Universidad de Córdoba

Arquitectura y Tecnología de Computadores

Práctica 3

MIPS y el panorama actual

Arquitecturas Avanzadas de Procesadores

D. Miguel Ángel Montijano Vizcaíno (el1movim@uco.es)
D. Héctor Martínez Pérez (el2mapeh@uco.es)

Curso 2025/2026

1 Introducción

En la primera práctica se vio el funcionamiento principal del entorno MARS; en la segunda, se vieron las principales instrucciones del ISA de MIPS. Finalmente, esta última práctica servirá para unificar los conocimientos adquiridos, además de ponerlos en valor abordando problemáticas de la época actual.

Los objetivos principales de esta última sesión de prácticas son:

- Comprender la panorámica actual referente a la arquitectura de computadores.
- Ser capaces de aplicar los conocimientos adquiridos a problemas reales en el mundo de la informática actual.

2 Hardware

Año 2000, uno de los principales objetivos en el campo de la informática empezaba a cambiar de rumbo. Pese a que siempre se ha trabajado en la optimización y el aceleramiento del procesamiento de los datos, se vivía una era donde uno de los principales objetivos era la recolección de estos. Con el cambio de milenio, vino también un cambio de era; alrededor del año 2010, el principal objetivo dejaba de ser la recolección de datos, y la prioridad pasaba a ser su procesamiento. Esto se conoce como: ***era del big data y la Inteligencia Artificial***.

El campo de la arquitectura de computadores siempre ha tenido un papel fundamental en el mundo de la informática. No obstante, en este cambio de era, los esfuerzos por crear nuevos procesadores capacitados para procesar la mayor cantidad de datos en el menor tiempo posible han sido mucho mayores.

Vivimos una época donde la Inteligencia Artificial domina; cualquier tipo de dispositivo debe ser capaz de llevar a cabo consultas a redes neuronales en el menor tiempo posible, los procesadores de grandes servidores deben acelerar el entrenamiento de estas redes neuronales al máximo, etc. Y todo esto, pendiente del consumo energético, entre otros muchos factores.

Esto ha conllevado la creación de multitud de diseños de arquitecturas de computadores, nuevas tecnologías para el procesamiento de datos, etc. Para abordar todos estos retos a los que nos enfrentamos. Un ejemplo destacable de estas nuevas tecnologías son:

- **Apple M4**: Capacitado con una de las mejores tecnologías actuales para la aceleración del producto matricial, denominada AMX (Advanced Matrix Extension).
- **Intel Sapphire Rapids**: Primera arquitectura en integrar la tecnología AMX. Soporta además AVX-512 y BF16.
  BF16: Ocupa 16 bits igual que *FP16*, pero dedica un mayor número de bits al exponente que FP16, ocho, y un menor número de bits a la mantisa, siete.
- **Jetson Orin (Cortex A78AE)**: Dispositivo empotrado, soporta las actuales instrucciones basadas en SDOT incorporadas en la unidad SIMD y soportadas por el ISA de ARM8.2-A.

- **RISC-V**: La nueva gama de procesadores que están empezando a emerger, físicamente, en la actualidad. Existen diferentes modelos como son: *C910* y *C906*, que soportan el ISA de rvv0.7, y otros como *C908* y *SpacemiT K1*, que soportan el ISA rvv1.0. Además, este último modelo de procesador, incorpora una unidad aritmética especial para el aceleramiento matricial.
  En SpacemiT K1, para el desarrollo de códigos vectoriales que usen FP16 como la unidad especial IME, aún no existen instrucciones de alto nivel que las soporten, por lo tanto, es imprescindible el uso de *asm*.

3 Software

En este punto, falta el nexo de unión entre todos los elementos que hemos visto hasta el momento. Como se ha comentado, vivimos en la era del big data y la inteligencia artificial. Como dato relevante, las acciones relacionadas con: identificación de personas, objetos, animales, etc. en imágenes (tarea cuotidiana en dispositivos móviles, de seguridad, etc.) se llevan a cabo mediante inferencia con redes neuronales convolucionales.

Por otro lado, el 30 de noviembre de 2022, OpenAI liberó su primer generador de lenguaje denominado ChatGPT (nombre derivado del modelo de transformer decoder usado, GPT-3.5). Pese a que eran muchos los años en los que se llevaba trabajando en este tipo de modelos de inteligencia artificial (transformers encoders y transformers decoders), hasta esta fecha no llegaron a tener este impacto tan grande.

