HB. Tema 1 - Operaciones vectoriales y complejidad en tiempo y memoria

Iraida Abad, Xiomara Cáceres y Naroa Iparraguirre

16/01/2025

Índice

1 Introducción

El problema abordado consiste en comparar dos enfoques para la multiplicación de matrices. Se realizará de dos maneras distintas, con bucles anidados y operaciones vectorizadas, contrastando la eficiencia de ambos algoritmos, los tiempos y uso de memoria.

2 Código Implementado

A continuación, se presenta el código utilizado en Python:

2.1 Implementación Vectorizada

```
1 import numpy as np
2
3 def multiplicacionMatrizVectorizada(A):
4     matrix = np.dot(A,A)
5     return matrix
```

Listing 1: Multiplicación vectorizada con numpy

```
time1 = time.time()
tracemalloc.start()
A21 = multiplicacionMatrizVectorizada(A)
print("Matriz A**2 con funciones prexistentes\n", A21)
usedMemory1, _ = tracemalloc.get_traced_memory()
tracemalloc.stop()
time2 = time.time()
```

Listing 2: Calculo del tiempo y uso de memoria con la multiplicación vectorizada con numpy

2.2 Implementación en bucles

```
def multiplicacionMatrizBucles(A):
    matrix = np.zeros((n,n))
    for  i in range(n):
        for j in range(n):
            for k in range(n):
                matrix[i,j] = matrix[i,j] + A[i,k]*A[k,j]
    return matrix
```

Listing 3: Multiplicación con bucles

```
time3 = time.time()
tracemalloc.start()
A22 = multiplicacionMatrizBucles(A)
print("Matriz A**2 hard-coded\n", A22)
usedMemory2, _ = tracemalloc.get_traced_memory()
tracemalloc.stop()
time4 = time.time()
```

Listing 4: Calculo del tiempo y uso de memoria con la multiplicación con bucles

3 Comparación de tiempo y rendimiento

El tiempo de ejecución y el uso de memoria se midieron para matrices cuadradas de tamaños crecientes. Los resultados se graficaron y se observó una mejora significativa en la versión vectorizada. El código es el siguiente:

```
print("\nTiempo con funciones prexistentes: ", format(time2-time1))
print("\nTiempo con bucles hard-coded: ", format(time4-time3))

print("\nMemoria con funciones prexistentes: ", usedMemory1)
print("\nMemoria con bucles hard-coded: ", usedMemory2)
```

Listing 5: Calculo de tiempo y rendimiento

3.1 Resultados gráficos

Generamos los resultados de la siguiente forma:

```
def generarGrafica(n, tiempos_bucles, tiempos_numpy,
    memorias_bucles, memorias_numpy):
    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.plot(n, tiempos_bucles, label="Bucles", marker="o")
    plt.plot(n, tiempos_numpy, label="Vectorizado", marker="o")
    plt.title("Tiempo de Ejecuci n")
    plt.xlabel("Tama o de la Matriz ({t} x {t})".format(t=n))
    plt.ylabel("Tiempo (s)")
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(n, memorias_bucles, label="Bucles", marker="o")
    plt.plot(n, memorias_numpy, label="Vectorizado", marker="o")
    plt.title("Uso de Memoria")
    plt.xlabel("Tama o de la Matriz ({t} x {t})".format(t=n))
    plt.ylabel("Memoria (KB)")
    plt.legend()

    plt.tight_layout()
    plt.savefig("comparacion_matrices.pdf")
    plt.show()
```

Listing 6: Calculo de tiempo y rendimiento

La gráfica generada muestra la complejidad en el tiempo y memoria de ambos algoritmos:

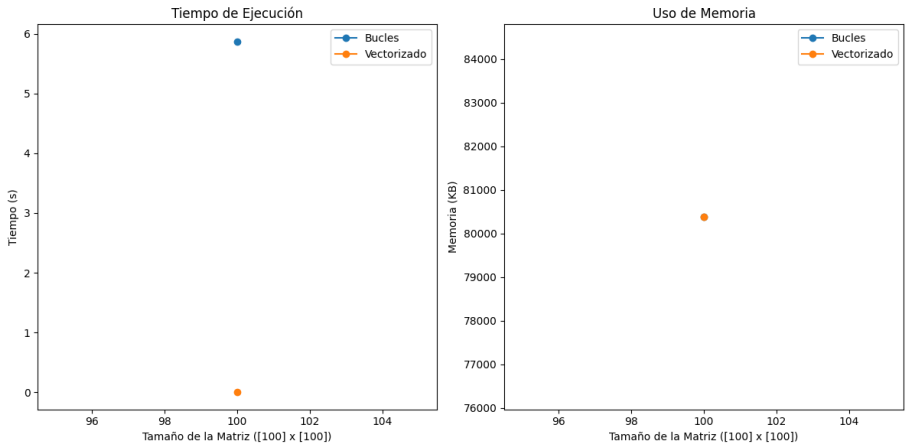

Figure 1: Comparación de tiempos de ejecución y uso de memoria

4 Conclusiones

El enfoque vectorizado con `numpy` supera ampliamente a la implementación con bucles en términos de tiempo de ejecución y uso de memoria. Esto refuerza la importancia de utilizar herramientas optimizadas para problemas computacionalmente intensivos.