Metal, macOS

Índice

1. Introducción **1**
1.1. Motivación . . . 1
1.2. Objetivos . . . 2
1.3. Estructura del Documento . . . 2

2. Estado del Arte **4**
2.1. Apple M1 . . . 4
2.1.1. Contexto . . . 4
2.1.2. Arquitectura . . . 5
2.2. Librerías BLAS . . . 11
2.2.1. Accelerate . . . 11
2.2.2. OpenBLAS . . . 12
2.2.3. BLIS . . . 13

3. Descripción del Trabajo **16**
3.1. Documentación y Puesta en Marcha . . . 16
3.2. Uso de las Distintas Librerías . . . 16
3.3. Acceso a los Cores . . . 17
3.4. División del Trabajo en Cores . . . 18
3.5. Uso de los Microkernels de BLIS . . . 19
3.6. Uso de la GPU . . . 20

4. Análisis de Resultados **22**
4.1. Rendimiento y Consumo de las Librerías . . . 22
4.1.1. Accelerate . . . 23
4.1.2. OpenBLAS . . . 26
4.1.3. BLIS . . . 30
4.2. Comparativa General . . . 33
4.3. Microkernels de BLIS . . . 37
4.4. Utilización de Icestorm y Firestorm Conjuntamente . . . 40
4.5. GPU . . . 42

5. Conclusiones y Trabajo Futuro **44**
5.1. Conclusiones del Trabajo . . . 44

5.2. Extensiones 45

6. Conclusions and Future Work **47**
6.1. Conclusions 47
6.2. Future Work 48

Bibliografía **50**

Índice de figuras

2.1. Estructura del M1 6
2.2. Vista física del M1 6
2.3. Estructura de la CPU 7
2.4. Operación gemm 14
2.5. Empaquetado de Matrices 14
2.6. Representación del empaquetado 15

3.1. División de gemm en dos 19
3.2. Optimización de la Funcionalidades por Dispositivo 20
3.3. Uso de la GPU en macOS 21

4.1. Uso de Firestorm con $n = m = k = 20000$ 23
4.2. Uso de Firestorm con $n = m = k = 40000$ 23
4.3. Uso de Firestorm con $n = m = k = 40000$ y limitando los threads a 1 23
4.4. GFLOPS en *sgemm* 24
4.5. GFLOPS en *dgemm* 24
4.6. GFLOPS en *sgemm* por core 25
4.7. GFLOPS en *dgemm* por core 25
4.8. Consumo en *sgemm* 25
4.9. Consumo en *dgemm* 25
4.10. GFLOPS en *sgemm* 26
4.11. GFLOPS en *dgemm* 26
4.12. Rendimiento en *sgemm* 27
4.13. Rendimiento en *dgemm* 27
4.14. Consumo en *sgemm* 27
4.15. Consumo en *dgemm* 27
4.16. Uso de Icestorm con 1 thread 28
4.17. Uso de Firestorm con 1 thread 28
4.18. Consumo en 1 thread 28
4.19. Uso de Icestorm con 2 thread 28
4.20. Uso de Firestorm con 2 thread 28
4.21. Consumo en 2 thread 28
4.22. Uso de Icestorm con 4 thread 29
4.23. Uso de Firestorm con 4 thread 29
4.24. Consumo en 4 thread 29

4.25. Uso de Icestorm con 1 thread 29
4.26. Uso de Firestorm con 1 thread 29
4.27. Consumo en 1 thread 29
4.28. Uso de Icestorm con 2 threads 30
4.29. Uso de Firestorm con 2 threads 30
4.30. Consumo en 2 threads 30
4.31. Uso de Icestorm con 4 thread 30
4.32. Uso de Firestorm con 4 thread 30
4.33. Consumo en 4 thread 30
4.34. GFLOPS en *sgemm* 31
4.35. GFLOPS en *dgemm* 31
4.36. Rendimiento en *sgemm* 32
4.37. Rendimiento en *dgemm* 32
4.38. Consumo en *sgemm* 32
4.39. Consumo en *dgemm* 32
4.40. Uso de Icestorm con 1 thread 33
4.41. Uso de Firestorm con 1 thread 33
4.42. Consumo en 1 thread 33
4.43. Uso de Icestorm con 2 thread 33
4.44. Uso de Firestorm con 2 thread 33
4.45. Consumo en 2 thread 33
4.46. Uso de Icestorm con 4 thread 33
4.47. Uso de Firestorm con 4 thread 33
4.48. Consumo en 4 thread 33
4.49. *sgemm* en Background 34
4.50. *sgemm* en User Interactive 34
4.51. *dgemm* en Background 34
4.52. *dgemm* en User Interactive 34
4.53. Consumo *sgemm* en Background 35
4.54. Consumo *sgemm* en User Interactive 35
4.55. Consumo *dgemm* en Background 35
4.56. Consumo *dgemm* en User Interactive 35
4.57. Efiicencia *sgemm* en Background 36
4.58. Eficiencia *sgemm* en User Interactive 36
4.59. Eficiencia *dgemm* en Background 36
4.60. Eficiencia *dgemm* en User Interactive 36
4.61. Rendimiento en *sgemm* 37
4.62. Rendimiento en *dgemm* 37
4.63. Consumo en *sgemm* 38
4.64. Consumo en *dgemm* 38
4.65. Comparativa en *sgemm* 38
4.66. Comparativa en *dgemm* 38
4.67. Rendimientos ante diferentes tamaños en *sgemm* 39
4.68. Rendimientos ante diferentes tamaños en *dgemm* 39
4.69. Rendimiento 16x16 40
4.70. Comparativa ante BLIS AMX 40

4.71. Rendimiento por tamaños 40
4.72. Consumo frente BLIS AMX 40
4.73. Rendimiento en *sgemm* 41
4.74. Rendimiento en *dgemm* 41
4.75. Comparativa de rendimiento entre Accelerate y la GPU en *sgemm* 42

Índice de tablas

3.1. Tabla de Clases de QoS 18

4.1. Consumo en *sgemm* 41
4.2. Consumo en *dgemm* 41

Capítulo 1

Introducción

"Hay belleza en algo que funciona y que funciona intuitivamente"
— Jony Ive

La inteligencia artificial (IA) está presente en la vida cotidiana desde hace algunos años y se está volviendo algo esencial en diversos tipos de tecnología (robótica, coches inteligentes, etc). Una rama esencial de ella es el Deep Learning, que imita la forma de aprender de los humanos a base de ejemplos y que ha dado muy buenos resultados. Las utilidades del Deep Learning son múltiples y cada vez más usadas en el día a día. Algunos campos donde han sido usados estos métodos y se han obtenido buenos resultados son el procesamiento de lenguaje natural, la visión por ordenador (es decir, reconocimiento de formas del mundo real a través de imágenes por parte una máquina), aplicaciones de ciencia climática, etc.

Los cálculos matriciales de grandes volúmenes de datos son con frecuencia operaciones realizadas por los diversos métodos de Deep Learning. Por tanto, es de gran interés comparar las distintas librerías que ofrecen cálculos matriciales optimizados y compararlos entre si para decidir qué opción es mejor a la hora de usar métodos de Deep Learning.

1.1. Motivación

Este proyecto tiene por objeto principal analizar diversos aspectos de múltiples librerías BLAS (librerías especializadas en álgebra lineal) y compararlas con la librería Accelerate, que ha sido diseñada por Apple. Estas librerías son ampliamente usadas en aplicaciones que en general son de uso científico y que tienen altos costes computacionales (editores de vídeo, editores de imagen, etc). Dada la potencia que han adquirido los teléfonos móviles en la última década, su uso también ha sido extendido a estos y se usan en multitud de aplicaciones (asistentes de voz, aplicaciones de edición de imágenes, juegos, etc).

La elección de este tema corresponde a que las librerías BLAS y el Deep Learning son una parte muy ligada a nuestros estudios, ya que con frecuencia hacemos uso de estos métodos o haremos uso de ellos en nuestra vida laboral. Otra parte importante de la elección de este proyecto ha sido el poder trabajar con una máquina tan actual como el Mac Mini M1, puesto que es uno de los ordenadores más actuales y potentes del mercado y supuso una gran revolución en el mundo tecnológico cuando fue presentado, por lo que poder aprender de él ha sido uno de los incentivos de la elección de este tema. Por tanto, este

proyecto ha sido visto como una oportunidad para aprender más profundamente acerca de estas librerías, su uso y sus implementaciones internas, así como del nuevo Mac Mini. Es decir, además de ser un tema que involucra muchos conceptos de actualidad, es un tema muy completo con el que se puede aprender mucho.

Por este motivo y por todas sus aplicaciones en el mundo actual he decidido escoger este tema.

1.2. Objetivos

El presente proyecto tiene por objetivo el analizar diversos aspectos como el consumo o el rendimiento de la librería Accelerate de Apple y comparar los datos obtenidos con los de otras librerías BLAS como BLIS u OpenBLAS, que son muy actuales y sirven bien para poder comparar resultados. Se analizan estos aspectos ya que los métodos de Deep Learning usan estas librerías con frecuencia debido a los numerosos cálculos que hacen constantemente (y con volúmenes de datos grandes habitualmente).

A la hora de elegir una u otra librería, hay diversos aspectos que pueden tenerse en cuenta y algunos pueden ser más prioritarios que otros. En el caso de las aplicaciones de ordenador o servidores, el consumo energético no suele ser una prioridad, siendo prioritario el rendimiento. Sin embargo, en aplicaciones móviles puede interesar más usar librerías más eficientes dado que las baterías suelen tener tamaños reducidos y el consumo energético es muy importante.

1.3. Estructura del Documento

Para este proyecto se ha decidido dividir el contenido en cinco capítulos que se describen a continuación:

- En el Capítulo 1 se realiza una introducción a la temática del proyecto y los objetivos que se han alcanzado.
- En el Capítulo 2 aborda el estado actual tanto del procesador Apple M1, presente en el Mac Mini utilizado, así como de las librerías BLAS, haciendo más hincapié en el funcionamiento interno de BLIS.
- En el Capítulo 3 se detalla el proceso que se ha seguido, desde las pruebas iniciales del proyecto hasta las diferentes pruebas de rendimiento y consumo que se han realizado, tanto a nivel de librerías BLAS como de GPU.
- En el Capítulo 4 se analizan diferentes aspectos de los resultados obtenidos con cada librería, describiendo las conclusiones obtenidas. Tras esto, se comparan estos mismos aspectos entre todas las librerías y se analiza también por separado implementaciones que imitan a la librería BLIS. Por último, se analiza el rendimiento de la librería Accelerate frente a la librería MetalPerformanceShaders, puesto que ambas usan partes distintas del chip M1 para obtener los resultados de las operaciones realizadas.

- En el Capítulo 5 se presentan las conclusiones obtenidas tras la ejecución de las diversas pruebas y se analiza la utilidad más óptima para cada librería. Además, en este capítulo se exponen las posibles ampliaciones que se pueden realizar para la continuación de este proyecto.

Capítulo 2

Estado del Arte

Antes de comenzar a desarrollar el trabajo realizado en el presente proyecto y analizar los resultados obtenidos, es necesario hacer un estudio tanto del ordenador donde se van a ejecutar las pruebas como de las librerías BLAS, que son las utilizadas en este proyecto.

Desde que se presentó en 2020 (Apple (2020)), el SoC Apple M1 ha sido una revolución en el mundo de la informática y sus distintas evoluciones presentadas en los últimos meses han seguido sorprendiendo. Además de un rendimiento muy bueno, Apple presentó este chip como muy eficiente. Sin embargo, hay múltiples aspectos de este chip que en la actualidad no son conocidos completamente o no se sabe cuál es su utilidad. El primer punto del presente capítulo hace un estudio y recopilación de todos los datos conocidos hasta la actualidad para saber sobre qué se está trabajando y poder comprender los resultados obtenidos.

Las librerías BLAS son esenciales a la hora de medir el rendimiento del Apple M1, puesto que están conformadas por un conjunto de funciones que ponen a prueba el rendimiento y consumo del ordenador de una forma sencilla y directa. Además, al ser algunas librerías multiplataforma, la comparación de rendimiento entre diversas máquinas se hace de forma fácil.

2.1. Apple M1

2.1.1. Contexto

Desde su fundación en 1976 hasta la actualidad, Apple ha llevado a cabo varias transiciones de procesadores en sus equipos. Las más recientes han sido la transición de PowerPC a Intel en los equipos Mac en 2005, la transición de procesadores Samsung a procesadores propios de Apple en 2010 (incluyendo el chip A4 en el iPhone 4, iPod touch 4 y en el iPad 1) y la más reciente de todas: la transición de procesadores Intel a procesadores propios de Apple en los equipos Mac, los llamados Apple Silicon. Esta última viene motivada por el buen rendimiento que Apple ha obtenido en sus dispositivos móviles (los chips más modernos como el A13 o el A14 del iPhone superan en rendimiento a muchos portátiles actuales) y el descontento de Apple con los procesadores Intel.

Para entender en detalle el descontento con los procesadores Intel, hay que remontarse

a mediados de 2015 y entender la nueva dirección que Apple estaba dando con sus ordenadores portátiles, la cual quedó patente en la presentación del MacBook de 12". Es esta serie de ordenadores, se sustituyeron todos los puertos existentes por puertos USB C, el teclado con mecanismo de tijera por el teclado de mecanismo de mariposa y se redujo notablemente el grosor del cuerpo del ordenador. Quedó corroborado que Apple tomaba esta nueva dirección cuando en 2016 Apple lanzó al mercado los MacBook Pro que seguían la misma línea que el MacBook de 12".

Sin embargo, las innovaciones anteriores trajeron consigo varios problemas: los teclados fallaban con el tiempo y hacían falta multitud de adaptadores para poder conectar periféricos a los ordenadores. Además, el diseño más compacto se tradujo en sobrecalentamientos frecuentes debido a que había menos espacio para la ventilación y enfriamiento de los componentes internos y los procesadores Intel generaban mucho calor en cuanto tenían que enfrentarse a cargas de trabajo algo grandes. Para evitar dañar el equipo, el sistema operativo bajaba la frecuencia del procesador (conocido como "throttling.en inglés) y este rendían menos. Así, unos procesadores que de entrada no eran excelentes en términos de rendimiento, comenzaron a rendir todavía peor. Fue este descontento con los procesadores Intel lo que, junto al éxito de los procesadores móviles propios, lo que motivó a Apple a desarrollar los Apple Silicon para sus futuros equipos Mac.

A finales de 2020, Apple presentó en el evento "One More Thing" los nuevos MacBook Air, el nuevo MacBook Pro de 13 pulgadas y el nuevo Mac Mini. A pesar de que trajeron multitud de innovaciones, la que mayor atención captó fue sin lugar a dudas el nuevo procesador que estos equipos montaban: el Apple M1. Este era un procesador de diseño propio de Apple que daría comienzo a una nueva transición y que a día de hoy sigue en marcha y que probablemente termine cuando el Mac Pro (el único equipo con procesador Intel que vende Apple) pase también a llevar Apple Silicon. Tal fue el éxito del Apple M1 que desde entonces, Apple ha presentado nuevas variantes del M1 como son el M1 Max o el M1 Ultra que van más enfocadas a profesionales (editores de vídeo, productores de música, desarrolladores, etc).

2.1.2. Arquitectura

El Apple M1 fue presentado por Apple como la gran parte de los procesadores que han presentado en las últimas décadas: se presentan sus componentes de forma general (la GPU, la memoria caché, las DRAM, etc), se destacan algunas innovaciones importantes (organización de los cores, eficiencia, etc) y se muestran algunos datos (eficiencia frente a competidores, número de transistores en el chip, etc). Sin embargo, Apple no tiende a hablar en profundidad de estos componentes, por lo que la mayoría de los datos recopilados en este trabajo son datos obtenidos por investigadores que han analizado el M1 y han publicado sus conclusiones.

Los equipos Mac previos al Apple M1 llevaban una placa base donde cada componente iba por separado y se interconectaban a través de dicha placa. En el M1, Apple ha reunido (Anandtech (2020a); SamaGames (2020)) la GPU, la CPU y el Neural Engine (el chip dedicado a Machine Learning, el cual se tiende a utilizar cuando se utilizan las frameworks de Core ML) en un solo chip. Estos tres componentes están interconectados por lo que Apple llama "Fabric", que es un método de interconexión basado en conectar directamente

los chips entre si a través del silicio con conexiones tan pequeñas como las que hay dentro del propio chip, lo que permite muchas más conexiones y más rápidas, que sustituye a las conexiones PCB típicas. Además, dentro del M1 encontramos también dos módulos de DRAM y una memoria caché.

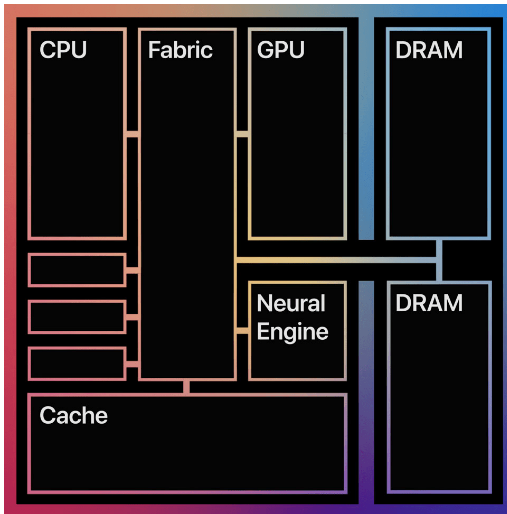

Figura 2.1: Estructura del M1

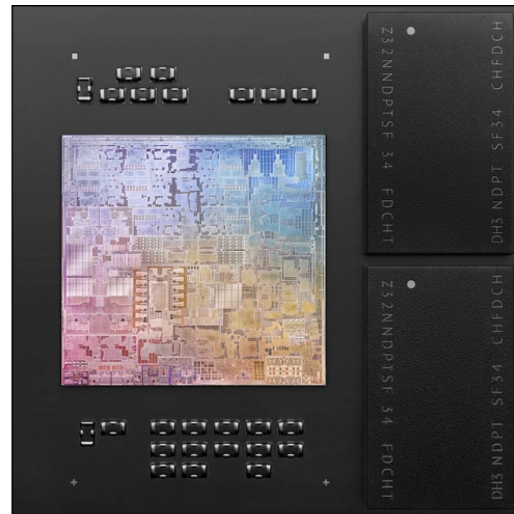

Figura 2.2: Vista física del M1

Junto a lo mencionado anteriormente, Apple ha fabricado el M1 con tecnología de 5 nm, por lo que ha podido meter 16 mil millones de transistores en un área de 119 mm ^{2}.

2.1.2.1. CPU

Desde que se presentó el iPhone 7 con su chip A10 Fusion, Apple ha dividido sus CPU en cores de alto rendimiento y cores de bajo consumo. Esta idea es la que Apple ha seguido también con el M1, que es muy similar al chip A14 presentado en Septiembre de 2020. Apple lanzó al mercado el iPhone 12, que integraba dicho chip y que integraba dos cores de alto rendimiento (llamados Firestorm) y cuatro de bajo consumo (llamados Icestorm). Además, integraba también una GPU de 4 cores. El M1 utiliza la misma estructura aunque tiene dos cores Firestorm más y cuatro cores más de GPU. Esto se ha hecho porque un ordenador puede permitirse más potencia y más consumo que un teléfono móvil, donde la batería es mucho más reducida.

El cluster entero de la CPU junto a la GPU del A14 ocupa 21,5 mm^2, mientras que en el M1, al añadir dos cores Firestorm más y doblar el tamaño de la GPU, el tamaño se incrementa en 31 mm^2. Cada core del Firestorm junto a sus cachés L1 y L2 ocupan 2,3 mm^2, mientras que en los Icestorm ocupan 0,6 mm^2. Todos los cores de Firestorm y Icestorm constan de una caché de instrucciones, una de datos y otra caché L2. Sin embargo, como Firestorm tiene una finalidad distinta a la de Icestorm, las cachés de su cores difieren entre si:

- Los cores de Firestorm tienen cada uno una caché de instrucciones L1 de 192 KB y otra L1 de datos de 128 KB, y los cores comparten una caché L2 de 12 MB.
- Los cores de Icestorm tienen cada uno una caché de instrucciones L1 de 128 KB y otra L1 de datos de 64 KB, y los cores comparten una caché L2 de 4 MB.

Además, Firestorm y Icestorm comparten una caché SLC que se cree que es de 16 MB y un coprocesador matricial conocido como AMXMedium (2021) (no es el nombre oficial puesto que Apple no lo ha hecho público, pero tampoco tiene el acceso restringido). Los dos tipos de core soportan el set de instrucciones de ARMv8.4-A, con instrucciones Neon SIMD para procesamiento paralelo de operaciones enteras y de coma flotante.