Dicho esto, la Figura 1 muestra un desglose del porcentaje del tiempo (de manera general) de las principales operaciones que se llevan a cabo durante la inferencia de distintos modelos de inteligencia artificial (teniendo en cuenta que ninguna de las operaciones está optimizada). Cabe destacar, que el cálculo de la convolución, mediante la transformación *im2col/im2row* se puede convertir en una GEMM.

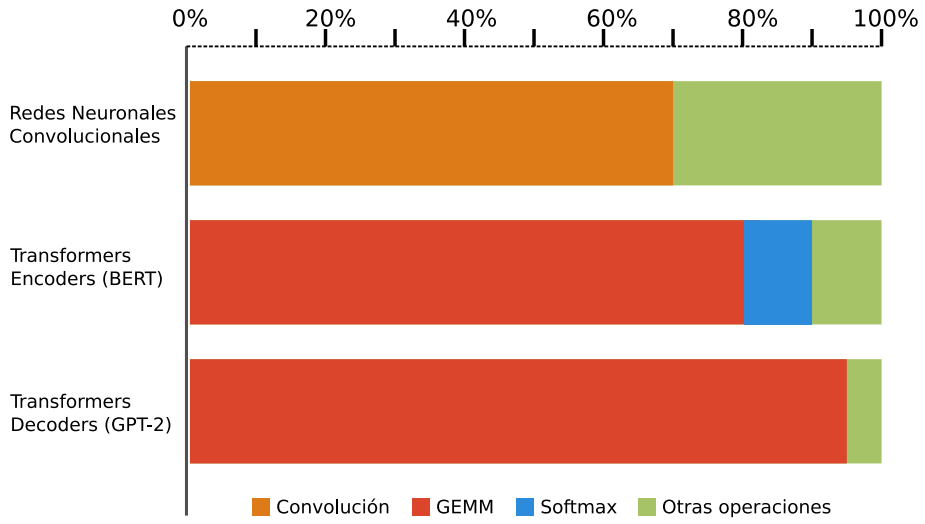

Figura 1: Desglose del tiempo de procesamiento de diferentes modelos de inteligencia artificial.

Dicho esto, se acaba de encontrar uno de los nexos entre las nuevas arquitecturas de procesadores emergentes, y uno de los mayores retos que continúa abordando la comunidad informática en la actualidad: La optimización del producto Matriz-Matriz (GEMM).

Finalmente, la última cuestión que falta resolver es: ¿Cual es el nexo de unión con las prácticas anteriores?

4 Vínculo entre Hardware y Software

A la hora de desarrollar a alto nivel con lenguajes de programación como C/C++, como ya sabemos, un paso obligatorio es la transformación del código desarrollado a código ensamblador mediante un compilador. Pese a que C/C++, dispone de multitud de compiladores altamente optimizados a la hora de generar ensamblador (gcc, g++, icc, etc.), cuando la optimización de un algoritmo es crucial, y debe de amoldarse perfectamente a las características de la arquitectura del procesador en el cual se va a ejecutar, la opción extremadamente recomendable es el uso directo del lenguaje ensamblador (*asm*).

Cuando desarrollamos un algoritmo en *asm* podemos controlar:

- Registros del procesador usados.
- Control de riesgos de datos.
- Control total de las instrucciones usadas.
- Mayor control en técnicas de *prefetching* sobre los diferentes niveles de caché y *software pipelining*.

Empezamos a observar la relación tan estrecha que existe entre la arquitectura del procesador y el desarrollo optimizado de algoritmos para una determinada arquitectura.

En la actualidad, existen multitud de bibliotecas desarrolladas que tienen en cuenta todos estos aspectos y se usan diariamente en servidores de Google, Facebook, OpenAI, telefonía móvil, dispositivos empotrados, etc. Un ejemplo de ellas son:

- **XNNPACK**: Desarrollada por Google y Facebook, usada en Pytorch, Android, etc. En el enlace adjunto puede observarse la implementación de un micro-kernel para GEMM en FP32 para procesadores A53 de ARM: `https://github.com/google/XNNPACK/blob/master/src/f32-gemm/1x12-aarch64-neonfma-cortex-a53.S.in`
- **BLIS**: Biblioteca de álgebra lineal. En ella participan grandes empresas/organizaciones como: AMD, Intel, Texas Instruments, Huawei, ARM, Texas University, etc. El enlace adjunto muestra un micro-kernel genérico para GEMM en FP32 enfocado a arquitecturas RISC-V: `https://github.com/flame/blis/blob/master/kernels/rviv/3/bli_czgemm_rviv_asm_4vx4.h`
- **MKL**: Biblioteca desarrollada por Intel, y específica para procesadores con arquitectura Intel. Su desarrollo es de código cerrado.

