

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

Gleddynuri M. Picha¹

School of Computer Science

Universidad Nacional de San Agustín de Arequipa

¹gpichac@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 imágenes.

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

$O(MAX^2)$, ya que hay dos bucles anidados que recorren la matriz A , que es de tamaño $MAX \times MAX$. 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

```

2541105 2418618 2407231 2432938 2546465 2494405 2503808 2541826 2531977 2463675 2404165 2535780 2525905 2475142 2537582 2463080 2480193
2605942 2539568 2534061 2509287 2523423 2532080 2655093 2441443 2522501 2471122 2499159 2540254 2647294 2549865 2617528 2498637 2556652 2
578719 2493911 2577750 2532324 2534419 2594266 2540793 2554509 2459153 2550251 2501460 2595672 2591718 2572461 2404390 2534267 2552071 24
51230 2525297 2516577 2534087 2601290 2524270 2489704 2474408 2462486 2550587 2455381 2480524 2555175 2519803 2402178 2508399 2480346 255
8211 2546198 2530939 2606774 2510872 2616632 2426209 2435119 2579132 2472603 2566640 2558798 2497005 2600989 2489717 2557059 2518365 2533
273 2493905 2570060 2565529 2415781 2508345 2568446 2546813 2550744 2592889 2505597 2523779 2440441 2455624 2496994 2595564 2541750 24700
30 2582525 2450931 2437490 2448621 2571770 2470766 2591597 2524302 2474660 2521488 2479883 2485414 2573079 2562157 2381691 2526544 244751
9 2586182 2578028 2479914 2601717 2498326 2487623 2505090 2527038 2503182 2529153 2476288 2466180 2435338 2504189 2463207 2579437 2547113
2559951 2585920 2565153 2492542 2582710 2541402 2361746 2469531 2614202 2536650 2629345 2525368 2527652 2506266 2480906 2488071 2601113
2493475 2534580 2475172 2557205 2604006 248515 2528518 2449553 2504785 2497270 2491431 2534921 2468386 2598934 2447039 2592379 2462396 2
584733 2530494 2454029 2448933 2565427 2443416 2549404 2682920 2499518 2541683 2470919 2408521 2492586 2519157 2467667 2584896 2543916 24
68172 2568587 2526365 2587200 2442955 2440056 2499108 2507298 2443735 2540983 2421992 2533066 2425411 2614148 2607471 2541628 2548234 259
6317 2472903 2533056 2533287 2532141 2538291 2557889 2559074 2473557 2446787 2559596 2437428 2517153 2533238 2655191 2519497 2485542 2588
913 2515939 2493524 2509362 2473868 2522893 2469725 2543745 2561291 2511982 2516800 2591629 2502615 2407629 2556613 2504044 2423108 25358
57 2541808 2528875 2595707 2542014 2523164 2538868 2632711 2527903 2479300 2451532 2615634 2450373 2456519 2441852 2538027 2488132 251281
4 2510516 2463422 2490279 2495556 2461196 2454136 2433594 2505193 2455214 2581768 2448218 2519562 2447711 2583742 2414423 2515624 2512603
2442981 2522995 2453253 2512006 2484115 2474390 2503409 2492289 2453110 2494034 2542063 2525834 2548829 2549463 2488357 2643209 2536257
2632689 2522951 2634293 2562835 2618308 2462501 2471592 2629341 2571938 2477037 2520793 2416828 2477731 2593814 2446538 2472451 2534363 2
566094 2545456 2536783 2351262 2495834 2572086 2567689 2535503 2588932 2519167 2422965 2560480 2510229 2542673 2591995 2525580 2494069 25
8245 2565432 2535670 2456814 2446314 2503043 2536750 2538086 2468899 2512570 2498752 2496539 2573214 2469255 2508903 2641908 2550399 246
8742 2551951 2589205 2527901 2451332 2533895 2458270 2442234 2490754 2446788 2565685 2529287 2451011 2478582 2596678 2527895 2485877 2504
481 2528337 2511319 2484317 2427456 2477225 2476886 2524302 2557593 2528291 2502383 2564459 2503510 2521795 2505800 2578789 2607410 26402
71 2500323 2482745 2456676 2508394 2546823 2478809 2568582 2471262 2560985 2410274 2536060 2534978 2598665 2506652 2410108 2506850 262555
7 2587681 2624090 2623375 2563492 2568301 2598572 2522376 2441158 2541661 2475810 2447012 2514222 2509606 2414863 2472876 2500748 2464138
2495035 2509240 2535058 2607961 2480181 2533238 2420531 2517813 2584382 2446196 2545818 2561274 2418512 2513406 2562886 2465828 2539112
2440527 2567524 2454080 2492339 2551437 2512690 2545423 2599622 2503371 2623574 2464984 2587684 2619238 2395326 2494035 2635348 2503179 2
482948 2543991 2510721 2480458 2500060 2396557 2519066 2519957 2539137 2610034 2547675 2541889 2614994 2606126 2404064 2528842 2613740 24
65828 2504964 2585700 2478962 2538265 2584242 2416264
Tiempo para realizar la multiplicación con : 0.002801 segundos
Tiempo para realizar la multiplicación con : 0.005240 segundos

```

```

// 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. implementación

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

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.

```

Matriz A (2x2):
52 43
5 97

Matriz B (2x2):
74 16
21 14

Matriz C resultante (2x2):
4751 1434
2407 1438

Tiempo tomado para la multiplicación de una matriz 2x2: 1e-06 segundos.
-----
Matriz A (3x3):
59 40 33
65 84 38
80 23 86

Matriz B (3x3):
86 44 80
8 50 40
22 18 16

Matriz C resultante (3x3):
6120 5190 6848
7098 7744 9168
8956 6218 8696

Tiempo tomado para la multiplicación de una matriz 3x3: 1e-06 segundos.

```

```
// 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.

que los datos que se necesitan estén más localizados en la caché, reduciendo los accesos a la memoria principal.

Para una matriz de tamaño $N \times N$, dividimos las matrices en bloques de tamaño $B \times B$.

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).

```
// 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

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

```

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 kcache (el cual se enfoca en la caché 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.