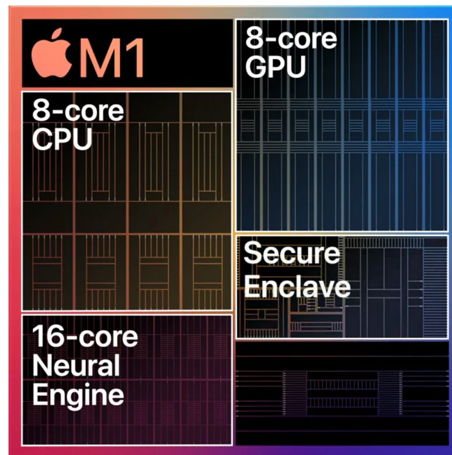

Figura 2.3: Estructura de la CPU

Hasta el momento no se ha encontrado forma de elegir explícitamente qué tipo de core queremos usar, pero sí podemos indicarle al sistema cuál queremos usar indicando la calidad de servicio (QoS) que queremos en nuestras aplicaciones (aunque esto no garantiza que se vayan a ejecutar en esos cores).

Utilizando el trabajo de Dougall Johnson (Johnson (2020a,b)) se puede ver que cada core de Firestorm consta de las siguientes unidades:

- Seis unidades no iguales entre si para tratamiento de enteros.
- Dos unidades de load, una de load/store y otra de store, teniendo estas dos últimas acceso al AMX.
- Cuatro unidades no iguales entre si para tratamiento de números en coma flotante y SIMD.

Los cores Icestorm tienen menos unidades que Firestorm:

- Tres unidades no iguales entre si para tratamiento de enteros.
- Una unidad de load y una de load/store, teniendo esta última acceso al AMX.
- Dos unidades no iguales entre si para tratamiento de números en coma flotante y SIMD.

2.1.2.2. GPU

La GPU del M1(Anandtech (2020b); Notebookcheck (2020)) consta de ocho cores y ocupa 31 mm^2 más que la GPU del A14. Sin embargo, hay una ligera diferencia entre la

GPU del M1 de los MacBook Air más básicos y el resto de MacBook/Mac presentados: a pesar de tener ocho cores, el MacBook Air más básico tiene uno desactivado, frente al resto de modelos que tienen los ocho cores funcionales. Esto se debe probablemente a que a la hora de pasar los controles de calidad, hay chips M1 que no pasan los controles para los ocho cores, por lo que se utilizan como chips con GPU de 7 cores (aunque tengan ocho físicamente, son siete a efectos prácticos). Esto se hace para aprovechar chips que, de no hacerse esto, irían a la basura. Esta diferencia es algo que el usuario medio no apreciaría y es una forma de ofrecer un M1 a un precio menor con casi las mismas características que el presente en el MacBook Pro.

Todos los cores de la GPU constan de 16 unidades de ejecución, que se componen de 8 ALU cada una, haciendo un total de 1024 ALU. Estas son capaces de ejecutar, de forma simultanea, 24,576 threads. El acceso a la GPU se hace mediante las librerías de Metal y similares. La frecuencia máxima de la GPU es de 1278 MHz y el throughput (en FP32) alcanza los 2,6 TFLOPS. Además, la GPU consta también de 64 unidades de texturizado (aquellas unidades dedicadas a hacer "sampling") que pueden alcanzar, en conjunto, un máximo de 82 gigatexels/s, y 32 unidades de renderizado que alcanzan, en conjunto, 41 gigapixels/s.

2.1.2.3. Memoria

El Apple M1 consta de una memoria RAM unificada que comparten CPU, GPU y algunos coprocesadores. Tiene gran ancho de banda y baja latencia. La memoria es una LPDDR4X SDRAM de 8/16 GB (según configuración, siendo los modelos más básicos de los MacBook Pro/Air los que llevan 8 GB. La ventaja de esta memoria unificada es que las transferencias de datos entre distintas unidades del chip se hacen directamente desde la memoria unificada y no tienen que acceder a memorias intermedias que puedan tener distintas unidades.

2.1.2.4. Coprocesadores

En su presentación, Apple mencionó el Neural Engine y el Secure Enclave Processor, que son los principales coprocesadores del M1. Sin embargo, hay algunos otros que están ocultos y que Apple, aunque no ha restringido su uso, tampoco ha documentado, por lo que su acceso es bastante complicado.

El Neural Engine, utilizado para Machine Learning (siendo introducida la primera versión de este coprocesador por primera vez en el chip A11 del iPhone X), consta de 16 cores con acceso directo a la memoria y con una caché L2 compartida de 4 MB. Este coprocesador parece ser que funciona con tensores de 5 dimensiones (los tensores son una generalización de los vectores) con enteros de 8 bits sin signo, con signo y FP16. Este solo puede ser accedido a través de la librería de Core ML que provee Apple. Con esto, se pueden hacer convoluciones y multiplicaciones de matrices, pero de baja precisión en comparación con otros coprocesadores existentes.

Dicho Neural Engine ha sido investigado por George Hotz (conocido popularmente como "Geohotz muy famoso por ser de las primeras personas en obtener root en iPhone en 2007) y estima que cada core del Neural Engine alcanza los 6875,5 GOPS. Además, se cree

que estos son MAC (multipply-accumulate) de 32x32 a 335 MHz.

Otro de los coprocesadores bien documentados por Apple es el Secure Enclave Processor (SEP) que se encuentra dentro del Secure Enclave (Apple (2021d)). Este último sirve como un lugar seguro donde almacenar todos los datos sensibles del usuario (contraseñas, huellas dactilares del Touch ID, registros faciales del Face ID, tarjetas, etc) y cuenta con el SEP como unidad de procesamiento de datos.

El SEP se usa exclusivamente para el Secure Enclave y utiliza una versión adaptada del microkernel L4 de Apple. Funciona a una velocidad baja para consumir poco e incluye memoria encriptada, arranque seguro, varios generadores de números aleatores y el motor AES.

Sin embargo, hay otros coprocesadores que Apple ni mencionó en su presentación ni ha documentado y que han sido incorporados al M1. Estos no son accesibles (aunque su uso tampoco está restringido) y ha sido necesaria ingeniería inversa por parte de varios investigadores para saber de su existencia y poder documentar lo que se ha podido. Uno de ellos el conocido como AMX (Apple Matrix Coprocessor, que no es el nombre oficial pero es como los investigadores se refieren a él). Este es usado por librerías como Accelerate y, aunque Apple no ha dado otra forma de acceder que no sea a través de estas librerías, investigadores como Dougall Johnson han logrado, a través de ingeniería inversa a la librería Accelerate, obtener acceso al AMX y documentarlo en gran parte.

El AMX es un coprocesador enfocado a realizar operaciones matriciales y con más precisión que el Neural Engine, puesto que trabaja con FP32 y FP64. A pesar de que se creía que inicialmente que había un AMX para Firestorm y otro para Icestorm, investigaciones posteriores revelaron que solo hay un AMX en el M1, y es compartido entre Firestorm y Icestorm.

El trabajo de Johnson (Johnson (2022)) ha permitido tener una documentación bastante completa acerca del AMX. Tras las investigaciones de este, se descubre que este coprocesador no es especulativo y las instrucciones se envían a través de instrucciones de load via CPU. También se documentan que el estado del coprocesador consiste en dos registros de 0x200 byte, conocidos como $amx0(''x'')$ y $amx1(''y'')$. Además, hay un tercer registro de 0x1000 byte conocido como $amx2(''z'')$. Los registros x e y son definidos en las cabeceras de Accelerate como grupos de registros donde cada fila de 64 bytes es un registro, mientras que a z lo describen algo más: "64 registros en una matriz MxN. Además, existe un registro conocido como AMX_STATE_T _EL1 de 64 bit que posiblemente registre si el AMX está activado o no (aunque se cree que registra otros estados también). Cada uno de estos registros realizan el load/store con filas de 0x40 bytes.

Las instrucciones del AMX tienen la estructura:

$$0x00201000|((op\&0x1F) << 5)|(operando\&0x1F)$$

Para activar el AMX, se ha de establecer que $op = 17$ y $operando = 0$. Para desactivarlo, hay que usar la misma operación pero el operado ha de valer 1. Si se intentan ejecutar instrucciones sin que el AMX esté activado previamente se tratarán como instrucciones ilegales. Por tanto, la operación 17 sirve para activar/desactivar el coprocesador, mientras que el resto de operaciones sirven para lo siguiente:

- Las operaciones 0-7 sirven para hacer load/store en los registros x, y y z. Más concretamente, las operaciones 0 y 1 corresponden a los load de x e y. Las operaciones 2 y 3 hacen load en los mismos registros, y las operaciones 4 y 5 hacen load y store en z respectivamente. Las operaciones 6 y 7 hacen load y store en z pero en orden diferente.
- Las operaciones 8 y 9 sirven para manipular datos (extracción de filas/columnas).
- Las operaciones de la 10 a la 16 sirven para diversas multiplicaciones.
- De las operaciones 18 a la 22

Otro coprocesador que Apple no ha documentado ha sido el Display Coprocessor (Org (2021)) pero que el grupo Asahi Linux ha investigado para poder crear un distro (todavía no acabado) de Linux al M1. Cuando la gente piensa en la GPU, piensa en una sola unidad cuando en realidad son dos: la GPU propiamente dicha y el controlador de pantalla. Normalmente, estos controladores suelen ser dispositivos con algunos registros, aunque con el DCP, Apple ha decidido meter en él la mitad del driver de pantalla y crear una interfaz para que se comuniquen entre si las dos mitades del driver en lugar de tener el driver en un solo lugar como es habitual.

Además de estos, hay otros coprocesadores de los que apenas hay información pero que se sabe que están presentes en el M1, que son:

- Always On Processor (AOP), que controla sensores ambientales y activaciones de sistemas.
- Apple Video Decoder (AVD), para decodificación de vídeo.
- Apple Video Encoder (AVE), que codifica vídeo.
- Power Management Processor (PMP), para la gestión de energía del chip.
- Apple Graphics (AGX) que no se sabe si es un coprocesador en sí o es un alias de la GPU.

Todos los coprocesadores mencionados anteriormente tienen su propio sistema operativo: RTKit. Este es un sistema propio de Apple para estos dispositivos que tiende a utilizar en sus coprocesadores.

2.1.2.5. Fabric

Esta sección del chip (Notebookcheck (2021); Company (2021)) M1 es una de las responsables de la alta velocidad del mismo, puesto que permite a los componentes conectarse directamente a través de una placa de silicio que lleva el cableado (tan pequeño como el que hay dentro de los propios chips) directamente de una unidad a otra. Esto es una gran ventaja respecto a las conexiones PCB (Printed Circuit Board) donde las conexiones posibles son menores y más lentas. Este método de interconexión de las distintas unidades no es nuevo para Apple, puesto que lo utilizaron por primera vez en el Mac Pro actual, que consta de una AMD Vega Pro II Duo donde hay dos GPU interconectadas mediante otra conexión Fabric.

Es en esta sección del chip donde se encuentran varios coprocesadores y donde se realizan algunas tareas de entrada/salida. Para acceder a los controladores de entrada/salida, hay que acceder a DART (Device Address Resolution Table), que es un componente cuya finalidad es mapear el espacio de direcciones de periféricos PCI. Algunos de estos son la pantalla, los puertos Thunderbolt, Wi-Fi, Ethernet, Bluetooth, etc.

2.2. Librerías BLAS

Las librerías BLAS (Wikipedia (2022)) son rutinas de bajo nivel que realizan operaciones de álgebra lineal de forma muy eficiente. Estas surgieron por primera vez como una librería de Fortran en 1979 y su interfaz fue estandarizada al cabo de poco tiempo. Desde entonces, esta librería es tomada como la librería de referencia. Por tanto, la mayoría de librerías BLAS que se han creado con el mismo propósito tienen rutinas que se ajustan a la interfaz BLAS, lo que es una ventaja a la hora de programar, puesto que se puede enlazar con distintas librerías sin necesidad de cambiar el código. Algunas de estas librerías son ATLAS, Intel MKL (para dispositivos con procesadores Intel), OpenBLAS (de código abierto y optimizada para multitud de arquitecturas populares), Accelerate (librería de Apple optimizada para sus equipos), BLIS, etc.

Dentro de BLAS hay tres niveles de operaciones:

- Nivel 1: realiza operaciones escalar-vector y vector-vector.
- Nivel 2: este nivel se encarga de operaciones vector-matriz.
- Nivel 3: las operaciones entre matrices se realizan aquí y es donde las pruebas de este documento se han llevado a cabo.

En esta sección se presta especial atención a estas tres últimas ya que han sido las librerías usadas en este proyecto. Se hace un análisis en detalle más en profundidad de BLIS, ya que he tenido la suerte de tener de tutor a Rafael Rodríguez, que ha trabajado en el desarrollo de algunas secciones de BLIS (como el microkernel de ARMv8), y he podido ver más en profundidad su funcionamiento interno.

2.2.1. Accelerate

Accelerate (Apple (2022a,b))es una de las múltiples librerías que Apple suministra a los desarrolladores y va enfocada al procesamiento de imágenes, señales, redes neuronales, BLAS, etc. Esta librería fue introducida en macOS en 2013 y su objetivo era reunir en una sola librería muchas de las librerías existentes por entonces y agruparlas en una sola. Esto tiene sentido ya que muchas de las librerías que reunió suelen usarse conjuntamente.

Esta librería ha sido utilizada en este proyecto ya que en el Mac Mini M1 utilizado para las pruebas ha dado un rendimiento excelente y sirve de comparación frente a otras librerías que no tienen acceso al AMX. Además, es esta librería la que ha permitido hacer ingeniería inversa y sacar las instrucciones necesarias para poder acceder al AMX.

Esta librería se puede utilizar en los siguientes sistemas operativos: iOS 4.0 o posterior, iPadOS 4.0 o posterior, macOS 10.3 o posterior, tvOS 9.0 o posterior y watchOS 2.0 o posterior. Accelerate reune las siguientes librerías:

- **BNNS**: reúne las subrutinas necesarias para la creación y ejecución de redes neuronales, tanto para entrenamiento como para inferencia
- **vImage**: se compone de multitud de funciones para tratamiento de imágenes. Entre sus utilidades están el procesamiento de imágenes de gran tamaño, procesamiento de vídeo en tiempo real, conversión de formatos de imagen, etc.
- **vDSP**: muy útil para el procesamiento de señales digitales, ya que puede hacer transformadas de Fourier en una y dos dimensiones, aritmética de matrices y vectores, convoluciones, conversiones de tipos, etc.
- **vForce**: esta librería contiene funciones a aplicar sobre vectores de cualquier tamaño (por ejemplo, funciones trigonométricas, logarítmicas, etc). Se puede utilizar tanto para precisión simple como para precisión doble.
- **Sparse Solvers**: las Sparse Solvers son un conjunto de funciones destinadas a resolver sistemas de ecuaciones lineales en matrices donde la mayoría de los elementos de esta son ceros.
- **BLAS y LAPACK**: estas también van enfocadas a resolver problemas comunes de álgebra lineal, aunque van más destinadas a funciones básicas (suma de matrices, de vectores, etc). Es la antigua vecLib.

Aunque no están incluídas en Accelerate, cabe destacar las librerías Apple Archive (para compresión multithread de directorios, ficheros y datos sin pérdida de calidad), Compression (contiene algoritmos de compresión de datos) y simd (para realizar operaciones en vectores y matrices pequeños), ya que están altamente relacionadas con ella y bien podrían encontrarse dentro de Accelerate.

2.2.2. OpenBLAS

OpenBLAS (Group (2022b,c,a))es una librería open source que implementa los tres niveles de operaciones BLAS (vector-vector, matriz-vector y matriz-matriz) que va enfocada al uso en arquitecturas populares, haciéndola una buena alternativa a Accelerate. Esta librería es un fork de GotoBLAS2, ya que cuando el creador de esta última abandono el proyecto, el resto del equipo retomó el trabajo bajo el nombre de OpenBLAS.

Uno de los puntos fuertes de esta librería es que es multiplataforma, es decir, a diferencia de Accelerate (que está solo disponible en macOS), OpenBLAS está disponible para macOS, Windows, Linux y. FreeBSD. Además, en ciertas arquitecturas de Intel, obtiene mejor rendimiento que la propia IntelMKL en el apartado BLAS. Sin embargo, uno de los inconvenientes es que está desarrollado por voluntarios, por lo que depende de que algún miembro del equipo tenga tiempo para adaptarla a las nuevas arquitecturas que vayan saliendo al mercado. Mientras que Accelerate, esta está disponible para cualquier equipo (lleve o no una nueva arquitectura) que corra macOS desde el primer día, OpenBLAS puede no estar adaptada a una nueva arquitectura hasta pasados unos meses (o años quizás).

Las arquitecturas para las que está disponible son x86, x86-64, PowerPC64, AArch64 (ARM), IBM Z, SPARC y RISC-V.

2.2.3. BLIS

BLIS (BLAS-like Library Instantiation Software) (Science of High-Performance Computing (SHPC,S); Jianyu Huang (2022)) es otra librería open source publicada por primera vez en 2013 y que ha ido desarrollándose con los años hasta estar adaptada para múltiples procesadores concretos. Se expone al usuario mediante dos interfaces: la interface tradicional BLAS, y la interfaz de CBLAS y tiene como finalidad la portabilidad. Sin embargo, esto no impide que se obtenga un rendimiento muy bueno con esta librería.

Los objetivos de BLIS son los siguientes:

- **Alta portabilidad**: este objetivo se logra con kernels computacionales enfocados a arquitecturas concretas. Por tanto, una vez implementado este kernel, las operaciones de nivel 2 y 3 se ven mejoradas en términos de rendimiento al instante. Es decir, teniendo estos kernels computacionales, el acceso a BLIS es bastante sencillo.

- **Almacenamiento generalizado de matrices**: BLIS ofrece indicar si se ha optado por almacenamiento de matrices por columnas, por filas o por un stride general (esto es muy útil para tensores). Frente al BLAS tradicional (que usa almacenamiento por columnas) y a la interfaz de CBLAS (que permite almacenamiento por filas pero no ofrece mezclas de formatos), esto es un logro significativo.

- **Buen soporte del dominio complejo**: la librería ofrece multitud de funciones no disponibles en BLAS tradicional (como las formas de Hermite complejas).

- **Soporte del multithreading**: por defecto esta característica está desactivada, pero puede configurarse para que soporte OpenMP o POSIX threads (pthreads). Esto será muy útil a la hora de paralelizar operaciones matriciales.

Gracias a la ayuda del profesor Rafael Rodríguez se ha podido ver en detalle el funcionamiento interno de esta librería. En concreto, los esfuerzos se han centrado en comprender cómo funcionan los microkernels de BLIS y cómo se utilizan. Estos microkernels contienen código ensamblador (con desenrollado de bucles y algunas optimizaciones adicionales) que realiza las operaciones de forma muy eficiente. Para cada tipo de operación y para cada arquitectura concreta es necesario tener un microkernel que permita realizar las operaciones deseadas.

Muchos de los trabajos de las últimas dos décadas han ido enfocados a optimizar el rendimiento de la operación *gemm* (GEneral Matrix Multiplication, ver Figura 2.4), puesto que es una parte crucial de BLAS. Este trabajo ha sido realizado por diferentes entidades como Apple, Intel, AMD, instituciones open source, etc.

Cuando la operación *gemm* se realiza en FP32 se denomina *sgemm* (Single precision GEneral Matrix Multiplication), mientras que si se realiza en FP64 (es decir, doble precisión en los cálculos), la operación se denomina *dgemm* (Double precision GEneral Matrix Multiplication). Existen otras operaciones *gemm* como son *cgemm* y *zgemm* (precisión simple y doble con números complejos respectivamente) que no han sido utilizadas en las

pruebas de este proyecto pero. que BLIS soporta.

A pesar de que BLIS lleva consigo otras optimizaciones, en este proyecto se detalla cómo se usan los microkernels (de una forma general) y qué realizan estos. La idea general es, en base a cinco bucles anidados (ver Figura 2.5), aprovechar al máximo la velocidad de las cachés, realizando poco a poco la operación *gemm*.

