Reconocimiento de objetos por transformers ViT y procesamiento de imagenes para vision computacional

Gleddynuri M. Picha[1]
School of Computer Science
Universidad Nacional de San Agustín de Arequipa
[1]gpichac@unsa.edu.pe,

***Resumen* – El reconocimiento de objetos en imágenes es esencial en la visión por computadora, con aplicaciones que van desde la seguridad hasta la medicina. Los transformers Vision Transformer (ViT) han surgido como una alternativa prometedora para esta tarea, aprovechando el aprendizaje profundo y la atención. Sin embargo, enfrentan desafíos significativos. La arquitectura original de los ViT fue diseñada para la clasificación de imágenes, lo que dificulta tratar con objetos múltiples o superpuestos. La falta de información espacial detallada limita su desempeño. Integrar información de posición sigue siendo un desafío, lo que dificulta la localización precisa de objetos. La eficiencia computacional es preocupante debido a su alto costo en comparación con otras arquitecturas. Además, la generalización a diferentes condiciones de iluminación es un desafío importante. Aunque prometen en la detección de objetos, los transformers ViT deben abordar estos desafíos para desbloquear su potencial en aplicaciones del mundo real.**

***Palabras clave* – Vision computacional, Reconocimiento de objetos, ViT, procesamiento de imagenes.**

***Abstract* – Object recognition in images is essential in computer vision, with applications ranging from security to medicine. Vision Transformer (ViT) transformers have emerged as a promising alternative for this task, leveraging deep learning and attention. However, they face significant challenges. The original architecture of ViT was designed for image classification, making it difficult to deal with multiple or overlapping objects. The lack of detailed spatial information limits its performance. Integrating positional information remains a challenge, hindering precise object localization. Computational efficiency is concerning due to its high cost compared to other architectures. Additionally, generalization to different lighting conditions is a significant challenge. Although promising in object detection, ViT transformers must address these challenges to unlock their potential in real-world applications.**

***Index Terms*—Computer vision, object recognition, ViT, image processing.**

I. Problema 1

Estos bucles están recorriendo la matriz Implementacion Análisis Acceso a la memoria:

En el primer conjunto de bucles, los elementos de la matriz A[i][j] se acceden fila por fila, lo cual es generalmente eficiente en sistemas que almacenan matrices en memoria en orden de filas (row-major order). Este acceso secuencial es óptimo para la caché, ya que los datos contiguos se cargan juntos en la caché. En el segundo conjunto de bucles, el acceso a la matriz se realiza columna por columna. En sistemas que almacenan matrices en orden de filas, este tipo de acceso es menos eficiente para la caché, ya que los elementos accedidos no están contiguos en memoria. Esto puede resultar en más fallos de caché, lo que reduce el rendimiento. Complejidad temporal:

Ambos conjuntos de bucles tienen una complejidad temporal de

O(MAX2), ya que hay dos bucles anidados que recorren la matriz A, que es de tamaño MAX×MAX. Esto significa que la cantidad de operaciones crece cuadráticamente con respecto a MAX.

Eficiencia de caché:

El primer conjunto de bucles será más eficiente en términos de acceso a la caché en la mayoría de los sistemas, debido al acceso secuencial a los datos en memoria. El segundo conjunto de bucles, al acceder a los datos en forma dispersa, puede experimentar más fallos de caché, lo que lo hace menos eficiente [1].

