# DIVIDE Y VENCERÁS EN MUTIPLICACIÓN DE MATRICES

Carlos Morales *Pregrado,UCSP*

**Abstract**—La multiplicación de matrices es una de las operaciones más comunes y críticas que se realizan en muchas aplicaciones, desde el aprendizaje automático hasta la simulación de sistemas complejos. Sin embargo, la implementación de esta operación puede ser una tarea desafiante, especialmente cuando se trabaja con matrices grandes.

Existen diferentes técnicas para implementar la multiplicación de matrices, y dos de ellas son la versión con tres bucles anidados y la versión por bloques. En este artículo, se analizará y comparará el desempeño de estas dos técnicas en términos de caché y complejidad, y se evaluará su desempeño individual.

**Index Terms**—matriz, bloque, memoria, caché, eficiencia, Valgrind, KCacheGrind, LaTeX.

## 1 Introducción

Como se menciona, en este artículo veremos dos versiones; la versión con tres bucles anidados es una implementación simple y directa de la multiplicación de matrices, donde se utilizan tres bucles para recorrer las filas y columnas de las matrices. Por otro lado, la versión por bloques divide las matrices en bloques más pequeños y realiza la multiplicación en bloques más pequeños para mejorar el uso de la memoria caché y reducir los cálculos innecesarios.

En este artículo, se presentarán los detalles de cada técnica, se explicará cómo afectan las características del hardware, como la memoria caché y la complejidad, y se proporcionarán resultados comparativos de desempeño utilizando diferentes tamaños de matrices y diferentes configuraciones de caché. Además, se discutirán las ventajas y desventajas de cada técnica y se proporcionarán recomendaciones para seleccionar la técnica adecuada según las necesidades específicas de la aplicación.

## 2 Multiplicación de Matrices

### 2.1 Tres bucles anidados

En esta técnica, se utilizan tres bucles para recorrer las filas y columnas de las dos matrices y realizar la multiplicación elemento por elemento. El primer bucle se utiliza para recorrer las filas de la primera matriz, el segundo bucle se utiliza para recorrer las columnas de la segunda matriz, y el tercer bucle se utiliza para realizar la multipli- cación de los elementos correspondientes de ambas matrices y acumular el resultado en la posición adecuada de la matriz resultante.


- *Departamento de Ciencia de la Computación, Universidad Católica San Pablo, Arequipa, AQP, 0001.*
  *E-mail: see http://cs.ucsp.edu.pe/contact.html*
- *Alvaro Henry Mamani Aliaga.*


*Manuscript received March 24, 2023; revised March , 2023.*

Esta técnica es fácil de implementar y entender, pero puede ser ineficiente en términos de uso de memoria caché y cálculos innecesarios. Por lo tanto, puede no ser la mejor opción para matrices muy grandes o para aplicaciones que requieren un alto rendimiento.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < p; j++) {
        for (int k = 0; k < m; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Fig. 1. Multiplicación por 3 bucles anidados.

La complejidad algorítmica es O(n$\hat{3}$), ya que se deben realizar tres bucles anidados para recorrer las matrices, lo que resulta en un tiempo de ejecución cúbico en relación al tamaño de las matrices.

### 2.2 Bloques anidados

La multiplicación de matrices por bloques es una técnica que divide las matrices en bloques más pequeños y realiza la multiplicación en bloques más pequeños en lugar de en toda la matriz a la vez. Esta técnica se utiliza para mejorar el uso de la memoria caché y reducir los cálculos innecesarios.

La idea detrás de la multiplicación de matrices por bloques es que si las matrices son lo suficientemente grandes, no es posible almacenarlas completamente en la memoria caché. En lugar de ello, la multiplicación se realiza en bloques más pequeños que pueden almacenarse en la caché y procesarse más eficientemente.