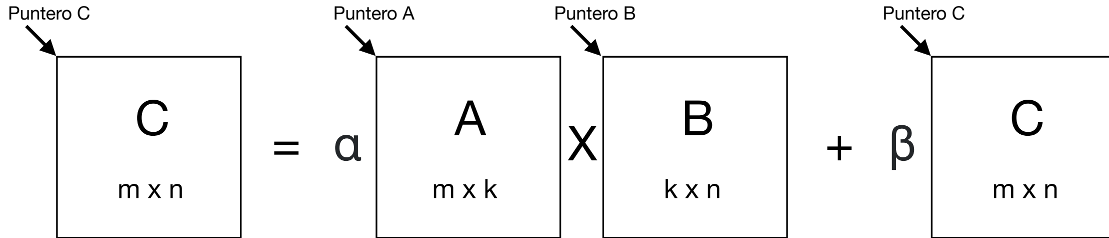

Figura 2.4: Operación gemm

Siendo las matrices A, B y C de tamaños m x k, k x n y m x n respectivamente, entre los bucles 1 y 3 se realiza lo que se llama 'empaquetado'. Esto consiste en obtener pequeños bloques de cada matriz y subirlos a la memoria caché para poder tener accesos mucho más rápidos. A continuación, se anidan otros dos bucles en torno al microkernel, que realiza las operaciones pertinentes.

Loop 1 **for** $j_c = 0, \ldots, n-1$ **in steps of** n_c
Loop 2 **for** $p_c = 0, \ldots, k-1$ **in steps of** k_c
$B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$ // Pack into B_c
Loop 3 **for** $i_c = 0, \ldots, m-1$ **in steps of** m_c
$A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$ // Pack into A_c
Loop 4 **for** $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r // Macro-kernel
Loop 5 **for** $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r
$C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ // Micro-kernel
$\mathrel{+}= A_c(i_r : i_r + m_r - 1, 0 : k_c - 1)$
$\cdot\ B_c(0 : k_c - 1, j_r : j_r + n_r - 1)$
endfor
endfor
endfor
endfor
endfor

Figura 2.5: Empaquetado de Matrices

Con este proceso es principalmente con lo que la librería BLIS obtiene en gran parte sus mejoras de rendimiento frente a multiplicaciones realizadas de forma genérica. Además, hay otras mejoras añadidas que no se ven en el presente proyecto que hacen que su rendimiento aumente aún más.

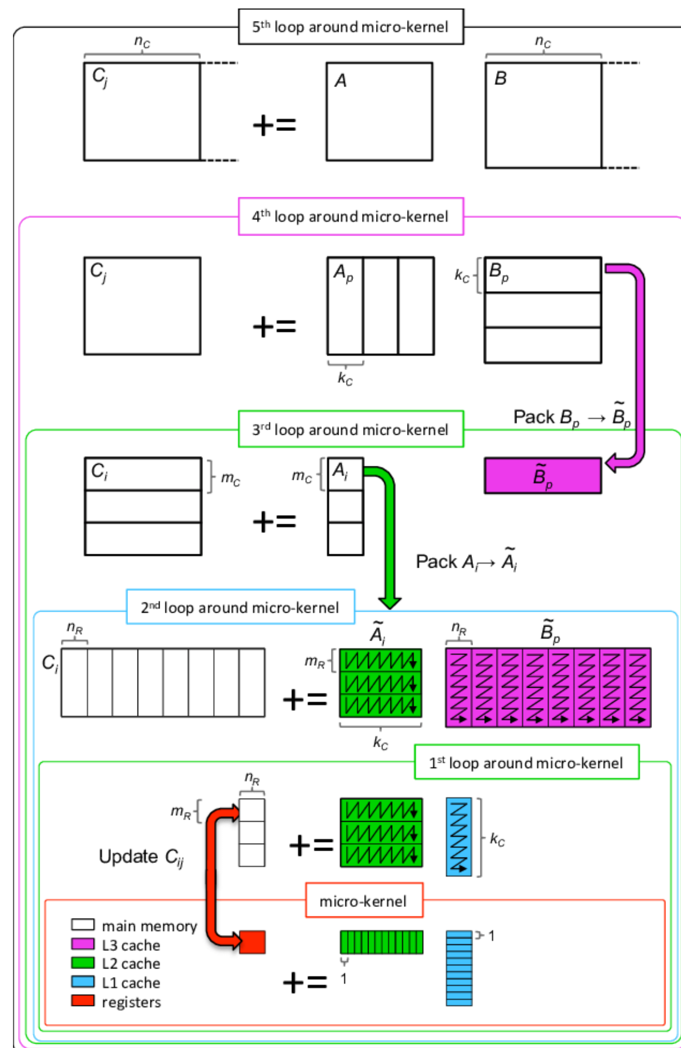

Figura 2.6: Representación del empaquetado

Capítulo 3

Descripción del Trabajo

3.1. Documentación y Puesta en Marcha

Los trabajos iniciales se centraron en leer acerca del M1 y de las distintas librerías BLAS. Se hizo especial hincapié en BLIS debido a la gran documentación que existe sobre esta librería y que al ser de código abierto se puede tener acceso a su implementación, frente al resto de librerías que no tienen tanta documentación disponible en Internet.

Tras la lectura y comprensión de la temática del proyecto, las siguientes tareas a realizar fueron implementaciones de multiplicaciones simples de matrices en Swift con Accelerate y con Metal (la librería de Metal Performance Shaders realiza multiplicaciones matriciales en la GPU) con éxito. Para la familiarización con las pruebas posteriores, el código en Swift se tradujo a C.

3.2. Uso de las Distintas Librerías

Una vez realizadas las pruebas iniciales, comenzó la familiarización con BLIS y sus microkernels. El profesor Rafael Rodríguez preparó un código que utilizaba la rutina *dgemm* de BLIS para posteriormente llamar a una función que imitaba en gran parte el funcionamiento de BLIS: empaquetaba bloques de cada matriz, llamaba al microkernel y se realizaba la multiplicación por partes. La finalidad de este código es servir de base para el resto de pruebas con las diferentes librerías y poder comparar los resultados obtenidos entre si. Además, gracias a este código se puede comprobar que los resultados obtenidos son correctos. En el código de pruebas, se establece que los valores α y β valen 2, y las matrices A, B y C se obtienen utilizando una función que rellena las matrices con valores aleatorios.

Sin embargo, el código provisto utilizaba el microkernel para la arquitectura Haswell de Intel. Por tanto, el primer trabajo consistió entonces en analizar cómo se podía hacer uso del microkernel para la arquitectura ARMv8 para sustituirlo por el microkernel para Haswell. Para ello fue necesario analizar en profundidad el código y buscar por los archivos de configuración de BLIS. En la carpeta de configuración se encuentra el tamaño de bloques necesario para la arquitectura ARMv8. Tras unos días, se resolvió y se comenzó a adaptar el código que realizaba *dgemm* tanto en Accelerate como en OpenBLAS.

La librería Accelerate viene por defecto en macOS, por lo que su implementación fue bastante rápida. Sin embargo, la instalación de OpenBLAS en el M1 fue algo tediosa porque hay poca información al respecto de cómo instalarla en el M1. Tras su instalación se pudo hacer uso de la librería sin problemas. Para poder tener un código equivalente al de BLIS, otra de las tareas fue investigar cómo modificar el número de threads que utilizan los programas. Para OpenBLAS es sencillo porque la propia librería suministra una función que permite cambiar el número de threads.

Tras investigar extensivamente la documentación propia de Apple, los foros de desarrolladores de la propia compañía y el manual integrado en el sistema, se llega a la conclusión de que la mayoría de Accelerate funciona en single thread, aunque hay algunas secciones de vDSP que funcionan en multithreading. Es preciso indicar que vDSP, a pesar de que tiene funcionalidades similares a BLAS, no realiza exactamente las mismas operaciones. Respecto a las funciones BLAS, Apple no específica si las funciones usan o no multithreading, aunque tras las pruebas realizadas se concluye que es el sistema el que elige cuántos threads usar en función del tamaño del problema. Para las pruebas realizadas, Accelerate funciona en single thread.

Tras estas adaptaciones se tenía un código para las tres librerías que realizaba la operación *dgemm* con un thread con un tamaño inicial de $m = n = k = 1000$, que iba aumentando de 5000 en 5000 hasta llegar a 31000 (inclusive). Este proceso se repetía con 2 y 4 threads en OpenBLAS y BLIS.

Otra de las pruebas realizadas es medir el rendimiento de las librerías no solo en *dgemm* sino también en *sgemm*. Por tanto, todo el código anterior fue adaptado para tener una versión de precisión simple. Tras la comprobación de que los resultados producidos eran correctos se procedió a utilizar la última librería que formaría parte del proyecto: BLIS para el AMX, desarrollado por el investigador RuQing Xu (Xu (2021a,b)).

Tras tener todos los códigos de las pruebas, se ejecutaron las primeras pruebas de rendimiento y se comprobó que estas se solo se ejecutaban en los cores Firestorm. La siguiente tarea fue averiguar si se podían usar los cores Icestorm.

3.3. Acceso a los Cores

El trabajo entonces se centró en conseguir utilizar los cores Icestorm. Apple se caracteriza por tender a restringir el acceso a su sistema lo máximo posible y macOS no es una excepción. Mientras que en Linux encontramos la función *task_set*(), la cual permite elegir en qué unidades del procesador queremos ejecutar nuestros programas, en macOS esa función no existe ni hay equivalente. Instalando Linux en el Mac Mini M1 se podría utilizar esta funcionalidad, pero el único distro disponible públicamente ha sido publicado recientemente y es una distribución muy inmadura, ya que no funcionan muchos dispositivos del procesador (no se tiene acceso a ninguno de los aceleradores por el momento, por ejemplo).

Sin embargo, Apple no restringe del todo el acceso a los tipos de core, puesto que tiene en su sistema distintas clases de calidades de servicio (Apple (2022g)). Estas sirven para indicarle a macOS la prioridad que ha de tener el programa frente a los demás (por ejemplo,

una aplicación que interacciona constantemente con el usuario va a ser más prioritaria que una que hace copias de seguridad semanales en segundo plano, ya que su ejecución debería ser una tarea menor). A pesar de que en la documentación oficial de Apple se detallan seis calidades de servicio, en las pruebas se han utilizado las cuatro siguientes:

Tabla 3.1: Tabla de Clases de QoS

	Prioridad
CLASS_USER_INTERACTIVE	Máxima
CLASS_USER_INITIATED	Alta
CLASS_UTILITY	Media
CLASS_BACKGROUND	Baja

Las otras dos clases no incluídas son CLASS_DEFAULT y CLASS_UNSPECIFIED. Estas son casos especiales, ya que la primera es asignada automáticamente a un programa cuya QoS no ha sido indicada explícitamente y su prioridad se situaría entre Utility y User Initiated, es decir, se ejecutaría en los cores Firestorm. La segunda puede estar presente en alguna API antigua que indique al thread explícitamente que no ha de inferirse una QoS al thread. Por tanto, las opciones más atractivas para el proyecto son las mencionadas en la Tabla 3.1.

Cabe destacar que las calidades de servicio no garantizan que el código se vaya a ejecutar en un tipo de core u otro, aunque sí que indican al sistema en cuál se desea ejecutar. En las pruebas realizadas, todas las clases se ejecutan en los cores Firestorm salvo la clase Background, que se ejecuta en los cores Icestorm cuando la carga es ligera. Sin embargo, si la tarea a realizar usa más de un thread, el sistema comenzará a utilizar también los cores Firestorm. Esto se ha comprobado y se puede ver un análisis en detalle en la sección siguiente.

Adicionalmente se hizo una versión de cada código donde los tamaños de matrices eran fijos (en concreto $n = m = k = 16000$) y las operaciones se repetían cinco veces con 1, 2 y 4 threads. Esto se hace así para tener la certeza de que los resultados son fiables y comparables entre si.

3.4. División del Trabajo en Cores

La siguiente tarea del proyecto fue ver si se podía hacer uso tanto de Firestorm como de Icestorm a la vez para realizar la operación *gemm* simultáneamente. Esto se puede realizar creando dos threads donde a uno se le asigna calidad de servicio Background y al otro una calidad de servicio superior.

Para realizar la operación *gemm* de forma normal hay que pasarle a la función los punteros a las matrices A, B y C, indicarle los tamaños m, n y k y pasarle los escalares αy β (si solo queremos hacer AxB se pondría β a cero). Además, hay que indicarle el stride (elementos a los que se encuentra la siguiente columna, en este caso será m porque se utilizan matrices cuadradas) de cada matriz, y la función se encarga de hacer la operación.

Una idea para dividir el trabajo entre los cores es enviar a Firestorm una gran parte de

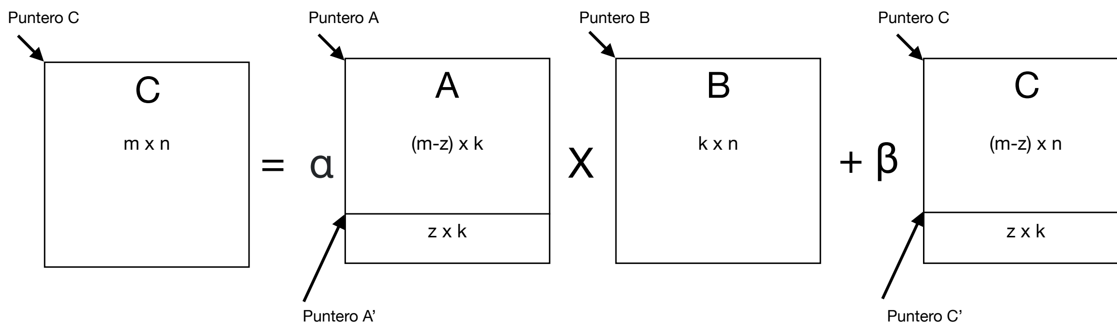

Figura 3.1: División de gemm en dos

la matriz A, pasarle B entera y pasarle la misma proporción de C que se pasa de A (ver Figura 3.1). Los strides son los mismos y solo hay que ajustar el tamaño m. A los cores Icestorm se les envía una porción pequeña de A y C (llamadas A' y C' en la Figura 3.1, aunque sigue siendo el mismo array de A y C) y se le envía B entera. También hay que ajustar el tamaño que se le envía a Icestorm. Los tamaños n y k se mantienen iguales, pero ahora m será z, que es, según los análisis, 1/11 de m. Las proporciones, en resumen, son las siguientes:

Matriz	Tamaño Original	Tamaño Tras División
A	m x k	(10/11)*m x k
B	k x n	k x n
C	m x n	(10/11)*m x k
A'	-	(1/11)*m x k
C'	-	(1/11)*m x k

Para el cálculo de la proporción, es suficiente con ver los GFLOPS que se obtiene de media en Firestorm y Icestorm y ver qué porcentaje de GFLOPS de Firestorm es capaz de hacer Icestorm. Por ejemplo, en *sgemm* , Accelerate obtiene de media 105 GFLOPS en los cores Icestorm frente a 850 GFLOPS de media en Firestorm. Combinados, ambos suman 955 GFLOPS de media, representando los 105 GFLOPS de Icestorm un 11 % del rendimiento total. Se podría utilizar este porcentaje (asignarle a Firestorm 9/10 del trabajo y el otro décimo a Icestorm) pero como no siempre rinden igual, es preferible asignarle algo menos a Icestorm (1/11 del trabajo, es decir, un 9 % en lugar del 11 % inicial) para evitar el posible caso de que Firestorm acabe antes y el rendimiento se vea empeorado porque se espera a la finalización de Icestorm.

Esta prueba se ha realizado en Accelerate y en el Capítulo 4 se muestran los análisis de uso de los dos tipos de core en las pruebas realizadas, así como un análisis del rendimiento obtenido.

3.5. Uso de los Microkernels de BLIS

Otra de las pruebas consiste en ejecutar las mismas pruebas de rendimiento *dgemm* en los microkernels disponibles para el AMX. La versión de BLIS que hace uso del AMX no es

la oficial. Por tanto, han quedado diversas pruebas del investigador RuQing Xu en él. Llama la atención que al invocar a estas funciones, BLIS utiliza un microkernel de 32x16 para *dgemm* , aunque queda otro microkernel para DGEMM de 16x16 que no se ha utilizado. Por tanto, la tarea de esta sección consiste en utilizar dichos microkernels y comparar su rendimiento con los demás, así como con BLIS.

La utilización de estos microkernels es sencilla, puesto que el trabajo se basa en eliminar algunos parámetros de las funciones que no se consideran necesarios y renombrar las propias funciones para evitar conflictos con la propia librería. En las pruebas se han utilizado los microkernels para ARMv8 y para el co-procesador AMX.

3.6. Uso de la GPU

La última tarea del proyecto consiste en comparar el rendimiento de Accelerate, que trabaja con la CPU y el AMX, frente al rendimiento de la GPU, que es accesible desde las distintas librerías de Metal (Apple (2022e)). Estas son más complejas de utilizar y no hay tanta información acerca de su uso. Además, se ha llegado a la conclusión de que las librerías de Metal están escritas en Swift y en Objective-C pero no en C. La propia documentación de Apple no incluye nada en C y en todos los artículos que tratan el uso de Metal el lenguaje utilizado es Objective-C o Swift. No es así con Accelerate, del que hay multitud de artículos acerca de su uso en C. Es por esto que las pruebas realizadas se han llevado a cabo en Swift, tanto en Accelerate como en Metal, para que los rendimientos puedan ser comparados.

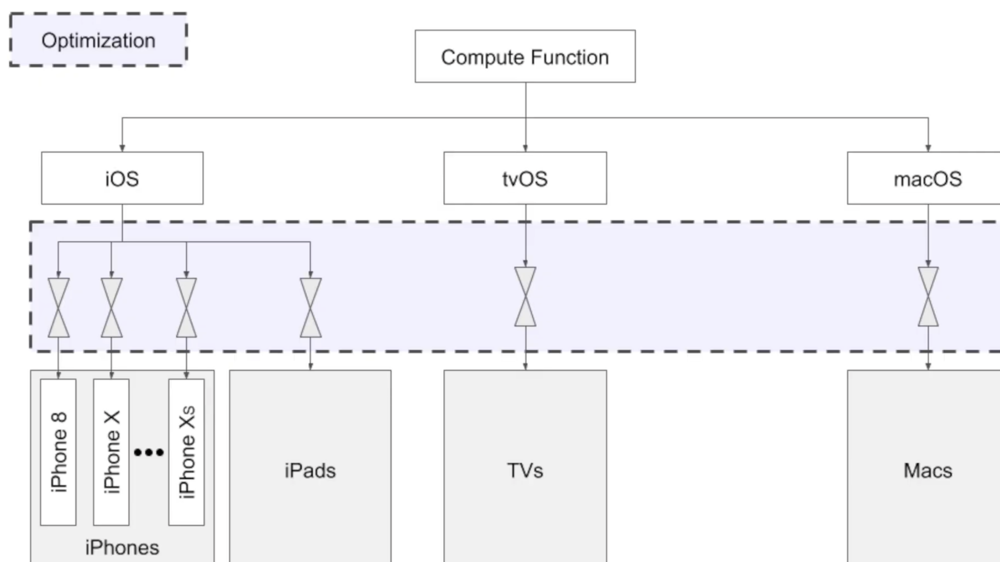

Figura 3.2: Optimización de la Funcionalidades por Dispositivo

Metal es una librería que reúne otras librerías que hacen uso de la GPU, encontrándose entre ellas Metal Performance Shaders (Apple (2022f)) , que es accesible en iOS, macOS, y tvOS. En la actualidad, MPS tiene cuatro funcionalidades: aplicación de filtros de alto rendimiento a imágenes, entrenar redes neuronales, trazado de rayos y álgebra lineal. Es esta

última aplicación la que se ha investigado al ser la que permite realizar cálculos matriciales.

Dentro de la librería se encuentran distintos comandos de computación que se pueden enviar a la GPU para ser procesados. Sin embargo, estos comandos son optimizados por la librería para cada dispositivo concreto (ver Figura 3.2) y son creados mediante kernels. Para cada funcionalidad de la GPU que se quiera usar hay un kernel específico. Por ejemplo, para utilizar la funcionalidad de multiplicación de matrices, que es la que concierne al proyecto, se ha de usar el kernel de MPSMatrixMultiplication. Estos comandos que se envían a la GPU son conocidos como Shaders.

Para que la CPU se pueda comunicar con la GPU (ver Figura 3.3), hay que crear un objeto Dispositivo (Device), que está adaptado a cada dispositivo en particular, y este es utilizado para crear una cola de comandos (Command Queue). Esta última se utiliza para crear un buffer de comandos (Command Buffer). A continuación, hay que crear un kernel desde la librería MPS que permite codificar comandos y enviarlos al buffer de comandos del dispositivo. Una vez enviados los comandos deseados al buffer, se hace un commit de estos y se ponen a la cola de comandos, que son tratados a continuación por la GPU.