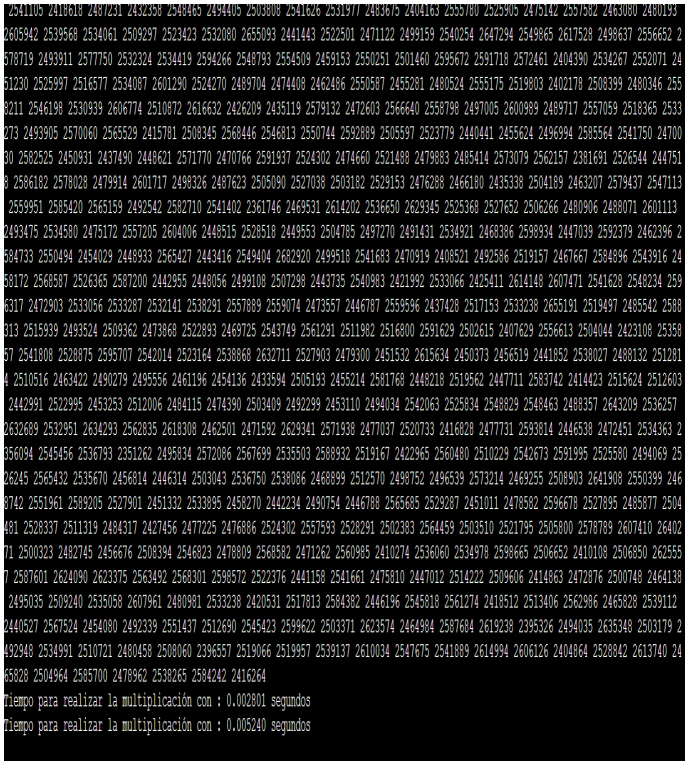
```
// Primera  multiplicación loop
void loop1(int A[MAX][MAX], int x[MAX], int y[MAX]) {

    // Multiplicación de matriz
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            y[i] += A[i][j] * x[j];
        }
    }
}

// Segunda multiplicación
void loop2(int A[MAX][MAX], int x[MAX], int y[MAX]) {

    // Multiplicación de matriz
    for (int j = 0; j < MAX; j++) {
        for (int i = 0; i < MAX; i++) {
            y[i] += A[i][j] * x[j];
        }
    }
}
```

resultado




```cpp
// Multiplicación clásica de matrices con tres bucles anidados
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        for (int k = 0; k < size; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

// Función para imprimir una matriz
void print(int** matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cout << matrix[i][j] << " ";
        }
```

resultado

II. Problema 2

A. *implementacion*

la matriz A tiene dimensiones N×M, la matriz B tiene dimensiones M×P, y la matriz C es el resultado de tamaño N×P.

Complejidad Temporal La multiplicación clásica de matrices requiere tres bucles anidados:

El primer bucle itera sobre las filas de la matriz C (índice i). El segundo bucle itera sobre las columnas de la matriz C (índice j). El tercer bucle itera sobre las columnas de A y las filas de B (índice k).

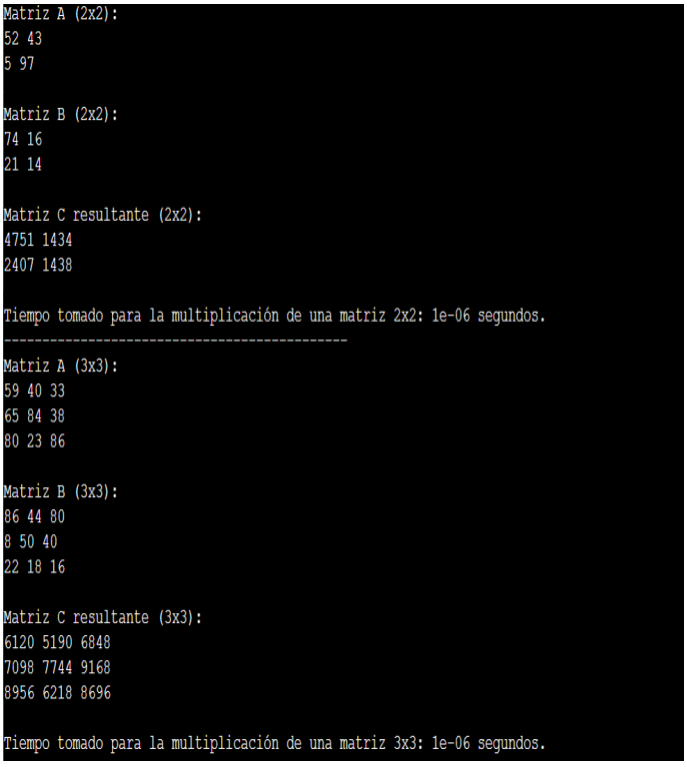
Esto significa que el tiempo de ejecución crece cúbicamente con respecto al tamaño de la matriz, lo que puede volverse prohibitivo para matrices grandes.

Acceso a la Memoria y Eficiencia de Caché El rendimiento de la multiplicación clásica de matrices depende no solo de la complejidad temporal O(N 3), sino también de la eficiencia en el uso de la caché y el acceso a la memoria.