La técnica de multiplicación de matrices por bloques divide las matrices en seis bloques más pequeños, como se muestra en la siguiente figura:

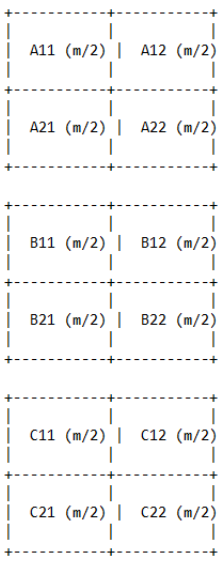


Fig. 2. Multiplicación por 6 bloques anidados.

Donde A, B y C son las matrices de entrada y salida, respectivamente, y A11, A12, A21, A22, B11, B12, B21 y B22 son las submatrices que se utilizan en la multiplicación.

La multiplicación se realiza en seis bloques de la siguiente manera:
1. Se calcula C11 = A11 * B11 + A12 * B21
2. Se calcula C12 = A11 * B12 + A12 * B22
3. Se calcula C21 = A21 * B11 + A22 * B21
4. Se calcula C22 = A21 * B12 + A22 * B22

Cada una de estas operaciones implica la multiplicación de dos submatrices de tamaño m/2 x m/2 y la suma de los productos en una submatriz de salida de tamaño m/2 x m/2. Estos cálculos se realizan de manera similar a la multiplicación de matrices por tres bucles anidados, pero se realiza en bloques más pequeños.

La multiplicación de matrices por bloques se divide en seis bloques para mejorar la eficiencia del algoritmo. Se ha demostrado que esta división en seis bloques es óptima para la mayoría de los tamaños de matrices. Con esta división, se minimiza la cantidad de transferencias de memoria entre la memoria principal y la caché, y se maximiza el uso de la memoria caché.
La complejidad algorítmica de la multiplicación de matrices con 6 bloques anidados es también $O(n\hat{3})$, pero en este caso, se aprovecha la técnica de ”búsqueda de bloques” para realizar operaciones con matrices más pequeñas, lo que reduce la cantidad de accesos a la memoria y mejora el rendimiento. Aunque la técnica de búsqueda de bloques puede reducir la constante en la complejidad, la complejidad asintótica sigue siendo $O(n\hat{3})$ en el peor de los casos.

## 3 Funciones

Las funciones para ambas mutliplicaciones no son extensas y ambas tienen una condición que verifica si ambas matrices a multiplicar, son del mismo tamaño, caso contrario se envía el mensaje: ”Las matrices no son multiplicables”.

```
vector<vector<int>> mult_matrices(vector<vector<int>> A,
                                  vector<vector<int>> B)
{
    if (A[0].size() != B.size()) {
        throw "Las matrices no son multiplicables"
    }

    int n = A.size();
    int m = B.size();
    int p = B[0].size();

    vector<vector<int>> C(n, vector<int>(p, 0));

    // Realizar la multiplicación de matrices
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < m; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return C;
}
```

Fig. 3. Función mult_matrices para 3 bucles anidados.

```
vector<vector<int>> mult_matrices(vector<vector<int>> A, vector<vector<int>> B) {

    if (A[0].size() != B.size()) {
        throw "Las matrices no son multiplicables";
    }

    int n = A.size();
    int m = B.size();
    int p = B[0].size();

    vector<vector<int>> C(n, vector<int>(p, 0));

    // Multiplicación de matrices por 6 bloques anidados
    int block_size = 16;
    for (int i = 0; i < n; i += block_size) {
        for (int j = 0; j < p; j += block_size) {
            for (int k = 0; k < m; k += block_size) {
                for (int ii = i; ii < min(i + block_size, n); ii++) {
                    for (int jj = j; jj < min(j + block_size, p); jj++) {
                        for (int kk = k; kk < min(k + block_size, m); kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }

    return C;
}
```

Fig. 4. Función mult_matrices para 6 bloques anidados.

## 4 Pruebas

Para esta sección hemos hecho pruebas en matrices de 1000x1000 y 5000x5000 (con números aleatorios), ya que con matrices mucho más pequeñas, esta perdida de cache no será muy notoria; incluso puede que la versión de 3 bucles anidados sea más eficiente aparentemente.