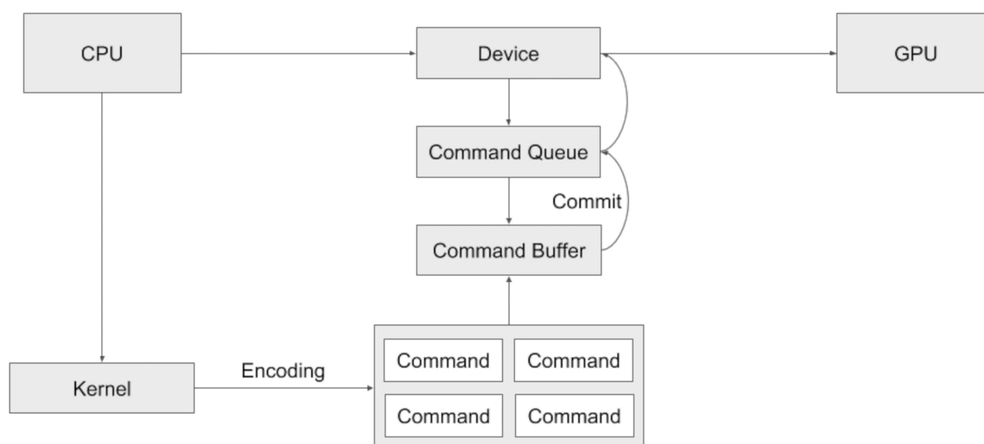

Figura 3.3: Uso de la GPU en macOS

Tras lograr acceder a la GPU a través de MPS, las pruebas consisten en realizar una comparativa de rendimiento y consumo en FP32. No han sido realizadas en FP64 puesto que Apple no soporta todavía FP64 en estás librerías.

Capítulo 4

Análisis de Resultados

En esta sección se analizan los resultados de todas las pruebas realizadas y se obtienen conclusiones en base a ellos. Primero se analizan los resultados de cada librería, tanto de rendimiento como de consumo. Posteriormente se hace una comparativa general entre ellas de rendimiento, consumo y eficiencia. A continuación se analizan los resultados de las diferentes implementaciones de BLIS, tanto la versión para ARMv8 como la versión para el AMX, comparándolas con las imitaciones que hacen uso directo de los diversos microkernels.

Posteriormente se analiza el rendimiento en Accelerate repartiendo la carga entre los cores Icestorm y Firestorm, tanto en consumo como en rendimiento.

Para finalizar, se analizan los rendimientos obtenidos con *sgemm* con Accelerate y MetalPerformanceShaders en Swift.

4.1. Rendimiento y Consumo de las Librerías

Las primeras pruebas van enfocadas a realizar las operaciones *sgemm* y *dgemm* en el M1 con 1, 2 y 4 threads. Todas las pruebas se han realizado teniendo la certeza de que el ordenador no estaba siendo utilizado para nada más que las pruebas de rendimiento y consumo.

Las pruebas se han realizado con un tamaño fijo $n = m = k = 16000$ para medir tanto la potencia consumida como el rendimiento, y se han repetido cinco veces seguidas con el mismo número de threads y calidad de servicio. Tras superar las pruebas, se aumenta el número de threads a y 2 y se repiten cinco veces, y este proceso se repite una última vez con 4 threads. Cuando estas pruebas acaban, se repite el mismo proceso utilizando una calidad de servicio superior. Esto es repetido hasta que se han probado todas las calidades de servicio.

Adicionalmente se han repetido todas las pruebas con tamaños iniciales de $m = n = k = 1000$, sumando de 5000 en 5000 hasta llegar a 31000 (inclusive). Esto se ha realizado para poder analizar el rendimiento de las diversas librerías frente diversos tamaños. En las librerías en las que ha sido necesario (BLIS, OpenBLAS y BLIS AMX) se ha realizado un análisis del consumo de energía y porcentaje de utilización de los dos tipos de cores con

calidad de servicio Background con 1, 2 y 4 threads con matrices de $m = n = k = 16000$. Se comprueba así que con más de un thread el planificador utiliza los cores Firestorm junto a los Firestorm en esta calidad de servicio.

4.1.1. Accelerate

La librería Accelerate es la primera que se tiene en cuenta puesto que se sabe que usa el co-procesador matricial AMX e interesa ver si es la mejor alternativa para aplicaciones de Deep Learning o hay otras alternativas mejores. Las pruebas ejecutadas sobre Accelerate son siempre con un solo hilo, y en este caso tiene sentido hablar de cores Icestorm (calidad de servicio Background) y cores Firestorm (calidades de servicio Utility, User Initiated y User Interactive).

La investigación para ver si Accelerate funciona single threaded ha sido extensa, puesto que Apple no menciona nada sobre si las funciones BLAS hacen uso del multithreading o no. Atendiendo a la documentación oficial de Apple, solo se menciona que algunas funciones de vDSP (la enfocadas al cálculo matricial) pueden llegar a utilizar multithreading (Apple (2022d)). Sin embargo, a pesar de que vDSP cubre muchas funcionalidades de BLAS, esta última difiere en algunos aspectos de vDSP. Aún así, teniendo ambas funcionalidades similares, se puede suponer que si el cálculo matricial de vDSP activa el multithreading, el cálculo matricial de BLAS también lo hace.

La suposición realizada anteriormente se confirma realizando pruebas sobre esta librería. Atendiendo a la página 7 del manual de Accelerate (ver *man* 7 *Accelerate*) el multithreading, en las funciones que lo requieran, será controlado por el Grand Central Dispatch (esta es una Utilidad del Sistema creada por la propia Apple (Apple (2022c)) que abstrae el uso de threads del usuario y elige el número de threads y las CPU en las que ejecutar las tareas en función de la carga de trabajo). Existe además una variable de entorno llamada 'VECLIB_MAXIMUM_THREADS'(Apple (2021a)) que sirve para poder limitar el número de threads que utiliza la librería. Sin embargo, tras las pruebas realizadas, esta variable no afecta a las funciones BLAS.

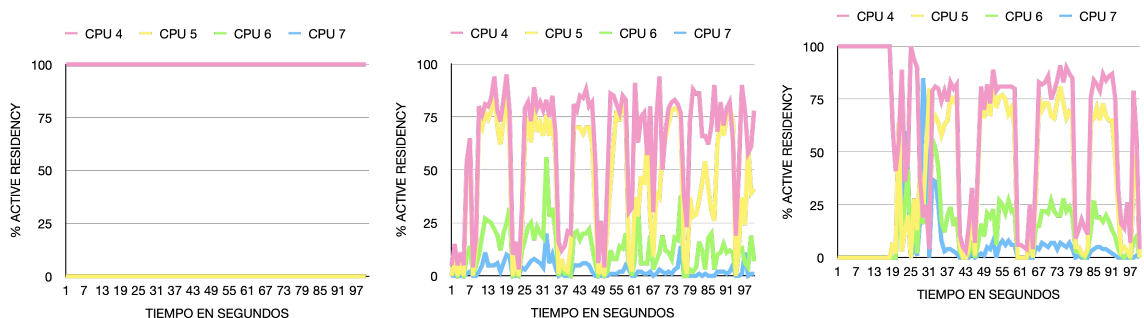

Figura 4.1: Uso de Firestorm con $n = m = k = 20000$

Figura 4.2: Uso de Firestorm con $n = m = k = 40000$

Figura 4.3: Uso de Firestorm con $n = m = k = 40000$ y limitando los threads a 1

Cuando no se limitan los threads con la variable de entorno 'VECLIB_MAXIMUM_THREADS'y se establece un tamaño $n = m = k = 20000$ (ver Figura 4.1), las operaciones se realizan en una única CPU de Firestorm. Sin embargo, cuando el tamaño se aumenta a

$n = m = k = 40000$ (ver Figura 4.2) se puede apreciar que el sistema usa las 4 CPU combinadas, y aún limitando los threads a 1 mediante variable de entorno (ver Figura 4.3) el sistema continúa utilizando 4 threads. Por tanto, el uso de threads que hace Accelerate no se puede controlar y depende de la implementación que Apple haya decidido realizar, donde el uso de threads depende en exclusiva del tamaño del problema y el sistema quien los controla.

Dado que las pruebas se han realizado con un tamaño $n = m = k = 16000$, Accelerate funciona single threaded en las pruebas y así queda reflejado en los análisis realizados en la presente sub-sección.

En las pruebas de precisión simple (ver Figura 4.4), Accelerate obtiene una media de 108 GFLOPS en Icestorm, 905 GFLOPS de media en calidad de servicio Utility y 917 GFLOPS de media tanto en calidad de servicio User Initiated como User Interactive. Se puede apreciar en estos números que las calidades de servicio donde la interacción con el usuario es alta dan algo más de rendimiento que la de utilidad.

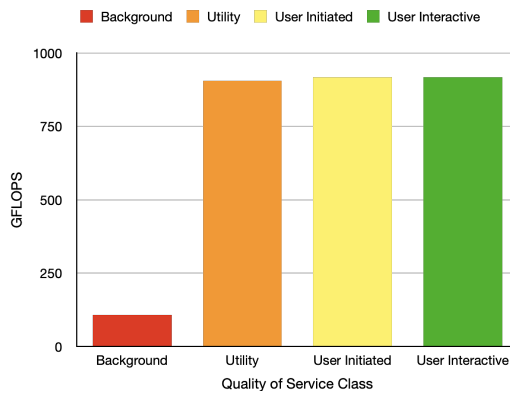

Figura 4.4: GFLOPS en *sgemm*

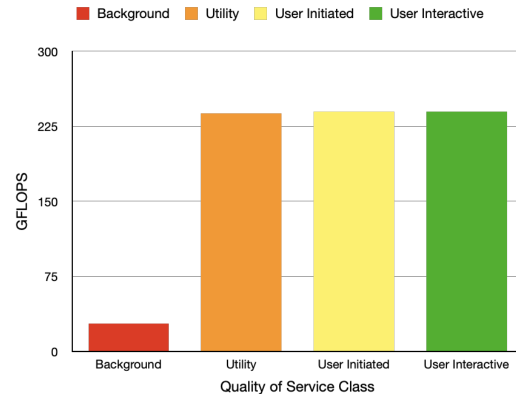

Figura 4.5: GFLOPS en *dgemm*

Respecto al rendimiento en precisión doble (ver Figura 4.5), Accelerate obtiene 28 GFLOPS de media en los cores Icestorm frente a los 240 GFLOPS que obtiene en Firestorm. Esta diferencia de rendimiento es normal puesto que el cálculo con doble precisión es más costoso que el de precisión simple. Es bastante común ver aplicaciones de Deep Learning utilizando precisión simple (o incluso menos) porque, aunque son menos precisas, son más rápidas y según el uso que interese hacer puede ser mejor opción usar menos precisión.

Atendiendo a la diferencia entre el rendimiento en precisión simple y doble, se puede apreciar que el rendimiento en precisión doble es, aproximadamente, un tercio del rendimiento obtenido en precisión simple. Esto indica que el AMX posiblemente haya sido diseñado para trabajar mejor en FP32 que en FP64, puesto que las librerías como BLIS (ARMv8) y OpenBLAS ven reducido su comportamiento a la mitad (comportamiento habitual) mientras que Accelerate y BLIS (AMX) (como se ve en una sub-sección posterior) ven reducido su rendimiento en torno a un tercio. Es decir, las librerías que hacen uso del co-procesador AMX rinden comparativamente mejor en precisión simple que en doble.

Otra característica apreciable (ver Figuras 4.6 y 4.7) es que Accelerate rinde mejor con matrices de tamaños pequeños (inferiores a $n = m = k = 26000$) que con grandes. Mientras

que con matrices de tamaño $n = m = k = 1000$ obtiene 1200 GFLOPS en precisión simple, esta cifra se ve reducida a la mitad cuando $n = m = k = 31000$. Esto puede indicar que Accelerate obtiene peor rendimiento con matrices de tamaños grandes por una peor gestión de memoria o por cómo ha sido diseñada la librería. Sin embargo, esto es una suposición, ya que Accelerate es una librería propietaria de la cual se desconoce su implementación y no se puede saber nada con certeza.

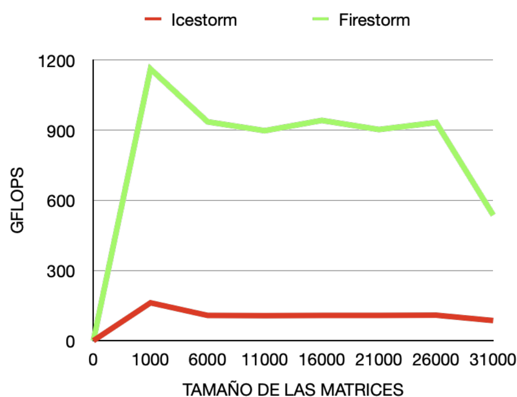

Figura 4.6: GFLOPS en *sgemm* por core

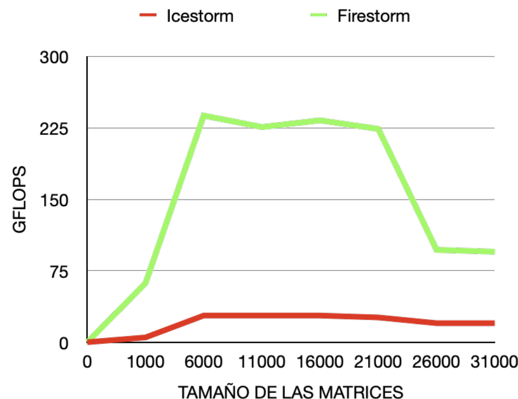

Figura 4.7: GFLOPS en *dgemm* por core

Analizando el consumo energético, Accelerate es de las librerías que menos consume y que mejor uso hace de la energía que consume. En precisión simple (ver Figura 4.8), los cores Icestorm consumen de media 208 mW, ascendiendo esta cifra a 4749 mW de media en la calidad de servicio Utility, 4979 mW en calidad de servicio User Initiated y 5060 mW de media en la calidad de servicio User Interactive. Cuando las pruebas se realizan en precisión doble (ver Figura 4.9) estas cifras aumentan, consumiendo 224 mW de media en Icestorm y 5400 mW de media en Firestorm. Este ligero aumento se debe a que el cálculo en FP32 consume menos energía que el cálculo en FP64.

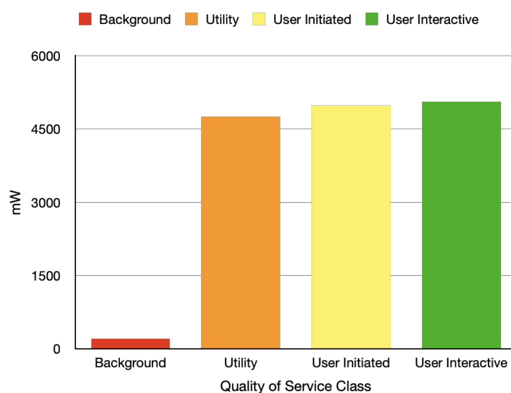

Figura 4.8: Consumo en *sgemm*

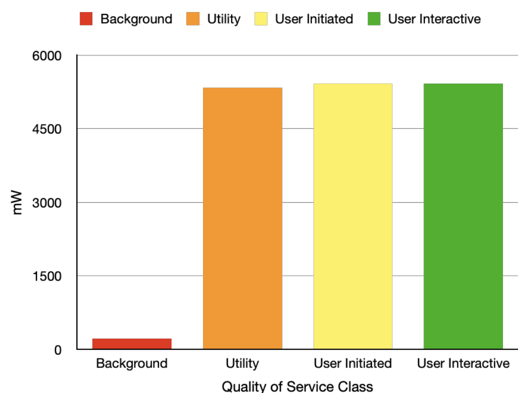

Figura 4.9: Consumo en *dgemm*

Estas cifras, que son bajas respecto al resto de librerías, corroboran la intención de Apple de crear una librería de gran rendimiento pero de bajo consumo al mismo tiempo. Esto se debe a que, al ser la librería propietaria del procesador, Apple conoce todos los detalles de este, estando en posición por tanto de hacer un diseño más eficiente. Esta librería es usada tanto en equipos macOS/tvOS, donde el consumo energético no es tan impor-

tante, como en equipos iOS/watchOS, donde el consumo energético es la máxima prioridad.

4.1.2. OpenBLAS

La librería de OpenBLAS no tiene acceso al co-procesador AMX y da unos resultados más bajos en rendimiento y más altos en consumo. Debido a que OpenBLAS no tiene las instrucciones de acceso al AMX, todos los cálculos han de hacerse en la CPU (de uso general), mientras que las librerías que pueden acceder a este co-procesador (cuyo uso está pensado explícitamente para realizar cálculos matriciales) rinden mejor y consumen menos.

En las pruebas de rendimiento de precisión simple (ver Figura 4.7) se puede apreciar la diferencia que hay entre la calidad de servicio Background, que usa Icestorm junto a Firestorm a baja frecuencia, frente al resto de calidades de servicio que hacen uso pleno de los cores Firestorm. La calidad Background comienza obteniendo 15 GFLOPS de media con un thread, y a medida que se doblan los threads, el rendimiento aumenta proporcionalmente. En el resto de calidades de servicio, comienzan obteniendo de media 98 GFLOPS con un thread, 194 GFLOPS de media con dos threads y 354 GFLOPS de media con 4 threads.

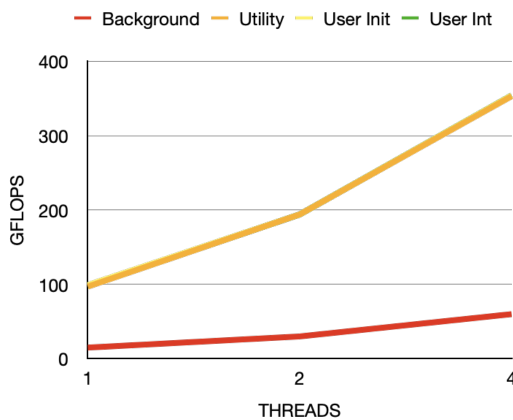

Figura 4.10: GFLOPS en *sgemm*

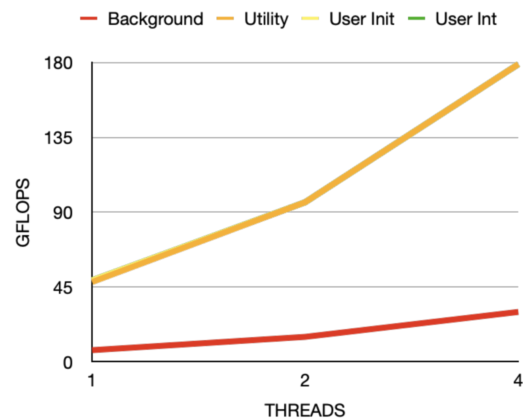

Figura 4.11: GFLOPS en *dgemm*

Respecto a las pruebas de precisión doble (ver Figura 4.8) se puede apreciar también esta diferencia entre calidades de servicio. La calidad Background obtiene 7 GFLOPS de media con un thread en Icestorm, y este rendimiento aumenta proporcionalmente al número de threads utilizados, alcanzando los 30 GFLOPS de media con 4 threads . Las calidades que usan los Firestorm a pleno rendimiento comienzan obteniendo 48 GFLOPS de media con un thread, 98 GFLOPS de media con dos y obtienen algo menos del doble con 4 threads (179 GFLOPS de media).

Ante matrices grandes (ver Figuras 4.9 y 4.10), OpenBLAS obtiene el mismo rendimiento que ante matrices pequeñas. Esto supone una diferencia respecto a Accelerate, que rinde mejor ante tamaños menores. Una posibilidad es que OpenBLAS haga una mejor gestión de la memoria y sea capaz de suministrar un rendimiento más constante que Accelerate, aunque no se puede saber debido a que la implementación de Accelerate no es conocida.

Las cifras de rendimiento obtenidas coinciden con las obtenidas por el investigador Ru-Qing Xu (Xu (2021a)).