Como puede observarse, el desarrollo de las partes críticas del código, se llevan a cabo en *asm*, si se mira más profundamente los códigos desarrollado, pese a su complejidad, después de las dos sesiones de prácticas anteriores, podemos ser capaces de entender ciertas zonas de código. Esto se debe a que, pese a que el ISA para cada arquitectura de procesador varíe, y tenga sus propias especificaciones, suelen conservar un estándar, véase la Tabla 1. Nótese la similitud entre las diferentes instrucciones, en especial, entre MIPS y RVV1.0.

Ámbito	**MIPS**	**ARMv8.2**	**INTEL**	**RVV1.0**
Carga 4 bytes	lw	ldr	mov	lw
Almacenamiento 4 bytes	sw	str	mov	sw
Suma	add	add	add	add
Resta	sub	sub	sub	sub
Multiplicación	mul	mul	imul	mul
Salto incondicional	j	b	jmp	j
Salto condicional	beq	b.eq	je	beq

Tabla 1: Diferencias y similitudes entre el ISA de distintas arquitecturas.

Dicho esto, queda reflejada la importancia del conocimiento del lenguaje *asm* para el desarrollo de algoritmos altamente optimizados, así como la similitud de las diferentes instrucciones del ISA de cada tipo de arquitectura.

Ejercicio 1: Optimizando los accesos a memoria

Una vez vista la importancia del manejo del lenguaje de bajo nivel *asm* y de la optimización del producto matriz-matriz (GEMM), abordemos la primera problemática: **los accesos a memoria**.

Antes de pasar a la descripción del ejercicio, es imprescindible la comprensión de los distintos tipos de almacenamiento de matrices en memoria. Estamos acostumbrados a la reserva de matrices mediante múltiples punteros; la problemática de este tipo de reservas es la localidad en memoria. Para facilitar los accesos a memoria, contigüidad de los datos, *prefetching*, etc. Los datos de una matriz deben almacenarse de manera contigua.

Los dos principales formatos de almacenamiento de matrices en memoria son:

- ***Row-major order***: La matriz se almacena dando prioridad a la contigüidad de las filas.
- ***Column-major order***: La matriz se almacena dando prioridad a la contigüidad de las columnas.

La Figura 2 muestra el esquema de los dos tipos principales de almacenamiento citados.

Dicho esto, implemente un código MIPS que, dada una matriz almacenada en memoria en formato *row-major order*, la muestre por pantalla con llamadas *syscall*. La matriz seguirá el mismo patrón que el que aparece en la parte superior de la Firgura 2 es decir,

en MIPS:

matrix: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Por pantalla se visualizará la matriz 4x4, tal y como aparece en esta misma parte superior de la Figura 2.

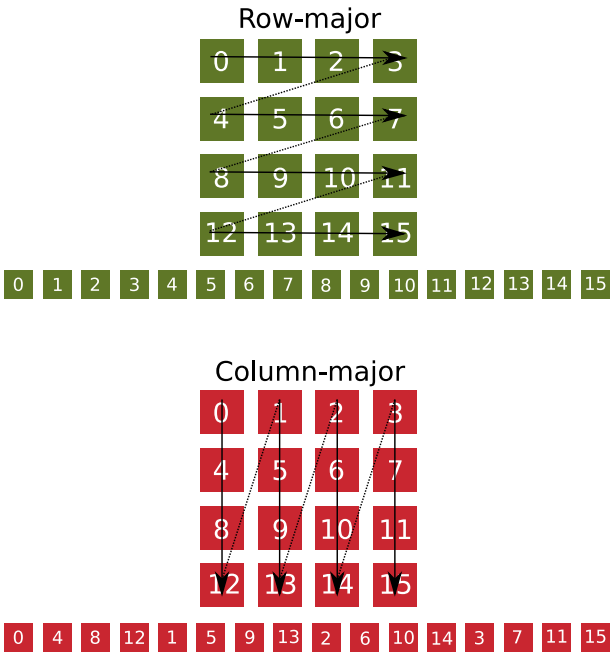

Figura 2: Esquemas de almacenamiento de matrices en memoria. *Row-major order* en la parte superior de la imagen y *column-major order* parte inferior de la imagen.