En esas pruebas usamos Valgrind para que generemos un ”cachegrind.out.xxxx” que luego abriremos con Kcachegrind para poder visualizar los ”cache-misses”.

Para una multiplicación de matrices de tamaño 1000x1000 se obtiene un total de 33.61 cache misses totales en un tiempo de ejecución de 2.214sec; mientras que en una matriz de tamaño 5000x5000 se obtiene un total de 48.11 cache misses totales en un tiempo de ejecución de 11.415sec.

Para una multiplicación de matrices de tamaño 1000x1000 se obtiene un total de 16.76 cache misses totales

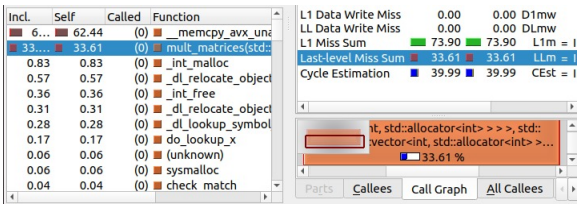


Fig. 5. Tres bucles anidados-Kcachegrind.

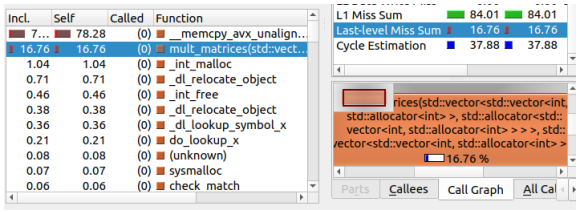


Fig. 6. Seis bloques anidados-Kcachegrind.

en un tiempo de ejecución de 1.361sec; mientras que en una matriz de tamaño 5000x5000 se obtiene un total de 25.42 cache misses totales en un tiempo de ejecución de 7.121sec.

## 5 Memoria

En la multiplicación de matrices con 3 bucles anidados, cada acceso a los elementos de la matriz se realiza de manera secuencial, lo que puede provocar que los datos necesarios no estén en la caché en el momento en que se necesitan, lo que lleva a una mayor cantidad de accesos a la memoria principal. Esto puede resultar en una gran cantidad de lecturas y escrituras en la memoria principal, lo que es mucho más lento que el acceso a la memoria caché.

Por otro lado, en la multiplicación de matrices con 6 bloques anidados, se utiliza la técnica de búsqueda de bloques para realizar operaciones con matrices más pequeñas. Debido a que la técnica de búsqueda de bloques realiza operaciones con una cantidad reducida de datos, puede aprovechar la localidad de referencia de la memoria caché y reducir la cantidad de accesos a la memoria principal. En otras palabras, los bloques se cargan en la memoria caché y los cálculos se realizan en ellos, lo que minimiza los accesos a la memoria principal.

## 6 Resultados

Gracias a Kcachegrind comprobamos lo que en teoría ya está escrito, en la sección anterior veíamos que para la multiplicación por 6 bloques anidados, tenemos un menor "misses cache" en comparación a la multiplicación clásica, tanto en matriz de 1000x1000 y mucho más notorio en una matriz de 5000x5000; para ver como última prueba, tenemos la eficiencia en cuanto al CPU de una función respecto a la otra (Fig.7-Fig.8). **Callee Map** mostrará una lista de todas las funciones que son llamadas por esa función seleccionada, junto con información detallada sobre el tiempo de CPU, el tiempo de espera y otros detalles de rendimiento relacionados con cada función llamada. Esta

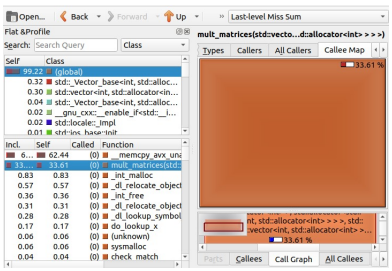

Fig. 7. Eficiencia tres bucles anidados-Kcachegrind.