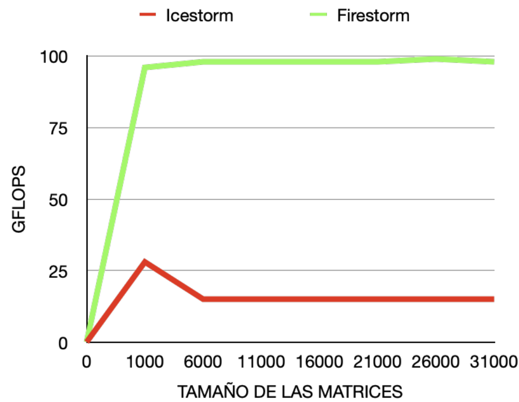

Figura 4.12: Rendimiento en *sgemm*

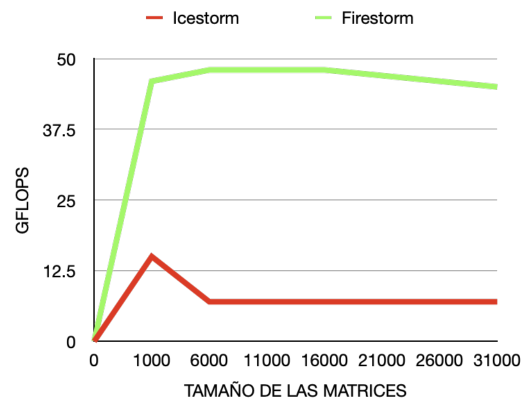

Figura 4.13: Rendimiento en *dgemm*

Analizando el consumo energético, en las pruebas en precisión simple (ver Figura 4.11), la calidad de servicio Background consume 176 mW de media con un thread y se eleva hasta los 3638 mW y 9235 mW de media con 2 y 4 threads respectivamente, mientras que las otras tres calidades de servicio parten de un consumo medio de 7300 mW con un thread y este se incrementa a una media de 12600 mW con 2 threads y 20500 mW con 4 threads.

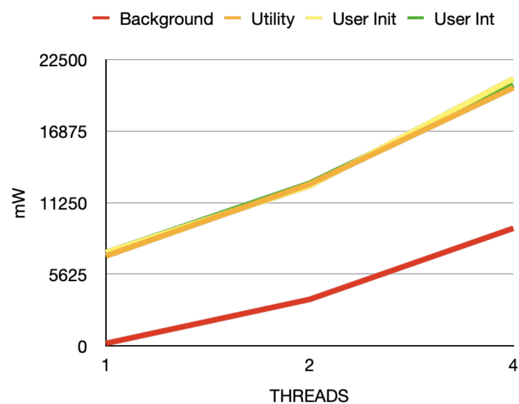

Figura 4.14: Consumo en *sgemm*

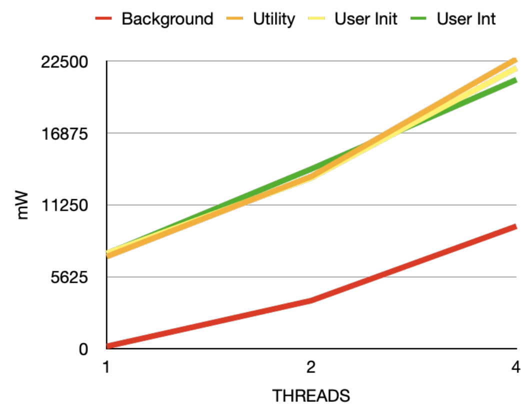

Figura 4.15: Consumo en *dgemm*

Cuando las pruebas se ejecutan en precisión doble, el consumo medio de potencia (ver Figura 4.12) comienza en 172 mW de media en QoS Background y aumenta a 3762 mW cuando entran en funcionamiento los cores Firestorm con dos threads, ascendiendo a 9235 mW de media cuando se usan 4 threads. Respecto a las otras tres calidades de servicio, se comportan de forma similar que en precisión simple: parten de 7300 mW de consumo medio con un thread, que aumenta a 13500 mW de media con dos threads y alcanzan 21500 mW de media con 4 threads.

En estas pruebas es donde, utilizando *powermetrics*, se descubren algunas anomalías que introduce el planificador de macOS. Cuando las pruebas se ejecutan en Background con 1 thread, el planificador va alternando dicho thread entre los diferentes cores del Icestorm. Sin embargo, cuando se utilizan dos threads, uno de ellos se ejecuta en Icestorm (con

el mismo comportamiento de alternamiento entre cores) y el otro se ejecuta en un core Firestorm. Cuando son 4 los threads utilizados, un thread se envía a los cores Icestorm, donde se alterna entre ellos, otro thread se ejecuta en un core Firestorm continuamente y los otros dos threads se van alternando entre otros el resto de cores Firestorm.

Lo mencionado previamente se puede ver analizando el porcentaje de uso de las 4 CPU Icestorm con 1, 2 y 4 threads en precisión simple (ver Figuras 4.13, 4.16 y 4.19 respectivamente), donde se puede apreciar que el uso medio de cada CPU está en torno al 30 %, lo que indica que el thread se va alternando entre los cores Icestorm. Se desconoce por qué el planificador realiza las tareas así y no hay información al respecto. Atendiendo al porcentaje de uso de las CPU Firestorm con 1, 2 y 4 threads (ver Figuras 4.14, 4.17 y 4.20) se ve que con un thread, Firestorm no se llega a usar, mientras que con 2 y 4 threads sí. Sin embargo, con dos threads el planificador decide usar un core de Firestorm mantener el resto apagados. Cuando los threads aumentan a 4, el planificador decide usar un core Firestorm para un thread y va alternando los otros tres cores para los otros dos threads.

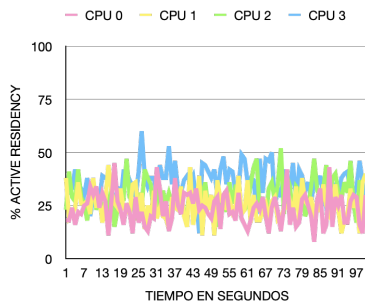

Figura 4.16: Uso de Icestorm con 1 thread

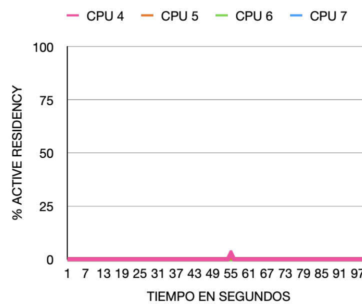

Figura 4.17: Uso de Firestorm con 1 thread

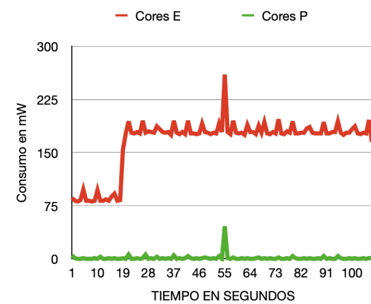

Figura 4.18: Consumo en 1 thread

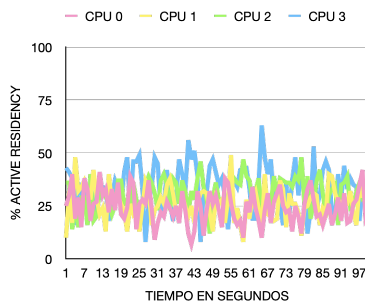

Figura 4.19: Uso de Icestorm con 2 thread

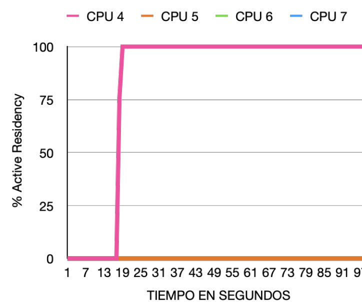

Figura 4.20: Uso de Firestorm con 2 thread

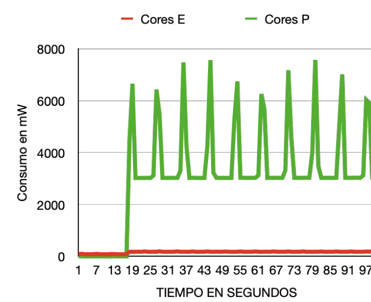

Figura 4.21: Consumo en 2 thread

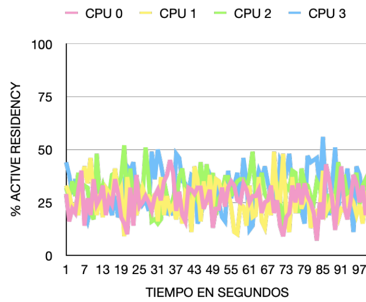

Figura 4.22: Uso de Icestorm con 4 thread

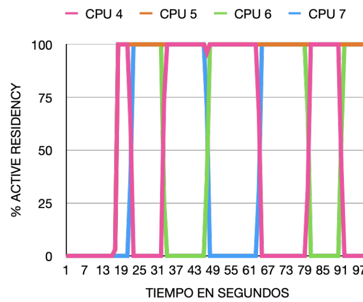

Figura 4.23: Uso de Firestorm con 4 thread

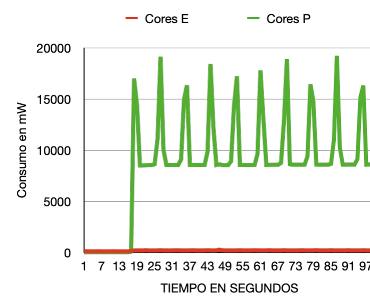

Figura 4.24: Consumo en 4 thread

Como consecuencia de este uso de Firestorm, el consumo de potencia se eleva considerablemente. Mientras que con un thread el consumo de potencia es muy bajo (225 mW de media por parte de Icestorm), este se eleva a una media de 3600 mW de media en total con 2 threads y 9235 mW de media con cuatro threads.

El comportamiento del sistema con las pruebas de precisión doble es muy similar al comportamiento en precisión simple. Se aprecia que el porcentaje de uso de Icestorm (ver Figura 4.22, 4.25 y 4.28) es también alrededor del 30 % en cada CPU Icestorm, independientemente del número de threads que se utilizan. Cuando se analiza el porcentaje de uso de los cores Firestorm con 1, 2 y 4 threads (ver Figuras 4.23, 4.26 y 4.29) se puede ver que el comportamiento es el mismo que con precisión simple también, es decir, los Firestorm permanecen apagados con 1 thread, mientras que con 2, se utiliza una CPU y las otras tres permanecen apagadas. Cuando se utilizan 4 threads, a diferencia de precisión simple (donde hay una CPU encendida constantemente y las otras 3 se van alternando), el planificador tiende a usar dos CPU a la vez que va alternando con las otras dos CPU.

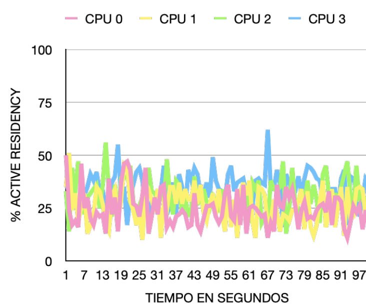

Figura 4.25: Uso de Icestorm con 1 thread

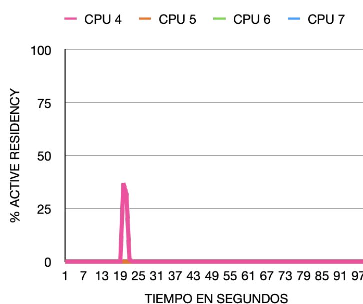

Figura 4.26: Uso de Firestorm con 1 thread

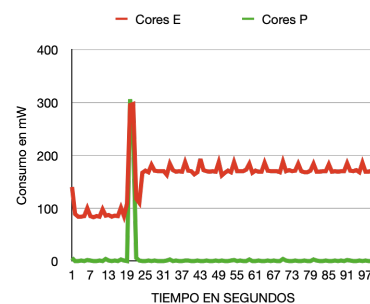

Figura 4.27: Consumo en 1 thread

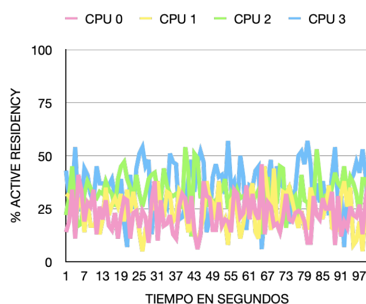

Figura 4.28: Uso de Icestorm con 2 threads

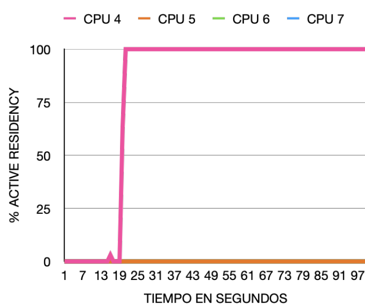

Figura 4.29: Uso de Firestorm con 2 threads

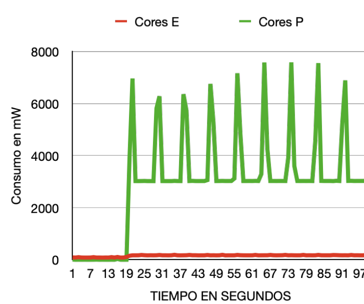

Figura 4.30: Consumo en 2 threads

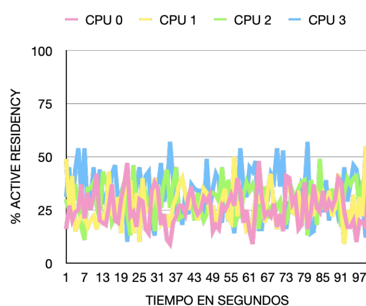

Figura 4.31: Uso de Icestorm con 4 thread

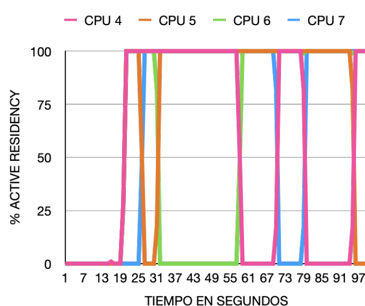

Figura 4.32: Uso de Firestorm con 4 thread

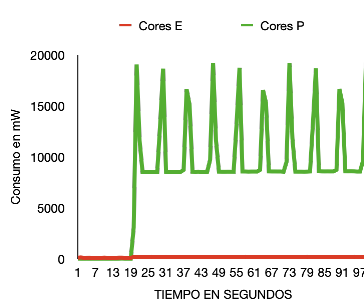

Figura 4.33: Consumo en 4 thread

4.1.3. BLIS

Las pruebas de rendimiento de la librería BLIS se han ejecutado tanto en BLIS para ARMv8 como en BLIS para el co-procesador AMX, existiendo grandes diferencias entre el rendimiento obtenido en una y otra. La versión de BLIS para ARMv8 obtiene unos resultados similares a los de OpenBLAS en términos de rendimiento y mejores en términos de consumo. Sin embargo, la versión para el AMX es la que mejor rinde de las cuatro, como se analiza en la presente sub-sección.

En las pruebas de precisión simple (ver Figura 4.34), BLIS ARMv8 obtiene 15 GFLOPS de media con un thread en calidad de servicio Background, 30 GFLOPS de media con dos threads y 59 GFLOPS de media con 4. Esto contrasta con BLIS para el AMX, que comienza en 114 GFLOPS de media con un thread, ascendiendo a 227 GFLOPS con dos threads y alcanzando los 452 GFLOPS de media con 4 threads. Cuando se usan las tres calidades de servicio superiores a Background (es decir, Utility, User Initiated y User Interactive), BLIS ARMv8 comienza obteniendo una media de 98 GFLOPS con un thread, que incrementa a 190 GFLOPS de media con 2 threads y alcanza los 361 GFLOPS con 4 threads. Cuando estas pruebas son ejecutadas en BLIS AMX con las tres calidades de servicio más altas, esta librería obtiene una media de 1030 GFLOPS, independientemente de los threads que se utilicen.

Se observa que BLIS AMX supera a BLIS ARMv8 (utilizando calidades de servicio superiores) con calidad de servicio Background (es decir, la menos prioritaria y que menos rendimiento obtiene). Esta diferencia de rendimiento es consecuencia del uso del AMX,

puesto que al ser un co-procesador diseñado específicamente para el cálculo matricial, el rendimiento que se obtiene es más alto que sin usarlo.

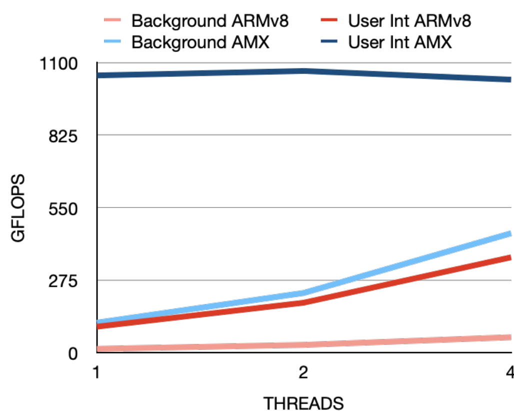

Figura 4.34: GFLOPS en *sgemm*

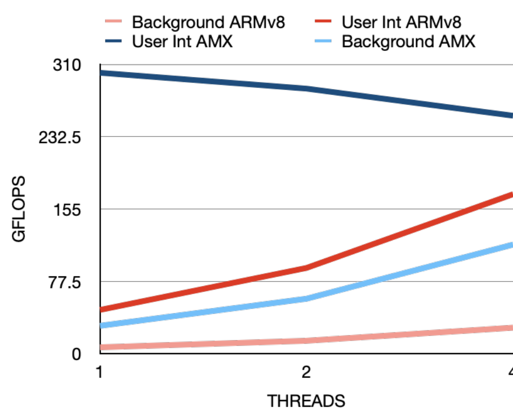

Figura 4.35: GFLOPS en *dgemm*

Respecto a los resultados obtenidos en precisión doble (ver Figura 4.35), la diferencia entre la versión AMX de BLIS y la versión ARMv8 parece disminuir en las calidades de servicio superiores. La versión para el AMX obtiene de media 240 GFLOPS frente a los 155 GFLOPS que obtiene BLIS ARMv8 con 4 threads. Se puede suponer que aumentando el número de threads de este último, superará en rendimiento a BLIS para el AMX.

Cuando la versión ARMv8 pasa de precisión simple a doble, su rendimiento se ve disminuido a la mitad. Sin embargo, la versión AMX ve disminuido su rendimiento a casi un cuarto cuando pasa a operar en precisión doble. Esto es posiblemente debido al co-procesador AMX, que trabajará mejor en FP32 que en FP64. Este mismo comportamiento también se ve en Accelerate, que se sabe que hace uso del AMX.

Atendiendo al rendimiento de ambas versiones de la librería frente a diversos tamaños de matrices (ver Figuras 4.36 y 4.37, donde se comparan ambas versiones de la librería con un solo thread), se puede apreciar que el rendimiento que da el AMX, especialmente en single thread, es muy bueno. Sin embargo, mientras que en BLIS ARMv8 mantiene un rendimiento constante, BLIS AMX sufre el mismo problema que Accelerate ante tamaños, es decir, su rendimiento decae con matrices de tamaños superiores a 21000x21000. Por tanto, al verse el mismo comportamiento que en Accelerate y al usar ambas librerías el AMX, se puede suponer que esto sea debido a algún aspecto de diseño del co-procesador y no de las librerías.

En lo que al consumo respecta (ver Figuras 4.38 y 4.39), ambas librerías consumen más o menos lo mismo tanto en precisión simple como en doble. Esto puede ser debido a que el sistema correrá los procesos a la misma frecuencia, sin importar si es precisión simple o doble. En ningún caso la versión para ARMv8 consume menos potencia que la versión para el AMX. Esto tiene sentido ya que el AMX, al estar diseñado para realizar una tarea específica, siempre va a consumir menos, por lo que la diferencia de consumo entre BLIS ARMv8 y BLIS AMX es normal.

Una de las conclusiones que se saca tras el análisis de BLIS y OpenBLAS es que ambas obtienen peor rendimiento y peor consumo debido a que no tienen acceso al AMX, aunque

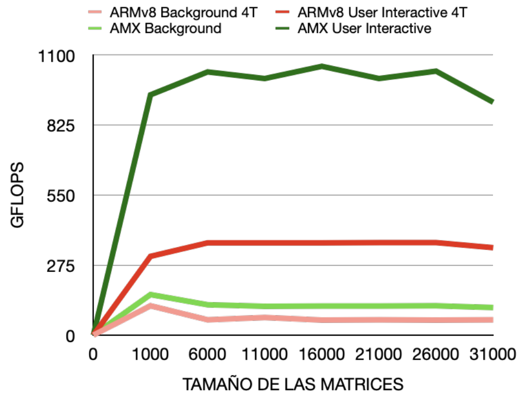

Figura 4.36: Rendimiento en *sgemm*

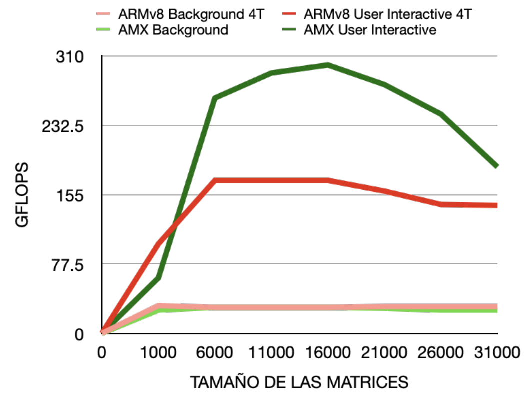

Figura 4.37: Rendimiento en *dgemm*

su rendimiento puede aumentar con más threads.