Ejercicio 2: GEMM

En este ejercicio, vamos a implementar una GEMM en código ensamblador para MIPS. Para ello, la Figura 3 muestra el esquema principal del producto matriz-matriz. Cabe destacar que existen 3 dimensiones distintas, **M, N y K**. Para simplificar el ejercicio, al igual que el anterior, se usará M=N=K=4.

Para llevar a cabo la implementación, vamos a tener en cuenta los siguientes formatos de almacenamiento:

- **Matriz A:** Almacenada en formato *col-major*.
- **Matriz B:** Almacenada en formato *col-major*.
- **Matriz C:** Almacenada en formato *row-major*.

Partiendo del Código 1, resuelva los ***TODO*** para completar el producto matricial. Para comprobar que el resultado es correcto, incluya el código desarrollado en el anterior ejercicio para observar el contenido de la matriz resultante ***mat_C***.

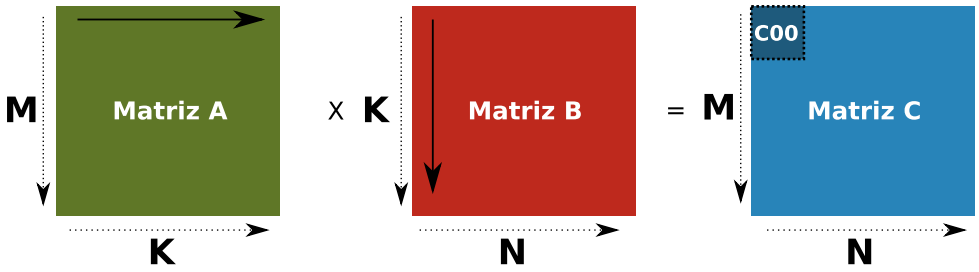

Figura 3: Esquema de una GEMM.

Código 1: Código MIPS GEMM.

```
1         .data
2         #A almacenada en column-major order
3 mat_A:  .word 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15
4         #B almacenada en column-major order
5 mat_B:  .word 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15
6         #C almacenada en row-major order
7 mat_C:  .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
8 M:      .word 4 #M=4
9 N:      .word 4 #N=4
10 K:     .word 4 #K=4
11        .text
12 main:  lw $s0, M($0)     #Carga M
13        lw $s1, N($0)     #Carga N
14        lw $s2, K($0)     #Carga K
15        li $t3, 0         #i=0
16        li $t4, 0         #j=0
17        li $t5, 0         #z=0
18 loopM:
19 loopN:
20        #TODO: Calcular y almacenar en $t6 -> i * N + j.
21        li   $t2, 0
22 loopK:
23        #TODO: Calcular y almacenar en $t7 -> A: z * M + i.
24        #TODO: Calcular y almacenar en $t8 -> B: j * K + z.
25        lw   $t0, mat_A($t7)
26        lw   $t1, mat_B($t8)
27        #TODO: Implemente el producto.
28        add  $t2, $t2, $t0
29        addi $t5, $t5, 1
30        bne  $t5, $s2, loopK
31        li   $t5, 0
32        #TODO: Almacene el resultado final de Dot-product.
33        addi $t4, $t4, 1
34        bne  $t4, $s1, loopN
35        li   $t4, 0
36        addi $t3, $t3, 1
37        bne  $t3, $s0, loopM
38 end:
39        li $v0, 10
40        syscall
```

Para facilitar la comprobación de resultados, el producto matricial debe de dar como resultado:

$$\begin{bmatrix} 56 & 62 & 68 & 74 \\ 152 & 174 & 196 & 218 \\ 248 & 286 & 324 & 362 \\ 344 & 398 & 452 & 506 \end{bmatrix}$$

Ejercicio 3: Optimización GEMM

El último ejercicio de esta práctica consistirá en optimizar la implementación del Ejercicio 2. El esquema anterior contemplaba una GEMM de tamaños dinámicos; en cambio, en casos reales, no suele ser lo habitual. Normalmente, el producto matricial se divide en una serie de productos pequeños, donde un *micro-kernel* muy optimizado lleva a cabo un producto de dimensiones fijas. Dicho esto, y sabiendo que el tamaño de las matrices no va a variar. Optimice el código anterior intentando usar el menor número posible de ciclos de instrucciones sabiendo que el tamaño de las matrices SIEMPRE será M=N=K=4.