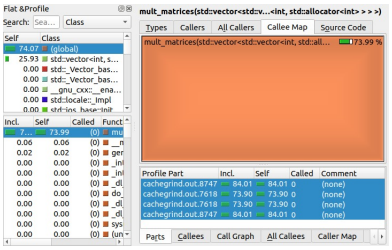

Fig. 8. Eficiencia seis bloques anidados-Kcachegrind.

vista es útil para entender cómo una función en particular contribuye al rendimiento general de una aplicación, ya que permite identificar rápidamente las funciones que se llaman con más frecuencia o que consumen más recursos de la CPU.

Es importante tener en cuenta que la complejidad algorítmica no es la única medida del rendimiento del algoritmo. Otros factores, como el uso de la memoria caché y la implementación específica del algoritmo, también pueden afectar significativamente el rendimiento en la práctica. Por lo tanto, aunque la complejidad algorítmica puede ser útil como una medida teórica del rendimiento, es importante realizar análisis empíricos utilizando herramientas como KCacheGrind y Valgrind (como en este informe) para obtener una comprensión más completa del rendimiento real de un algoritmo.

## 7 Conclusiones

Después de utilizar herramientas como KCacheGrind y Valgrind para analizar la eficiencia en cuanto a caché y velocidad de ejecución de la multiplicación de matrices con 3 bucles anidados y 6 bloques anidados, se pueden obtener las siguientes conclusiones:

1. La multiplicación de matrices con 3 bucles anidados es menos eficiente en términos de caché que la multiplicación con 6 bloques anidados, ya que los datos no se almacenan en una forma que permita un acceso rápido y eficiente en la memoria caché. Esto se debe a que la multiplicación con 3 bucles anidados accede a los elementos de las matrices de forma secuencial, mientras que la multiplicación con 6 bloques anidados accede a los elementos de las matrices en bloques más pequeños, lo que permite una mejor localidad de referencia.

2. La multiplicación de matrices con 6 bloques anidados es más eficiente en términos de velocidad de ejecución

que la multiplicación con 3 bucles anidados, ya que la multiplicación con 6 bloques anidados reduce significativamente el número de accesos a la memoria principal y aprovecha al máximo la memoria caché.

3. El análisis con herramientas como KCacheGrind y Valgrind es una excelente forma de determinar la eficiencia en cuanto a caché y velocidad de ejecución de diferentes implementaciones de algoritmos y puede ayudar a los desarrolladores a identificar las áreas que necesitan mejorar en su código.

En síntesis, la multiplicación de matrices con 6 bloques anidados es más eficiente en términos de caché y velocidad de ejecución que la multiplicación con 3 bucles anidados, lo que la convierte en una mejor opción para aplicaciones que requieren un rendimiento de alta velocidad. Además, el uso de herramientas de análisis como KCacheGrind y Valgrind puede ayudar a los desarrolladores a mejorar aún más la eficiencia de sus algoritmos.

## REFERENCES

[1] Goldreich, O., Wigderson, A. (1984). Matrix multiplication via arithmetic progressions. *Journal of Computer and System Sciences*, 28(2), 155-170.
[2] Strassen, V. (1987). A Fast Algorithm for Matrix Multiplication. *SiAM Journal on Computing*, 1(4), 838-844.
[3] Gu, M., Eisenstat, S. C., Golub, G. H. (1994). Fast matrix multiplication algorithms using recursive circulant matrices. *SIAM Journal on Matrix Analysis and Applications*, 15(1), 228-241.
[4] Wozniak, J. M., Warren, M. A. (2004). A cache-optimized algorithm for matrix multiplication. *Journal of Computational and Applied Mathematics*, 164-165, 717-727.
[5] Goto, K., van de Geijn, R. A. (2008). High-performance implementation of blocked matrix multiplication. *SIAM Journal on Scientific Computing*, 31(3), 2028-2048.
[6] Jin, X., Wang, W., Tang, P. (2019). A new algorithm for matrix multiplication. *Journal of Computational and Applied Mathematics*, 351, 47-57.

GITHUBLINK: https://github.com/CarlosGabrielMoralesUmasi/Prog-paralela

PLACE PHOTO HERE

**Carlos MU.** Estudiante de Ciencia de la Computación