Pruebas sobre el comportamiento de la memoria caché aplicado en Matrices

Zamalloa Molina Sebastian Agenor,
szamalloam@unsa.edu.pe

I. Introducción

La multiplicación de matrices es una de las operaciones más fundamentales y ampliamente utilizadas en diversas aplicaciones científicas y de ingeniería. Este artículo compara dos enfoques para realizar esta operación: la multiplicación de matrices clásica de tres bucles anidados y una versión optimizada por bloques. Además, se analiza el comportamiento del acceso a la memoria caché en una implementación con bucles anidados, evaluando cómo este afecta al rendimiento en términos de movimiento de datos entre la memoria principal y la caché.

II. Implementación de Bucles Anidados

Consideramos dos versiones de bucles anidados. La primera recorre la matriz de forma eficiente (orden fila por fila), mientras que la segunda accede a los elementos en una forma menos óptima (acceso por columna).

II-A. Código de Bucles Anidados

El siguiente código corresponde a las dos versiones de bucles anidados. La primera accede a la memoria en bloques contiguos, mientras que la segunda realiza saltos de memoria, afectando el rendimiento:

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

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

Listing 1. Implementación de bucles anidados en C++

II-B. Análisis del Rendimiento

El primer par de bucles accede a la memoria de manera contigua, lo que hace que el acceso a la caché sea eficiente. Sin embargo, el segundo par de bucles accede a los elementos de la matriz columna por columna, lo que genera múltiples *cache misses*. Esto resulta en un mayor tiempo de ejecución debido a la menor eficiencia en el acceso a la memoria.

III. Multiplicación de Matrices Clásica

La multiplicación de matrices clásica utiliza tres bucles anidados. Cada iteración del bucle interno realiza una multiplicación y una suma. Este enfoque tiene una complejidad temporal de $O(n^3)$.

III-A. Implementación

El siguiente código muestra la implementación clásica de la multiplicación de matrices:

```
void multiplyMatrices
(const vector<vector<double>>& A,
 const vector<vector<double>>& B,
 vector<vector<double>>& C){
    int n = A.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Listing 2. Implementación clásica de multiplicación de matrices

III-B. Complejidad y Eficiencia

El acceso a los elementos de la matriz `B` se realiza por columnas, lo cual resulta en un acceso no contiguo y, por tanto, un rendimiento subóptimo debido a las múltiples transferencias de datos entre la memoria principal y la caché. La complejidad temporal de este algoritmo es $O(n^3)$, ya que cada elemento de la matriz resultado requiere n multiplicaciones.

IV. Multiplicación de Matrices por Bloques

La multiplicación de matrices por bloques optimiza el acceso a la memoria caché dividiendo las matrices en subbloques más pequeños, lo que permite reutilizar los datos almacenados en la caché de manera más eficiente. Este enfoque también tiene una complejidad temporal de $O(n^3)$, pero mejora la eficiencia al reducir las transferencias de datos entre la memoria principal y la caché.

IV-A. Implementación

El siguiente código muestra la implementación por bloques de la multiplicación de matrices:

```
void multiplyMatricesBlocked
(const vector<vector<double>>& A,
 const vector<vector<double>>& B,
 vector<vector<double>>& C,
 int blockSize) {
    int n = A.size();
    for (int ii = 0; ii < n; ii += blockSize) {
        for (int jj = 0; jj < n; jj += blockSize) {
            for (int kk = 0; kk < n; kk += blockSize
                ) {
                for (int i = ii; i < min(ii +
                    blockSize, n); ++i) {
                    for (int j = jj; j < min(jj +
                        blockSize, n); ++j) {
                        for (int k = kk; k < min(kk
                            + blockSize, n); ++k) {
                            C[i][j] += A[i][k] * B[k
                                ][j];
                        }
                    }
                }
            }
        }
    }
}
```

Listing 3. Implementación de multiplicación por bloques

IV-B. Complejidad y Eficiencia

El acceso a los elementos de las matrices `A` y `B` se realiza en pequeños bloques que caben en la caché, lo que permite reutilizar los datos más eficientemente. Esto reduce las *cache misses* y mejora el rendimiento en comparación con la multiplicación clásica.

V. Resultados

En la Tabla I se muestran los tiempos de ejecución para matrices de diferentes tamaños.

Tamaño de la matriz	Tiempo Clásico (ms)	Tiempo Bloques (ms)
100x100	10	12
200x200	91	100
500x500	1596	1541
1000x1000	17759	12528

Tabla I
Comparación de tiempos de ejecución entre la multiplicación clásica y por bloques

VI. Analisis basado en Valgrid y kcachegrind

VI-A. Multiplicacion de matrices clásica

El análisis con *Callgrind* muestra los siguientes resultados para la multiplicación de matrices clásica:

- **Instrucciones totales**: 1,183,756,121 instrucciones.
- **Uso de caché**: El algoritmo experimenta un número elevado de reemplazos en caché, lo que se traduce en un alto número de fallos de caché. Esto se debe a que cada elemento de la matriz es accedido múltiples veces, pero los accesos no están organizados de forma que maximicen la localidad temporal o espacial de los datos.
- **Fallos de caché**: La caché L1 experimenta un número moderado de fallos debido a la falta de reutilización eficiente de los datos cargados. Cada acceso a la matriz A, B y C obliga a la recarga de bloques en la caché.
- **Ciclos de CPU**: Los ciclos de CPU son elevados debido a la alta cantidad de accesos a memoria y los fallos de caché asociados. Esto afecta directamente el tiempo de ejecución, haciendo que este algoritmo sea menos eficiente para matrices de gran tamaño.

VI-B. Multiplicación de matrices por bloques

El análisis con *Callgrind* para la multiplicación de matrices por bloques arroja los siguientes resultados:

- **Instrucciones totales**: 1,429,967,139 instrucciones (aproximadamente un 21 % más que el algoritmo clásico).
- **Uso de caché**: A pesar de ejecutar más instrucciones, el uso eficiente de la caché reduce el número de fallos en caché L1 y L2, optimizando el tiempo de ejecución general. Al acceder a los datos en bloques, se reutilizan los valores que ya están en caché, lo que reduce significativamente el número de accesos a la memoria principal.
- **Fallos de caché**: Se observa una disminución considerable en los fallos de caché en comparación con la multiplicación clásica. El algoritmo por bloques maximiza la localidad de los datos, permitiendo que más datos relevantes permanezcan en caché entre iteraciones.
- **Ciclos de CPU**: Aunque el número de instrucciones es mayor, la reducción en los fallos de caché y la optimización del uso de la memoria resulta en menos ciclos de CPU que el algoritmo clásico, lo que disminuye el tiempo total de ejecución.

VI-C. Comparación detallada del rendimiento

A continuación se presenta una comparación cuantitativa de las métricas clave de ambos algoritmos:

Métrica	Clásica	Bloques
Instrucciones Totales	1,183,756,121	1,429,967,139
Fallos de Caché (L1)	Elevados	Reducidos
Uso de Caché	Ineficiente	Eficiente
Ciclos de CPU	Alto	Moderado
Tiempo de Ejecución (estimado)	Lento	Rápido

Tabla II
Comparación del rendimiento entre los algoritmos de multiplicación de matrices clásica y por bloques.

VI-C1. Análisis del uso de caché: La clave del éxito del algoritmo de *Multiplicación por bloques* es su capacidad de optimizar el uso de la caché. Al procesar los elementos de las matrices en bloques, se mejora la localidad temporal y espacial de los datos, lo que minimiza los fallos de caché. Esto se traduce en menos accesos a la memoria principal, que es significativamente más lenta que la caché, resultando en un mejor rendimiento general.

VI-C2. Impacto en el rendimiento del sistema: El mayor número de instrucciones ejecutadas en el algoritmo por bloques podría interpretarse inicialmente como un inconveniente. Sin embargo, el análisis detallado demuestra que esta sobrecarga es compensada por una mejor eficiencia en el uso de

la memoria caché, lo que reduce el número de accesos a la memoria principal. Dado que los accesos a la memoria principal son costosos en términos de ciclos de CPU, la disminución de estos accesos permite que el algoritmo por bloques complete la multiplicación en menos tiempo que el algoritmo clásico.

VII. Conclusión

El análisis muestra que, aunque el algoritmo de *Multiplicación de matrices clásica* tiene un menor número de instrucciones ejecutadas, su ineficiencia en el uso de la caché resulta en un mayor tiempo de ejecución. Por otro lado, el algoritmo de *Multiplicación de matrices por bloques* realiza más instrucciones, pero su eficiente uso de la memoria caché reduce significativamente los ciclos de CPU y el tiempo de ejecución, haciéndolo más adecuado para aplicaciones que manejan matrices de gran tamaño. En escenarios donde la eficiencia de la caché es crucial, la *Multiplicación de matrices por bloques* es claramente la mejor opción.

VIII. Código fuente

El código completo de la implementación se encuentra disponible en el siguiente repositorio: GitHub - Cache-Memory-Matrix-Multiplication.