Acceso secuencial en matriz A:

El acceso a A[i][k] es eficiente, ya que se itera sobre cada fila de la matriz A de manera secuencial. Esto es favorable para sistemas que almacenan las matrices en orden de filas (row-major order).

B puede requerir cargar nuevos bloques de memoria, lo que incrementa el tiempo de ejecución.



```cpp
// Primera  multiplicación loop
void loop1(int A[MAX][MAX], int x[MAX], int y[MAX]) {

    // Multiplicación de matriz
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            y[i] += A[i][j] * x[j];
        }
    }
}

// Segunda multiplicación
void loop2(int A[MAX][MAX], int x[MAX], int y[MAX]) {

    // Multiplicación de matriz
    for (int j = 0; j < MAX; j++) {
        for (int i = 0; i < MAX; i++) {
            y[i] += A[i][j] * x[j];
        }
    }
}
```

```
Matriz A (6x6):
33 13 57 45 7 1
9 45 18 92 3 100
33 4 15 91 37 67
28 44 56 51 42 24
85 41 83 40 57 57
98 42 22 6 86 28

Matriz B (6x6):
58 46 24 75 89 26
75 21 81 89 64 18
7 91 61 62 93 55
85 77 95 19 68 4
75 18 97 48 75 82
27 33 79 50 59 67

Matriz C resultante (6x6):
7665 10602 10355 8407 12714 5048
14768 13435 21890 12688 17736 9348
14638 12851 19558 10616 17504 9642
13449 12783 18467 13673 18550 9856
17800 18311 24256 21516 28266 16166
16704 10326 18220 18094 21966 13466

Tiempo tomado para la multiplicación de una matriz 6x6: 2e-06 segundos.
```

III. Problema 3

La multiplicación de matrices por bloques es una optimización sobre la multiplicación clásica que mejora el acceso a la memoria caché, lo que puede resultar en un mejor rendimiento en sistemas modernos. Descripción del Algoritmo de Multiplicación por Bloques En la multiplicación por bloques, en lugar de operar sobre matrices completas, se dividen las matrices A, B, y C en pequeños bloques o submatrices, y la multiplicación se realiza bloque por bloque. Esto permite que los datos que se necesitan estén más localizados en la caché, reduciendo los accesos a la memoria principal.

Para una matriz de tamaño N×N, dividimos las matrices en bloques de tamaño B×B.

Complejidad Temporal La multiplicación por bloques sigue utilizando tres bucles principales (para los bloques ii, jj, kk) y otros tres bucles internos (para los elementos dentro de los bloques). Como tal, el número de operaciones sigue siendo proporcional a N 3 en el peor de los casos, es decir, la complejidad temporal sigue siendo:

O(N 3) Sin embargo, la optimización proviene de la localización de los datos en caché, lo que mejora el rendimiento práctico en sistemas con jerarquía de memoria (caché y RAM).

```cpp
    // Multiplicación por bloques usando seis bucles anidados
    for (int ii = 0; ii < size; ii += blockSize) {
        for (int jj = 0; jj < size; jj += blockSize) {
            for (int kk = 0; kk < size; kk += blockSize) {
                for (int i = ii; i < min(ii + blockSize, size); i++) {
                    for (int j = jj; j < min(jj + blockSize, size); j++) {
                        for (int k = kk; k < min(kk + blockSize, size); k++) {
                            C[i][j] += A[i][k] * B[k][j];
                        }
                    }
                }
            }
        }
    }
}

// Función para imprimir una matriz
void print(int** matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cout << matrix[i][j] << " ";
        }
```

resultado

```
Multiplicación para matriz 2x2:
Tiempo (clásica): 1e-06 segundos.
Tiempo (por bloques): 1e-06 segundos.
----------------------------------------
Multiplicación para matriz 3x3:
Tiempo (clásica): 0 segundos.
Tiempo (por bloques): 2e-06 segundos.
----------------------------------------
Multiplicación para matriz 4x4:
Tiempo (clásica): 1e-06 segundos.
Tiempo (por bloques): 1e-06 segundos.
----------------------------------------
Multiplicación para matriz 6x6:
Tiempo (clásica): 2e-06 segundos.
Tiempo (por bloques): 4e-06 segundos.
----------------------------------------
Multiplicación para matriz 10x10:
Tiempo (clásica): 6e-06 segundos.
Tiempo (por bloques): 1.4e-05 segundos.
----------------------------------------
```