Haciendo un análisis más detallado del consumo de energía cuando se utiliza la calidad de servicio Background, se puede ver (Figuras 4.40, 4.43 y 4.46) que, independientemente del número de threads, el planificador siempre va a ir alternando un thread por los diferentes cores Icestorm. En los cores Firestorm (ver Figuras 4.41, 4.44 y 4.47), con un thread todas las CPU se mantienen apagadas. Sin embargo, cuando se comienzan a utilizar dos threads o más, se observa el mismo comportamiento que en OpenBLAS, es decir, el planificador mantiene un thread en un core Firestorm y, en caso de haber más threads, los va alternando por los otros cores disponibles de alta potencia.

Las anomalías introducidas por el planificador se ven reflejadas en el consumo, que es muy elevado en cuanto los cores Firestorm comienzan a funcionar, a pesar de suministrar el mismo rendimiento que los cores Icestorm.

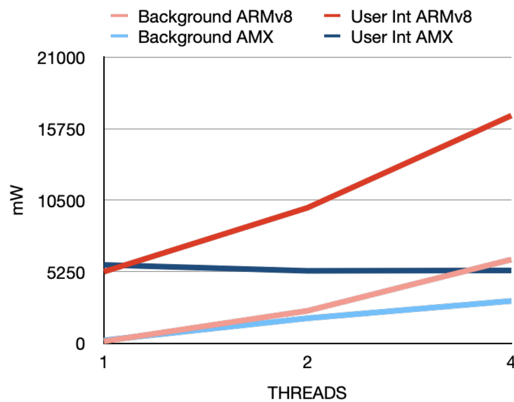

Figura 4.38: Consumo en *sgemm*

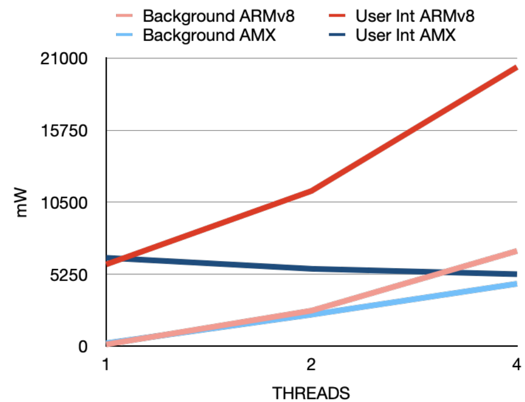

Figura 4.39: Consumo en *dgemm*

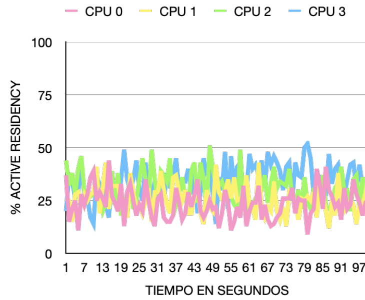

Figura 4.40: Uso de Icestorm con 1 thread

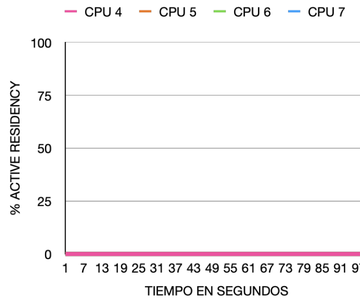

Figura 4.41: Uso de Firestorm con 1 thread

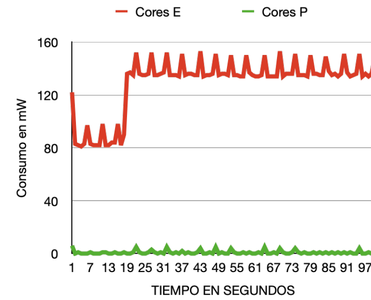

Figura 4.42: Consumo en 1 thread

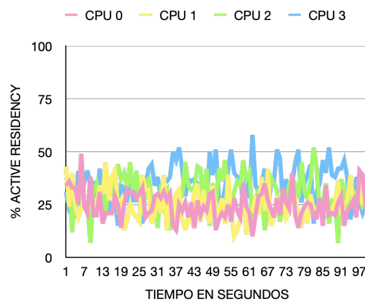

Figura 4.43: Uso de Icestorm con 2 thread

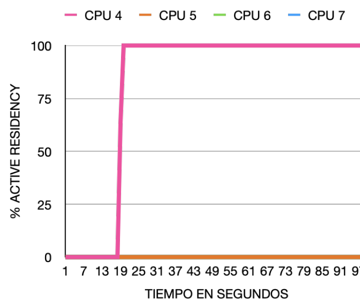

Figura 4.44: Uso de Firestorm con 2 thread

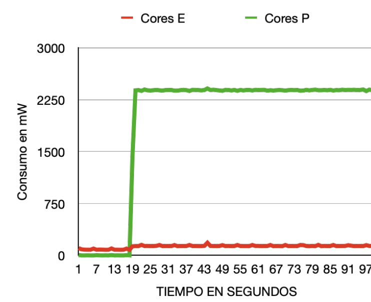

Figura 4.45: Consumo en 2 thread

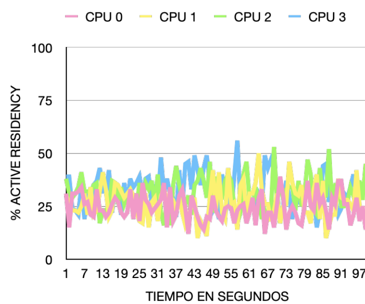

Figura 4.46: Uso de Icestorm con 4 thread

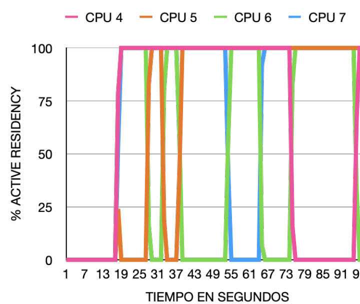

Figura 4.47: Uso de Firestorm con 4 thread

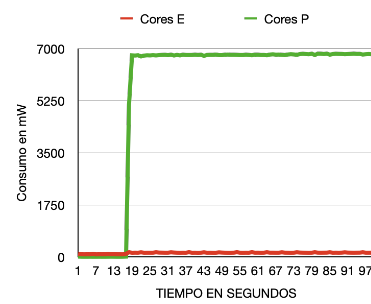

Figura 4.48: Consumo en 4 thread

4.2. Comparativa General

Tras haber analizado cada librería individualmente, uno de los objetivos del proyecto es analizar el rendimiento de cada librería frente a las demás y determinar si hay alternativas mejores a Accelerate, tanto en términos de rendimiento como de consumo. Además, es importante analizar la eficiencia de cada librería, puesto que según la aplicación que se desarrolle y la plataforma en la que se ejecute puede no ser demasiado útil una librería que obtiene resultados buenos de rendimiento pero consume demasiado.

Para la representación del rendimiento se han usado las calidades de servicio Background y User Interactive con un thread para Accelerate y BLIS AMX, y con 4 threads

para OpenBLAS y BLIS ARMv8. Es decir, se van a tratar de obtener todas las capacidades que pueden proporcionar tanto el procesador como el acelerador. En precisión simple (ver Figuras 4.49 y 4.50) las dos librerías que hacen uso del AMX (Accelerate y BLIS para el AMX) obtienen, de media, dos veces más GFLOPS que las librerías que no hacen uso de él. Las cifras obtenidas en las pruebas son muy similares a las obtenidas por el investigador RuQing Xu, tanto para Accelerate como para BLIS AMX y OpenBLAS (Xu (2021a)).

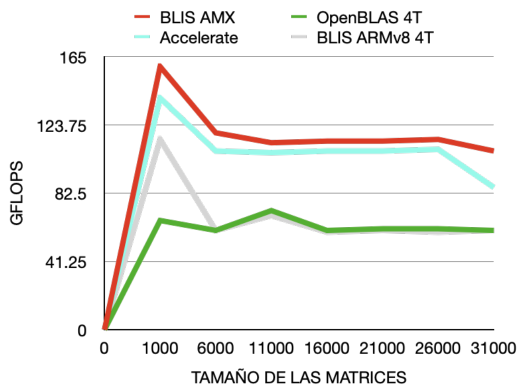

Figura 4.49: *sgemm* en Background

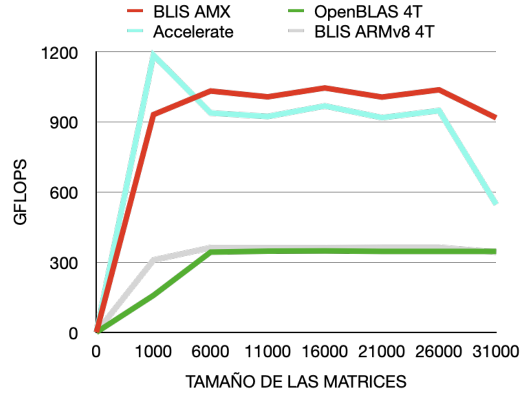

Figura 4.50: *sgemm* en User Interactive

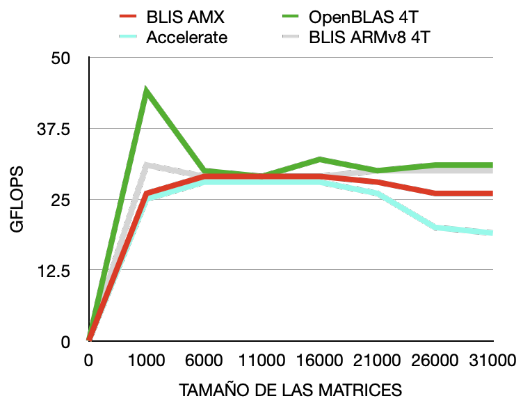

Figura 4.51: *dgemm* en Background

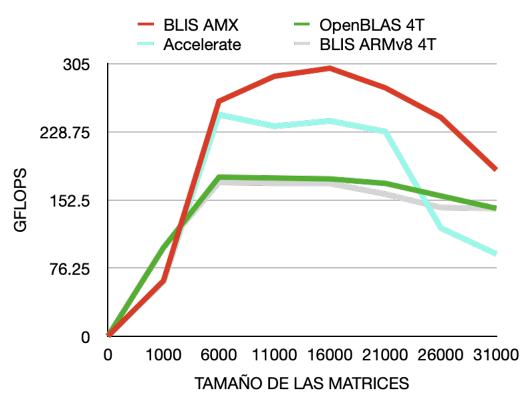

Figura 4.52: *dgemm* en User Interactive

Analizando el comportamiento de las diferentes librerías ante diversos tamaños de matrices (ver Figuras 4.49 y 4.50 para precisión simple y Figuras 4.51 y 4.52 para precisión doble), se puede ver que las librerías que hacen uso del AMX ven reducido su rendimiento a medida que el tamaño de las matrices crece, no siendo así en las librerías que no usan este co-procesador. Además, el paso de FP32 a FP64 trae consigo una reducción de rendimiento de cerca de un 75 % para las Accelerate y BLIS AMX, mientras que esa reducción de rendimiento es del 50 % (lo habitual en este tipo de pruebas) en BLIS ARMv8 y OpenBLAS.

Como se menciona previamente, estas diferencias son causadas por aspectos de diseño del AMX, y aquella librería que haga uso de este co-procesador obtendrá resultados similares (tanto de reducción de rendimiento como de comportamiento ante matrices grandes).

Respecto al consumo (ver Figuras 4.53 y 4.54 para precisión simple y Figuras 4.55 y 4.56 para precisión doble) queda patente que las librerías que usan el co-procesador matricial

consumen menos (al estar este diseñado para realizar una tarea específica, es decir, cálculo matricial) que aquellas que usan las CPU disponibles (que son de propósito general). En las gráficas adjuntas se ha decidido incluir Accelerate a pesar de utilizar solo un thread. Esto se ha realizado así para poder comparar con esta librería, pero para los valores de 2 y 4 thread de Accelerate se han usado los mismos que con un thread (puesto que no existen datos con 2 y 4 threads).

Tanto en precisión simple como en precisión doble, en las calidades de servicio superiores, Accelerate y BLIS AMX tienen un consumo constante. Esto es debido al uso del co-procesador. Sin embargo, cuando se utiliza calidad de servicio Background, BLIS AMX sí parece aumentar su consumo. Esto es debido a que, hasta que no se alcance la cantidad de datos máxima que puede tratar el AMX (que posiblemente sea la que pueda suministrar un core Firestorm a máxima frecuencia), esta librería tiene margen de aumento.

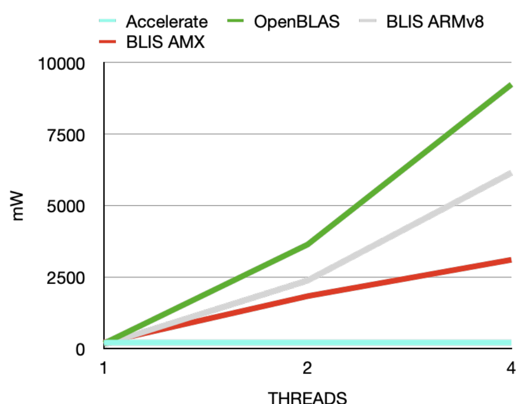

Figura 4.53: Consumo *sgemm* en Background

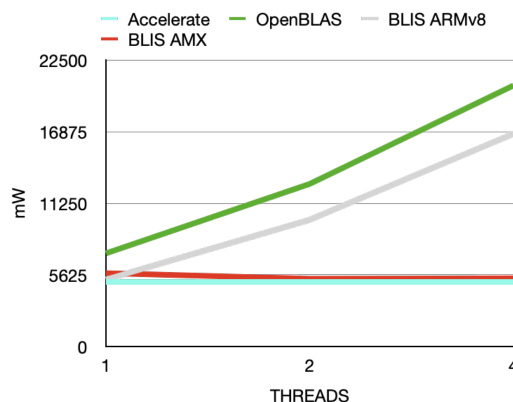

Figura 4.54: Consumo *sgemm* en User Interactive

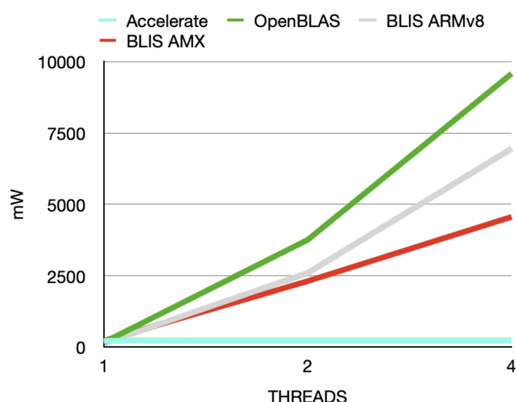

Figura 4.55: Consumo *dgemm* en Background

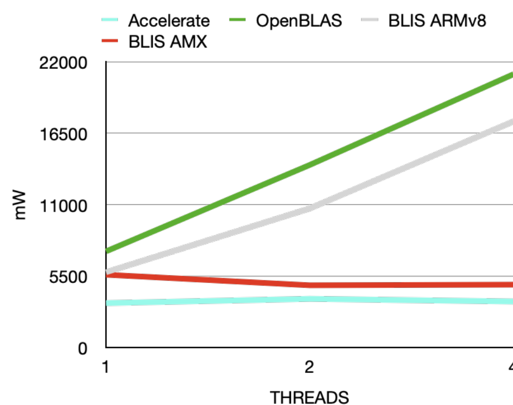

Figura 4.56: Consumo *dgemm* en User Interactive

El comportamiento de BLIS ARMv8 y OpenBLAS es el esperado, es decir, a mayor número de threads, mayor consumo. El consumo máximo alcanzado por una librería es de OpenBLAS, que alcanza los 21000 mW de media con 4 threads en precisión simple. De este consumo dista BLIS, que obtiene cifras de consumo inferiores obteniendo más o menos el mismo rendimiento.

Otro de los intereses del proyecto es, teniendo los resultados de rendimiento y consumo, analizar la eficiencia de las librerías, es decir, analizar qué librerías obtienen más GFLOPS por mW consumido. Este análisis puede ser muy útil si se necesita desarrollar una aplicación cuya prioridad sea el consumo de energía, puesto que determinará qué librerías hacen un mejor uso de la energía que consumen.

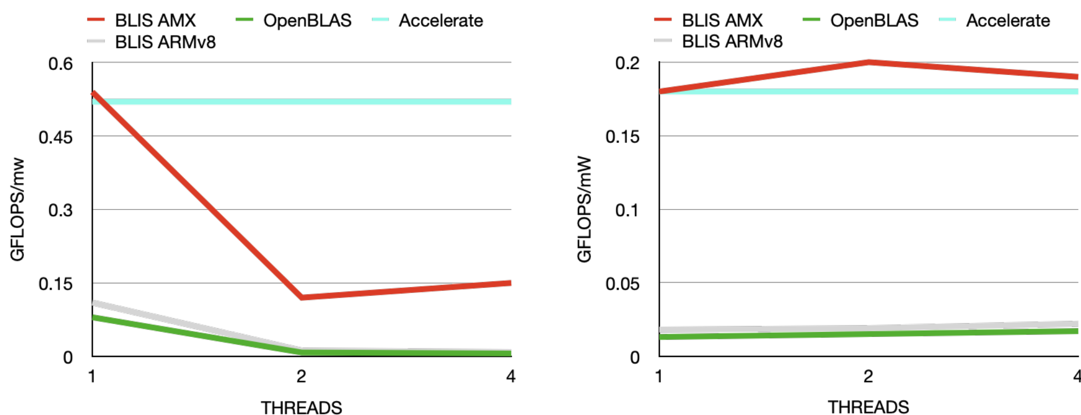

Figura 4.57: Efiicencia *sgemm* en Background

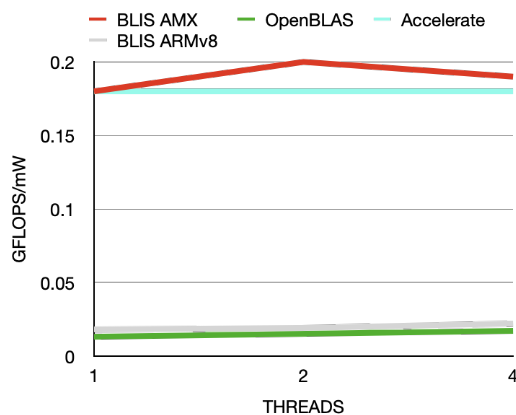

Figura 4.58: Eficiencia *sgemm* en User Interactive

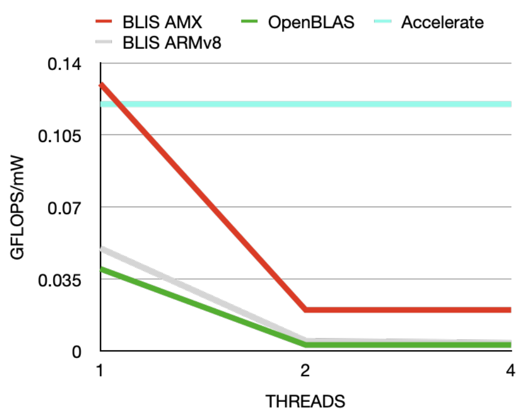

Figura 4.59: Eficiencia *dgemm* en Background

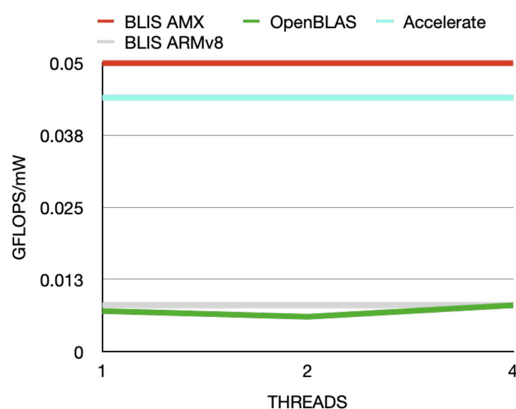

Figura 4.60: Eficiencia *dgemm* en User Interactive

Tanto en el análisis en precisión simple como de precisión doble (ver Figuras 4.57 y 4.59 respectivamente) en la calidad de servicio Background se puede apreciar (a excepción de Accelerate que corre en todo momento single threaded) la activación de Firestorm con dos o más threads, ya que a pesar de obtener más rendimiento, el consumo por core Firestorm es muy elevado. Esto provoca una bajada brusca de eficiencia. El rendimiento que reporta el core Firestorm es similar (en esta calidad de servicio) al que reporta el Icestorm pero consumiendo mucho más.

Para las calidades de servicio superior, a excepción de Accelerate, la eficiencia se mantiene muy similar con 1, 2 y 4 threads. A pesar de que en las gráficas apenas es discernible, tanto BLIS como OpenBLAS mejoran su eficiencia ligeramente. Esto se debe a que al

multiplicar por dos los threads, el rendimiento mejora el doble. Sin embargo, el consumo no es proporcional. Un ejemplo de esto se puede ver en la gráfica 4.56, donde el consumo de OpenBLAS con 2 threads es de 14000 mW de media frente a los 21000 mW que consume de media con 4 threads, es decir, no es proporcional al número de threads utilizados. Además, BLIS AMX es algo más eficiente que Accelerate, puesto que obtiene algo más de rendimiento consumiendo más o menos la misma energía.

4.3. Microkernels de BLIS

La librería BLIS se compone de multitud de microkernels adaptados para arquitecturas concretas. Entre ellas encontramos microkernels para Haswell, Sandy Bridge, Power 7, Power 8, etc. Oficialmente, BLIS no tiene un microkernel específico para el M1, pero sí que tiene un microkernel para ARMv8, que mejora bastante los resultados frente a una *sgemm* / *dgemm* realizada de forma genérica (es decir, sin optimización alguna).

En lugar de usar directamente la librería, una de las pruebas del proyecto es utilizar directamente este microkernel de ARMv8 en códigos simplificados que permiten una más sencilla modificación ya que no habría que modificar una librería completa. El objetivo final de estos códigos lo veremos más adelante cuando trataremos de utilizar de forma conjunto tanto los cores Icestorm como los Firestorm, pero antes de ello se quiere comparar el rendimiento de estos códigos simplificados con respecto a la librería.

El rendimiento obtenido en precisión simple (ver Figura 4.61) para la calidad de servicio Background parte de 13 GFLOPS con 1 thread, que se multiplica por dos a medida que también lo hacen el número de threads, alcanzando casi 60 GFLOPS. Cuando se utilizan las otras calidades de servicio, el rendimiento asciende a 87 GFLOPS, que también incrementan de forma proporcional al aumento de threads, alcanzando un máximo de 266 GFLOPS con 4 threads.

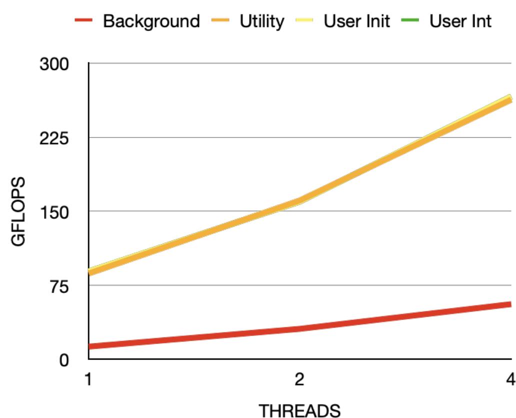

Figura 4.61: Rendimiento en *sgemm*

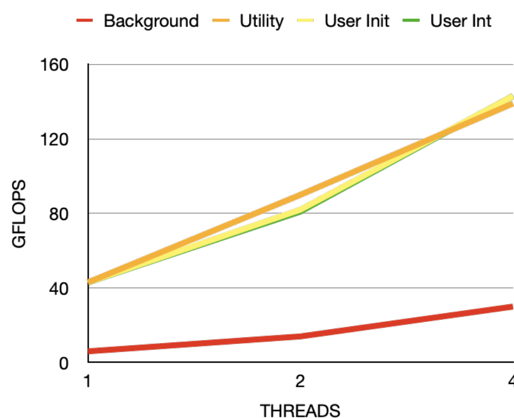

Figura 4.62: Rendimiento en *dgemm*

Cuando se ejecutan las pruebas de precisión doble (ver Figura 4.62), las cifras se reducen en torno a la mitad, obteniendo 6 GFLOPS en Background con 1 thread y aumentando proporcionalmente al aumento de threads. Para las otras tres calidades de servicio se obtienen 42 GFLOPS de media con 1 thread, que aumentan hasta obtener una media de 150

GFLOPS de rendimiento con 4 threads.

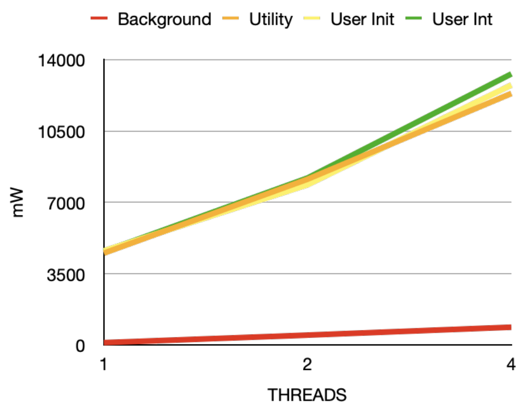

Figura 4.63: Consumo en *sgemm*

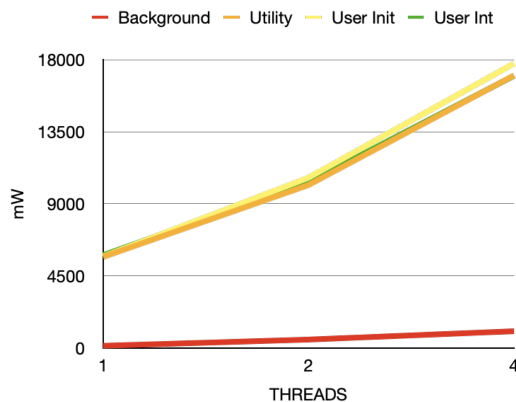

Figura 4.64: Consumo en *dgemm*

Respecto al consumo (ver Figuras 4.63 y 4.64) sucede lo mismo que al utilizar de forma normal la librería de BLIS, es decir, el consumo aumenta (pero no proporcionalmente a los threads) cuando se aumentan los hilos, y obtiene unos resultados muy similares a la propia librería.

Cuando se comparan el mejor y el peor rendimiento de BLIS (es decir, calidad de servicio Background y calidad de servicio User Interactive) con el mejor y el peor rendimiento de la replica que hace uso directo del microkernel, se puede apreciar (ver Figuras 4.65 y 4.66) que el rendimiento no dista demasiado entre las dos versiones. Sin embargo, la diferencia existente entre el código simplificado y la propia librería es que esta genera los hilos de manera más eficiente que la implementación simplificada. BLIS los genera al comienzo de cada rutina y usa los mismos durante toda la ejecución, mientras que los códigos simplificados crean y destruyen hilos en los diferentes bucles for. Por tanto, es una cuestión de escalado.

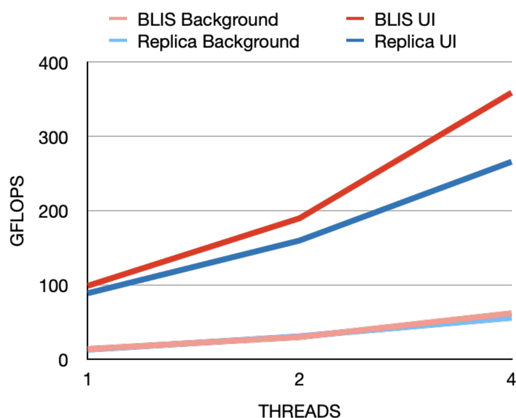

Figura 4.65: Comparativa en *sgemm*

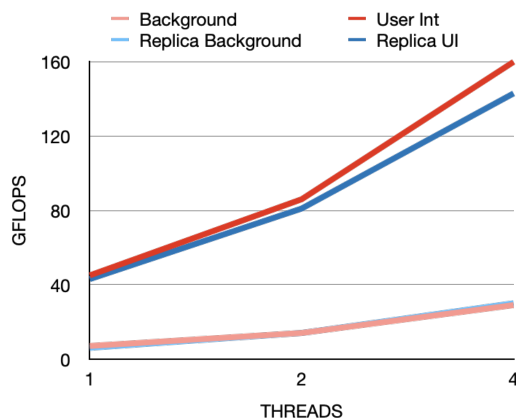

Figura 4.66: Comparativa en *dgemm*

Otro aspecto a analizar el rendimiento de las librería y la implementación directa del microkernel frente a diversos tamaños de matriz. Se puede apreciar (ver Figuras 4.67 y 4.68) que BLIS es más constante además de generar un mayor rendimiento que la ver-

sión simplificada. Este rendimiento tan constante sea debido a mejoras de la librería que mejoran la gestión de memoria para producir rendimientos más constantes que en la replica.

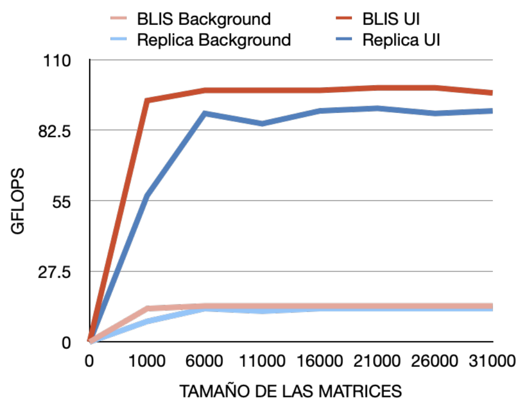

Figura 4.67: Rendimientos ante diferentes tamaños en *sgemm*

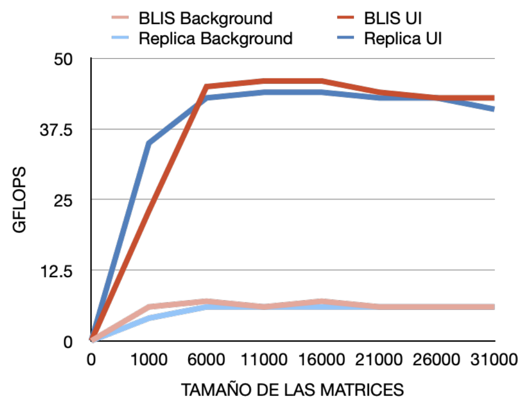

Figura 4.68: Rendimientos ante diferentes tamaños en *dgemm*

Cuando comparamos consumo, ambos códigos consumen casi lo mismo. En algunos casos consumen 100 mW más y en otros menos, pero en general las cifras son las mismas. Por tanto, la eficiencia de BLIS es mayor al consumir menos por cada GFLOP obtenido.

Cuando se utiliza la versión de BLIS para el AMX, se puede hallar dentro de esta librería un microkernel de precisión doble de 16x16 (estos son los tamaños de matrices que maneja la caché) que no se utiliza. Esto posiblemente se debe a que el investigador RuQing Xu (desarrollador de los microkernels para el AMX) fue probando distintas configuraciones porque no se conoce con exactitud lo que hay dentro del AMX, haciendo que Xu haya ido probando distintas configuraciones y haya acabado utilizando la que mejor rendimiento haya dado, que en este caso ha sido el microkernel de 32x16, que es el que se usa al utilizar la librería. Es interesante entonces ver cómo rinde este microkernel en comparación con la librería.

Cabe destacar que los parámetros que utiliza el microkernel no son los óptimos. Dentro de la carpeta de configuración de BLIS encontramos distintos valores de configuración para los distintos microkernels. Sin embargo, aunque algunos se pueden adaptar (como MR y NR, que se establecen ambos a 16), el resto de valores de configuración (como MC, KC y NC) solo vienen indicados para el microkernel de 32x16. Por tanto, el microkernel funciona correctamente pero no óptimamente.

Atendiendo al rendimiento (ver Figura 4.69) se puede ver que obtiene 70 GFLOPS de media en calidad de servicio Background y llega a alcanzar los 228 GFLOPS de media en calidades de servicio superiores. Cuando estos rendimientos se comparan (ver Figura 4.70) con el rendimiento de la librería, obtiene resultados similares en calidad Background pero dista de BLIS cuando la ejecución se lleva a cabo en los cores Firestorm. Es preciso destacar que estas comparativas son entre BLIS AMX y el código simplificado que hace uso del microkernel de 16x16, mientras que las comparativas previas son de BLIS ARMv8 con el código simplificado que hace uso del microkernel para ARMv8.

Si se analiza el rendimiento de las dos pruebas en single thread para calidades de servi-

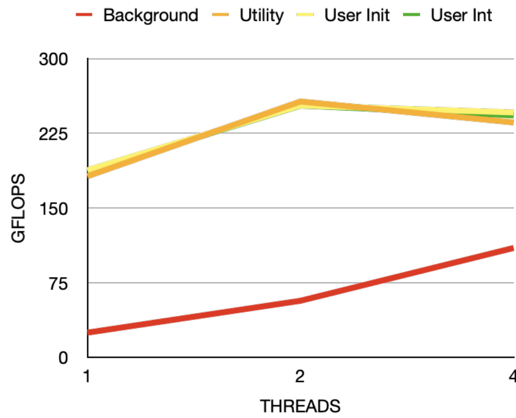

Figura 4.69: Rendimiento 16x16

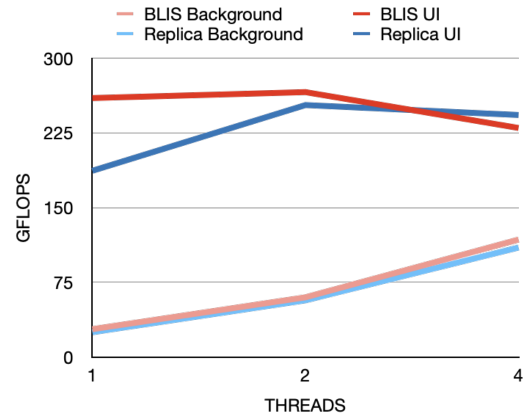

Figura 4.70: Comparativa ante BLIS AMX

cio Background y User Interactive (ver Figura 4.71), se puede apreciar que el rendimiento de la replica es bastante irregular frente a BLIS. Esto también es probable que sea debido a que BLIS gestiona mejor la memoria para poder mantener un rendimiento relativamente constante.

Otra de las cosas que llama la atención es el consumo (ver Figura 4.72), puesto que utilizar el microkernel de 16x16 es mucho más ineficiente en calidades de servicio altas. Sin embargo, en calidad de servicio Background, utilizar el microkernel directamente es más eficiente y obtiene resultados ligeramente inferiores al AMX, por lo que podría ser una opción viable si lo que se quiere tener en cuenta es el consumo de energía.

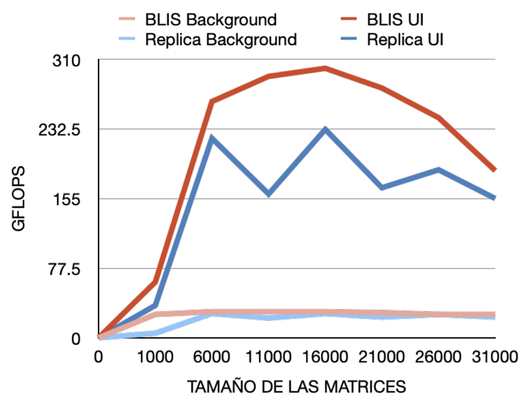

Figura 4.71: Rendimiento por tamaños

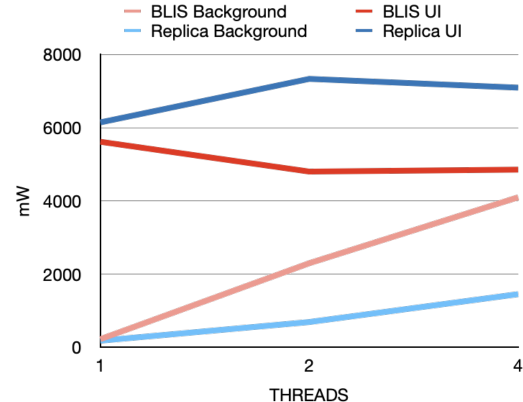

Figura 4.72: Consumo frente BLIS AMX

4.4. Utilización de Icestorm y Firestorm Conjuntamente

Una de las pruebas más interesantes del proyecto era ver si se conseguían utilizar los cores Firestorm y Icestorm conjuntamente para acelerar las operaciones y obtener así un ligero aumento de rendimiento. Tras lograrlo, los resultados han sido los esperados. No solo se obtiene más rendimiento, sino que se obtiene una eficiencia todavía mayor. Es preciso destacar que se cree que Accelerate no obtiene el máximo rendimiento posible haciendo uso del AMX. Esto queda patente cuando se compara con BLIS AMX, que obtiene algo más

de rendimiento. Por tanto, la finalidad de la presente sub-sección es lograr que Accelerate aumente su rendimiento algo más para acercarse a las cifras de BLIS AMX.

Cuando las pruebas se ejecutan en *sgemm* (ver Figura 4.73), Accelerate obtiene una media de 900 GFLOPS utilizando solo los cores Firestorm. Sin embargo, la pequeña mejora que hace uso también de Icestorm obtiene una media de 950 GFLOPS, que suponen una mejora de rendimiento del 5 %. Al ejecutar las pruebas en precisión doble (ver Figura 4.74), Accelerate obtiene 237 GFLOPS de media, mientras que la versión que distribuye la carga entre Firestorm y Icestorm obtiene una media de 255 GFLOPS. Esto supone una mejora de rendimiento del 8 % respecto a Accelerate.

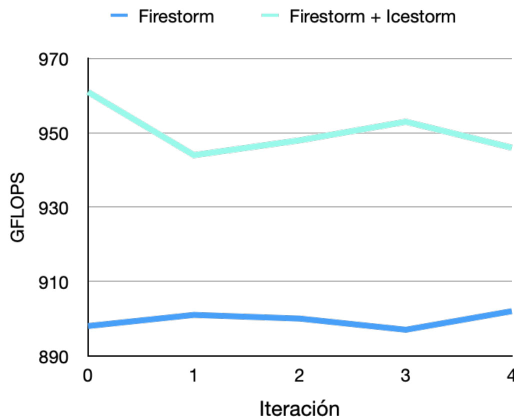

Figura 4.73: Rendimiento en *sgemm*

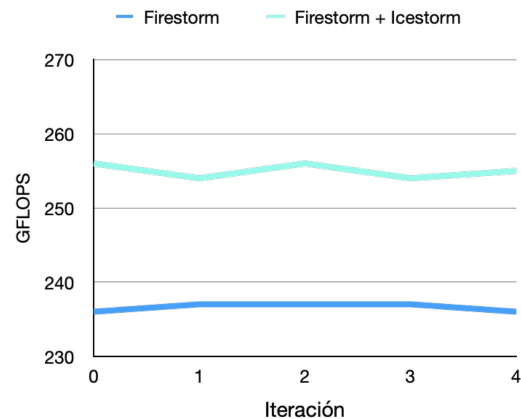

Figura 4.74: Rendimiento en *dgemm*

Cuando se atiende al consumo (ver Tablas 4.1 y 4.2) destaca que Accelerate usa mínimamente los cores Icestorm. Sin embargo, este poco uso que hace de Icestorm puede no ser beneficioso, ya que casi todo el trabajo en Firestorm. Además, del consumo indicado (38 mW en *sgemm* y 71 mW en *dgemm*) hay que tener en cuenta que entre 3-10 mW son para tareas propias del sistema. Por tanto, se puede apreciar que delegando un 10 % del trabajo en los cores Icestorm, que son altamente eficientes, el consumo total es similar o incluso inferior a Accelerate. En *sgemm* , el consumo de la versión que usa ambos tipos de core consume un 8 % menos mientras obtiene, de media, un 7 % más de rendimiento. En precisión doble, el consumo es un 3 % mayor, pero obtiene de media un 8 % más de GFLOPS.

Tabla 4.1: Consumo en *sgemm*

	Consumo Icestorm	Consumo Firestorm	Consumo Total
Accelerate	38 mW	4731 mW	4769 mW
Firestorm + Icestorm	216 mW	4294 mW	4511 mW

Tabla 4.2: Consumo en *dgemm*

	Consumo Icestorm	Consumo Firestorm	Consumo Total
Accelerate	71 mW	3763 mW	3834 mW
Firestorm + Icestorm	234 mW	3190 mW	3943 mW

Por tanto, Accelerate genera 0,19 GFLOPS/mW en *sgemm* frente a los 0,21 GFLOPS/mW de la versión que utiliza ambos cores. Cuando se trata de *dgemm* , Accelerate obtiene un consumo de 0,062 GFLOPS/mw frente a los 0,064 GFLOPS/mW de la versión que distribuye el trabajo entre los distintos tipos de cores. No es una mejora muy grande pero en aplicaciones donde el consumo es importante, es una mejora significativa.