IV. Problema 4

Comparación entre los dos métodos

Aspecto	Multiplicación clásica	Multiplicación por bloques
Complejidad de tiempo	$O(n^3)$	$O(n^3)$
Complejidad espacial	$O(n^2)$	$O(n^2)$
Rendimiento en caché	Menor aprovechamiento	Mejor aprovechamiento
Accesos a memoria	Desordenados, más costosos	Accesos más ordenados, mejor caché
Número de operaciones	n^3 multiplicaciones/sumas	n^3 multiplicaciones/sumas
Eficiencia para matrices grandes	Baja debido a la falta de optimización de la caché	Alta por la mejora en accesos a la caché

Análisis de rendimiento Acceso a la memoria: La multiplicación clásica realiza muchos accesos a memoria que no están optimizados para la caché, ya que los elementos de la matriz A y B son accedidos de forma aleatoria a través de filas y columnas. Esto genera muchos fallos de caché, lo que puede hacer que el rendimiento real sea más lento, especialmente para matrices grandes.

La multiplicación por bloques agrupa los elementos en bloques más pequeños y realiza las operaciones en esos bloques, lo que reduce la cantidad de accesos a la memoria y aumenta la localidad de caché. Como resultado, aunque ambas versiones tienen la misma complejidad asintótica, la multiplicación por bloques suele ser más rápida en la práctica, especialmente para matrices grandes, debido a un mejor uso de la caché.

Efecto de n grande: Para matrices pequeñas, la diferencia en rendimiento entre la multiplicación clásica y la por bloques puede ser mínima. Sin embargo, para matrices grandes (como 100x100 o más), la multiplicación por bloques será significativamente más rápida debido a la optimización de los accesos a memoria y la reducción de fallos de caché. 5. Conclusión Complejidad teórica: Ambas versiones tienen la misma complejidad asintótica O(n 3) en términos del número de operaciones. Rendimiento práctico: La multiplicación por bloques tiende a ser más eficiente en la práctica para matrices grandes, ya que hace un mejor uso de la caché del procesador, reduciendo el tiempo de acceso a la memoria.

V. Problema 5

Al ejecutar ambos algoritmos (la multiplicación clásica y la multiplicación por bloques) y evaluarlos usando Valgrind y KCachegrind, debes seguir estos pasos. El propósito es obtener una evaluación precisa del desempeño en términos de la memoria caché, uso de CPU, e instrucciones ejecutadas. Valgrind permitirá recopilar información de cachegrind (el cual se enfoca en la cache y la CPU), y KCachegrind es una herramienta gráfica para analizar los resultados.

Análisis de los Resultados:

Instrucciones ejecutadas: Compara el número total de instrucciones entre los dos algoritmos. La multiplicación por bloques, a pesar de tener más bucles, debería mostrar una ligera mejora en términos de eficiencia de caché. Fallos de caché: El principal objetivo de la multiplicación por bloques es reducir los fallos de caché. Observa cómo disminuyen estos en comparación con la multiplicación clásica. Rendimiento general de la CPU: Revisa el número de ciclos de CPU y el uso de las instrucciones de la CPU para verificar cuál es más eficiente en términos de rendimiento de procesamiento.

link de github https://github.com/Gleddy-mar/CPD-LAB1.git

Conclusiones:

Memoria caché: La multiplicación por bloques debería mostrar una menor cantidad de cache misses debido a su mejor manejo de la localidad de los datos. Ciclos de CPU: El número total de ciclos de CPU puede ser mayor en la multiplicación clásica, mientras que la multiplicación por bloques, aunque más compleja, podría reducir la cantidad de fallos de caché, lo que mejora el tiempo global de ejecución. Uso de memoria: Ambos algoritmos deberían usar una cantidad similar de memoria, pero la multiplicación por bloques optimiza mejor su acceso.

References

[1] P. Pacheco, *An introduction to parallel programming*. Elsevier, 2011.