4.5. GPU

Las últimas pruebas del proyecto consisten en medir el rendimiento de *gemm* en FP32 en Accelerate y compararlo con el rendimiento que se obtiene haciendo la misma operación en la GPU, que es algo más compleja de utilizar. Una de las primeras conclusiones que se obtienen tras estas pruebas es que posiblemente la GPU no admita tamaños de datos tan grandes en un solo comando. Esto se obtiene tras analizar el rendimiento frente a los diversos tamaños de matrices: cuando sobrepasan los 26.000, la GPU aparentemente termina en menos de 6 segundos, obteniendo un supuesto rendimiento de 11 TFLOPS. Sin embargo, esto no es posible porque el límite teórico de rendimiento de la GPU en FP32 es de 2,6 TFLOPS. Por tanto, es posible que haya que subdividir las matrices en comandos de menor tamaño.

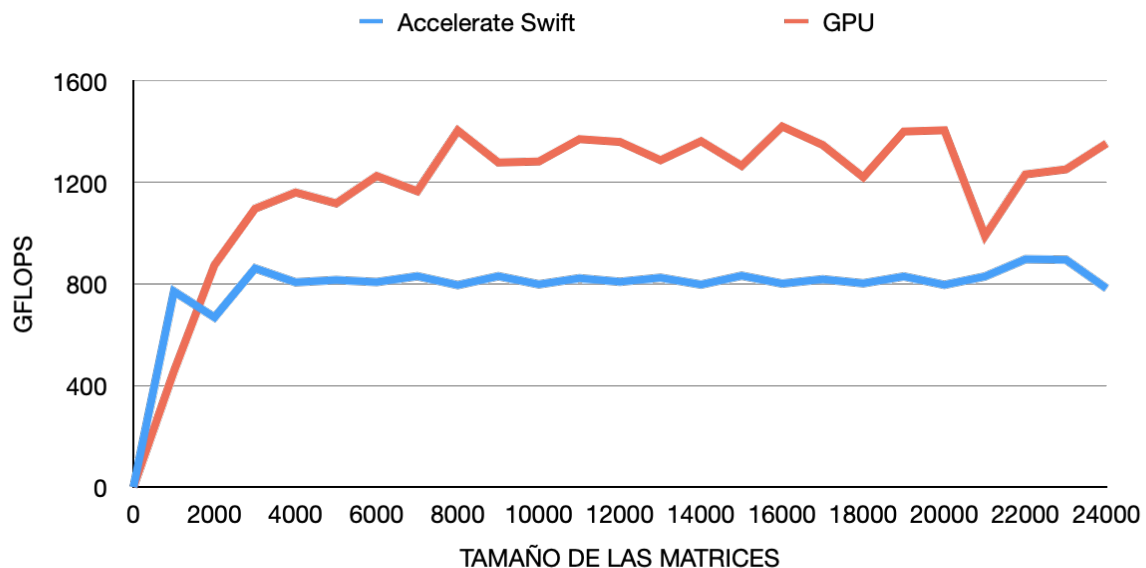

Figura 4.75: Comparativa de rendimiento entre Accelerate y la GPU en *sgemm*

Atendiendo a los resultados de rendimiento, se puede apreciar que para tamaños inferiores a 2000, Accelerate obtiene mejores resultados. Sin embargo, cuando el tamaño de las matrices sobrepasa ese número, el rendimiento de Accelerate decae mientras que el de la CPU se tiende a mantener constantemente por encima de los 1200 GFLOPS. Esto es normal puesto que las GPU están diseñadas para tratar volúmenes grandes de datos y obtener un rendimiento muy alto, mientras que Accelerate es una librería posiblemente diseñada para el tratamiento de volúmenes de datos más pequeños pero de tratamiento más rápido.

La GPU del M1 obtiene, en las pruebas realizadas, una media de 1223 GFLOPS, mien-

tras que Accelerate obtiene 814 GFLOPS de media. Estos resultados son normales puesto que la GPU está diseñada para realizar estas operaciones de datos tan grandes y además tiene un consumo de energía más elevado. Accelerate promedia un consumo de 4590 mW frente a los 7250 mW de la GPU. Estas cifras son similares a las que se han obtenido en otras pruebas de rendimiento por parte de investigadores.

Las pruebas no se han realizado en multithread porque se desaconseja su uso en la GPU según la propia documentación de Apple, ya que puede darse el caso de que se utilicen múltiples kernels que escriban en el mismo buffer de comandos. Apple considera el uso del multithreading con kernels algo no seguro, por lo que las pruebas se han realizado en single thread.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones del Trabajo

El presente proyecto se ha centrado en realizar un análisis de las distintas librerías BLAS sobre el chip M1 de Apple con la finalidad de determinar cual es más útil para el cálculo matricial según el dispositivo usado y las prioridades que se tengan (rendimiento, consumo, etc). Para ello, se han llevado a cabo las operaciones *sgemm* y *dgemm* en varias librerías y además se ha replicado, en el caso de BLIS, el funcionamiento de la librería con sus microkernels. Además, se ha logrado utilizar la GPU, que en algunos casos puede ser la mejor opción.

Tras finalizar las pruebas, se concluye que BLIS AMX es la librería que más rendimiento obtiene en relación al consumo de energía que realiza, que es menor que otras librerías que no usan el co-procesador AMX y que la librería Accelerate de Apple. Esta librería es posiblemente la mejor opción cuando se quiere obtener un gran rendimiento a la par que bajo consumo.

Respecto a Accelerate, tras este proyecto ha quedado patente que Apple ha diseñado Accelerate prestando mucha atención al consumo energético, puesto que al ser una librería que está en iOS y watchOS (es decir, sistemas móviles que han de consumir lo mínimo posible y dar buen rendimiento), la prioridad ha sido el consumo de energía más bajo que han podido conseguir. Esto no implica que hayan sacrificado en rendimiento, ya que las cifras de rendimiento obtenidas en las pruebas son muy buenas.

Respecto al resto de librerías puestas a prueba, sus cifras de consumo y rendimiento son peores puesto que no pueden hacer uso del AMX. Sin embargo, mientras que Accelerate y BLIS (la versión para el AMX) tienen un límite de rendimiento debido a que hacen uso del AMX, OpenBLAS y BLIS ARMv8 no cuentan con esta limitación, por lo que si el consumo no es una preocupación, con más threads se supera el rendimiento de Accelerate y BLIS AMX.

Se han analizado ciertos comportamientos no esperados por parte del planificador al usar cores de alto rendimiento a baja frecuencia (que consumen mucho) en lugar de realizar el trabajo en los cores más eficientes. Se desconoce la razón de esto ya que no se tiene acceso al planificador de macOS, por lo que las razones por las que hace esto son desconocidas.

Otra de las conclusiones obtenidas tras la finalización del proyecto es que se puede mejorar levemente el rendimiento de Accelerate haciendo de las calidades de servicio. Estas han sido exploradas y utilizándolas adecuadamente se puede lograr utilizar los dos tipos de cores simultáneamente. Esto permite aumentar el rendimiento de Accelerate un 7 % mientras el consumo apenas aumenta un 2 %. En algunos casos, esta mejora puede ser muy beneficiosa.

Una opción que da más rendimiento (a costa de un consumo medio de 7 W) es el uso de la GPU. Esta produce en torno a un 40 % más de rendimiento que Accelerate, pero su uso es algo tedioso y no hay mucha información al respecto. Además, se ha observado un comportamiento extraño en el uso de la GPU: cuando el tamaño de las matrices es muy grande ($n = m = k > 29{,}000$) la GPU no funciona bien enviando un solo comando con todos los datos. Los resultados obtenidos cuando se han mandado matrices superiores a ese tamaño han sido datos no realistas (obtener 11 TFLOPS de cálculo cuando el rendimiento máximo teórico es de 2,6 TFLOPS) o simplemente ha dejado de trabajar con los datos, y aunque el proceso indica que está en marcha y que la GPU está trabajando, el consumo energético de la GPU es nulo, indicando que ha fallado al realizar las operaciones y esta se ha apagado.

5.2. Extensiones

Relacionado directamente con el proyecto está la posible ampliación de crear un código que utilice el microkernel para AMX en un thread y el microkernel para ARMv8 en otro thread. Esto permitiría ver el rendimiento cuando Icestorm hace uso del AMX y Firestorm, del microkernel para ARMv8 y viceversa. Esto supone una ampliación directa del Capítulo 4 Sección 4.

Además, ante los problemas mencionados con la GPU anteriormente, una futura continuación del proyecto podría involucrar la división del trabajo adaptada a la GPU para poder resolver dichos problemas. Además, Swift es muy lento creando arrays aleatorios para matrices y la integración de código en C en proyectos en Swift es tediosa. Otra posible investigación sería la integración de funciones en C que creasen arrays aleatorios con un tamaño especificado previamente, ya que crearlos en Swift consume mucho tiempo (más que las operaciones de multiplicación en si).

Junto a estas propuestas de futuros proyectos se podrían añadir comparaciones frente a las nuevas versiones del chip M1 creadas por Apple. En Octubre de 2021, la compañía presentó los nuevos chips M1 Pro y M1 Max (Apple (2021b)), incorporados en los MacBook Pro de 14 y 16 pulgadas, y unos meses más tarde, en Marzo de 2022, volvieron a presentar una nueva versión llamada M1 Ultra (Apple (2021c)), incorporada en el nuevo Mac Studio. Estos chips fueron presentados como versiones mejoradas del chip M1 que van enfocadas a distintos tipos de consumidores. Los tres nuevos chips presentan, según los números iniciales, resultados muy buenos y dejan atrás al M1 en todo tipo de prueba de rendimiento. Apple va a dedicar el chip M1 a los equipos de baja potencia (MacBook Air, iMac, Mac Mini y MacBook Pro de 13"), el M1 Pro y M1 Max a los portátiles más potentes (MacBook Pro de 14z 16") y el M1 Ultra para el equipo más potente que tiene a la venta: el Mac Studio. Sería interesante repetir las mismas pruebas en dichos equipos y comparar con el M1.

Se menciona también al principio del proyecto que el M1 es muy similar al chip A14 del iPhone 12. Podría ser interesante obtener acceso root a un iPhone 12 (hay herramientas como unc0ver (Team (2021b))/Taurine (Team (2021a)) que aprovechan vulnerabilidades de software para obtener acceso root) y ejecutar en él las mismas pruebas. Al fin y al cabo, en un iPhone con acceso root se puede instalar acceso por ssh, homebrew y multitud de herramientas que se pueden instalar en macOS. Por tanto, sería una buena idea ver como rinden los Icestorm/Firestorm en el chip móvil frente a los del M1 y sacar conclusiones.

Chapter 6

Conclusions and Future Work

6.1. Conclusions

This project has focused on the analysis of the different BLAS libraries on the Apple M1 chip in order to determine which one is more useful for matrix computation depending on the device used and the priorities (performance, consumption, etc.). For this purpose, the operations *sgemm* and *dgemm* have been carried out in several libraries and, in addition, the operation of the library with its microkernels has been replicated in the case of BLIS. In addition, it has been possible to use the GPU, which in some cases may be the best option.

After finishing the tests, it is concluded that BLIS AMX is the library that obtains the best performance in relation to its power consumption, which is lower than other libraries that do not use the AMX co-processor and Apple's Accelerate library. This library is possibly the best choice when you want high performance and low power consumption.

Regarding Accelerate, after this project it has become clear that Apple has designed Accelerate paying close attention to power consumption, since being a library that is in iOS and watchOS (ie mobile systems that have to consume as little as possible and give good performance), the priority has been the lowest power consumption they could get. This does not imply that they have sacrificed on performance, as the performance figures obtained in the tests are very good.

With respect to the other libraries tested, their power consumption and performance figures are worse since they cannot make use of AMX. However, while Accelerate and BLIS (the version for the AMX) have a performance limit because they make use of the AMX, OpenBLAS and BLIS ARMv8 do not have this limitation, so if consumption is not a concern, with more threads the performance of Accelerate and BLIS AMX is exceeded.

Some unexpected behavior on the part of the scheduler has been analyzed when using high performance cores at low frequency (which consume a lot) instead of doing the work on the more efficient cores. The reason for this is unknown as there is no access to the macOS scheduler, so the reasons why it does this are unknown.

Another conclusion drawn after completion of the project is that Accelerate's perfor-

mance can be slightly improved by making qualities of service. These have been explored and by using them properly it is possible to use both types of cores simultaneously. This allows Accelerate performance to be increased by 7 percent while consumption is only increased by 2 percent. In some cases, this improvement can be very beneficial.

An option that gives more performance (at the cost of an average power consumption of 7W) is the use of the GPU. This unit produces about 40% more performance than Accelerate, but it is somewhat tedious to use and there is not much information about it. In addition, a strange behavior has been observed while using the GPU: when the size of the matrixes is very large (around $n = m = m = 29000$) the GPU does not work well sending a single command with all the data. The results obtained when sending matrices larger than that size have been unrealistic data (getting 11 TFLOPS of computation when the theoretical maximum performance is 2.6 TFLOPS) or it has simply stopped working with the data, and although the process indicates that it is running and the GPU is working, the GPU power consumption is zero, indicating that it has failed to perform the operations and the GPU has shut down.

6.2. Future Work

Directly related to the project is the possible extension of creating code that uses the microkernel for AMX in one thread and the microkernel for ARMv8 in another thread. This would allow to see the performance when Icestorm makes use of the AMX and Firestorm, the microkernel for ARMv8 and vice versa. This is a direct extension of Chapter 4 Section 4.

Also, given the aforementioned GPU issues, a future continuation of the project may involve GPU-enabled division of labor in order to address these issues. Also, Swift is very slow creating random arrays for arrays and integrating C code into Swift projects is tedious. Another possible investigation would be the integration of functions in C that create random arrays with a previously specified size, since creating them in Swift takes a lot of time (more than the multiplication operations themselves).

Along with these proposals for future projects, comparisons could be added against the new versions of the M1 chip created by Apple. In October 2021, the company introduced the new M1 Pro and M1 Max chips (Apple (2021b)), built into the 14-inch and 16-inch MacBook Pro, and a few months later, in March 2022, they re-introduced a new version called M1 Ultra (Apple (2021c)), built-in in the new Mac Studio. These chips were presented as improved versions of the M1 chip that are aimed at different types of consumers. The three new chips show, according to the initial numbers, very good results and leave the M1 behind in all kinds of performance tests. Apple is dedicating the M1 chip to low-power computers (MacBook Air, iMac, Mac Mini and MacBook Pro 13"), the M1 Pro and M1 Max to more powerful laptops (MacBook Pro 14" and 16") and the M1 Ultra for the most powerful computer it has for sale: the Mac Studio It would be interesting to repeat the same tests on these computers and compare with the M1.

It is also mentioned at the beginning of the project that the M1 is very similar to the A14 chip in the iPhone 12. It might be interesting to get root access to an iPhone 12 (there are tools like unc0ver (Team (2021b))/Taurine (Team (2021a)) that take advantage of software vulnerabilities to gain root access) and run on he the same tests. After all, on an iPhone with root access you can install ssh access, homebrew and a multitude of tools

that can be installed on macOS. Therefore, it would be a good idea to see how the Icestorm/Firestorm in the mobile chip perform against those in the M1 and draw conclusions.

Bibliografía

ANANDTECH. Apple Announces The Apple Silicon M1: Ditching x86 - What to Expect, Based on A14. https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2, 2020a. [Online; accessed 30-May-2022].

ANANDTECH. The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 to the Test. https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested/3, 2020b. [Online; accessed 30-May-2022].

APPLE. Presentacion Apple M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/, 2020. [Online; accessed 30-May-2022].

APPLE. Cuestion de Foro de Desarrolladores. https://developer.apple.com/forums/thread/666635, 2021a. [Online; accessed 30-May-2022].

APPLE. Nota de Prensa del M1 Pro y M1 Max. https://www.apple.com/es/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built/, 2021b. [Online; accessed 30-May-2022].

APPLE. Nota de Prensa del M1 Ultra. https://www.apple.com/es/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/, 2021c. [Online; accessed 30-May-2022].

APPLE. Secure Enclave. https://support.apple.com/en-ca/guide/security/sec59b0b31ff/web, 2021d. [Online; accessed 30-May-2022].

APPLE. Accelerate. https://developer.apple.com/documentation/accelerate, 2022a. [Online; accessed 30-May-2022].

APPLE. Accelerate Sample Code. https://developer.apple.com/accelerate/sample-code/, 2022b. [Online; accessed 30-May-2022].

APPLE. Documentacion de Dispatch. https://developer.apple.com/documentation/DISPATCH, 2022c. [Online; accessed 30-May-2022].

APPLE. Documentacion de vDSP. https://developer.apple.com/documentation/accelerate/vdsp, 2022d. [Online; accessed 30-May-2022].

APPLE. Documentación de Metal. https://developer.apple.com/metal/, 2022e. [Online; accessed 30-May-2022].

Apple. Documentación de Metal Performance Shaders. https://developer.apple.com/documentation/metalperformanceshaders, 2022f. [Online; accessed 30-May-2022].

Apple. Prioritizr Work With Quality of Service Classes. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/PrioritizeWorkWithQoS.html, 2022g. [Online; accessed 30-May-2022].

Company, T. E. L. Explainer: chipsets and Fabric. https://eclecticlight.co/2021/11/06/explainer-chipsets-and-fabric/, 2021. [Online; accessed 30-May-2022].

Group, O. OpenBLAS. https://en.wikipedia.org/wiki/OpenBLAS, 2022a. [Online; accessed 30-May-2022].

Group, O. OpenBLAS: An Optimized BLAS Library. https://www.openblas.net, 2022b. [Online; accessed 30-May-2022].

Group, O. OpenBLAS Documentation. https://github.com/xianyi/OpenBLAS, 2022c. [Online; accessed 30-May-2022].

Science of High-Performance Computing (SHPC) group, U.-A. BLIS (Software). https://en.wikipedia.org/wiki/BLIS_(software), 2021. [Online; accessed 30-May-2022].

Science of High-Performance Computing (SHPC) group, U.-A. BLIS Documentation. https://github.com/flame/blis, 2022. [Online; accessed 30-May-2022].

Jianyu Huang, R. A. v. d. G. Accelerate Sample Code. https://www.researchgate.net/publication/307564216_BLISlab_A_Sandbox_for_Optimizing_GEMM, 2022. [Online; accessed 30-May-2022].

Johnson, D. Firestorm Overview. https://dougallj.github.io/applecpu/firestorm.html, 2020a. [Online; accessed 30-May-2022].

Johnson, D. Icestorm Overview. https://dougallj.github.io/applecpu/icestorm.html, 2020b. [Online; accessed 30-May-2022].

Johnson, D. Analisis del AMX. https://gist.github.com/dougallj/7a75a3be1ec69ca550e7c36dc75e0d6f, 2022. [Online; accessed 30-May-2022].

Medium. The Secret Apple M1 Coprocessor. https://medium.com/swlh/apples-m1-secret-coprocessor-6599492fc1e1, 2021. [Online; accessed 30-May-2022].

Notebookcheck. Apple M1 8-core GPU. https://www.notebookcheck.net/Apple-M1-GPU-GPU-Benchmarks-and-Specs.503610.0.html, 2020. [Online; accessed 30-May-2022].

Notebookcheck. Apple's use of fabric in the M1 Pro and M1 Max chips points to how it will scale up its chips for the next Mac Pro. https://www.notebookcheck.net/Apple-s-use-of-fabric-in-the-M1-Pro-and-M1-Max-chips-points-to-how-it-will-scale-up-its 574212.0.html, 2021. [Online; accessed 30-May-2022].

Org, A. L. Reverse Engineering DCP. https://asahilinux.org/2021/08/progress-report-august-2021/, 2021. [Online; accessed 30-May-2022].

SamaGames. The power, consumption and efficiency of the Apple M1 processor, to the test: a before and after in numbers and in real use. https://samagame.com/blog/news/the-power-consumption-and-efficiency-of-the-apple-m1-processor-to-the-test-a-before-and-
2020. [Online; accessed 30-May-2022].

Team, O. Taurine. https://taurine.app, 2021a. [Online; accessed 30-May-2022].

Team, U. Unc0ver. https://unc0ver.dev, 2021b. [Online; accessed 30-May-2022].

Wikipedia. Basic Linear Algebra Subprograms. https://es.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms, 2022. [Online; accessed 30-May-2022].

Xu, R. BLIS for Apple Matrix Coprocessor. https://github.com/xrq-phys/blis_apple, 2021a. [Online; accessed 30-May-2022].

Xu, R. RuQing Xu - BLIS and TBLIS on SVE and Apple AMX - AHUG SC21. https://www.youtube.com/watch?v=xMiWe07Rjss, 2021b. [Online; accessed 30-May-2022].