Informe de Laboratorio: Pruebas sobre el comportamiento de la Memoria Caché

1st Justo Alfredo Perez Choque
dept. Ciencia de la Computación
Universidad Católica San Pablo
Arequipa, Perú
justo.perez@ucsp.edu.pe

I. Introducción

El objetivo de este laboratorio es estudiar el comportamiento de la memoria caché. Se implementaron dos ejercicios, primero una comparación de dos variantes de bucles anidados en C++, y se compararon sus tiempos de ejecución para observar cómo las diferencias en el acceso a la memoria pueden influir en el desempeño. Luego, se analizó la eficiencia de dos algoritmos de multiplicación de matrices, uno clásico y otro basado en bloques, evaluando su rendimiento en términos de *cache hits*, *cache misses*, y la gestión de la localidad espacial y temporal. El repositorio de todo el trabajo se encuentra en el repositorio[1].

II. Implementación

A. *Código del libro: Bucles anidados*

El código utilizado para este experimento consiste en dos funciones (`test1` y `test2`) que realizan operaciones similares sobre una matriz `A` y dos vectores `x` e `y`.

La principal diferencia entre `test1` y `test2` radica en el orden de los bucles anidados:

- En `test1`, el bucle externo itera sobre el índice `i`, mientras que el bucle interno lo hace sobre el índice `j`.
- En `test2`, el orden de los bucles se invierte; el bucle externo itera sobre `j` y el interno sobre `i`.

Ambas funciones realizan la operación:

$$y[i] += A[i][j] \times x[j]$$

```
void test1(double** A, double* x, double* y, int MAX
    ) {
    for (int i = 0; i < MAX; i++)
        for (int j = 0; j < MAX; j++)
            y[i] += A[i][j] * x[j];
}
void test2(double** A, double* x, double* y, int MAX
    ) {
    for (int j = 0; j < MAX; j++)
        for (int i = 0; i < MAX; i++)
            y[i] += A[i][j] * x[j];
}
```

B. *Código Multiplicación Matrices*

Se han implementado dos algoritmos para la multiplicación de matrices: la multiplicación clásica utilizando tres bucles anidados y la multiplicación por bloques utilizando seis bucles anidados.

```
void multMatrizClasica(int **m1, int **m2, int **res
    , int N ){

    for(int i = 0; i < N;i++){
        for(int j = 0; j < N; j++){
            for(int k = 0; k < N; k++){
                *(*(res+i)+j) += *(*(m1+i)+k) * *(*(
                    m2+k)+j);
            }
        }
    }
    return;
}


void multMatrizBloque(int **m1, int **m2, int **res,
     int N , int S){
    for (int ii = 0; ii < N; ii += S) {
        for (int jj = 0;jj < N; jj += S) {
            for (int kk = 0; kk < N; kk += S) {
                for (int i = ii;i < std::min(ii+S,N)
                    ; ++i) {
                    for (int j = jj;j < std::min(jj+
                        S, N); ++j) {
                        for (int k = kk; k < std::
                            min(kk +S, N);++k) {
                            res[i][j] += m1[i][k]*m2
                                [k][j];
                        }
                    }
                }
            }
        }
    }
}
```

III. Resultados y Analisis

A. *Código del libro: Bucles anidados*

Los resultados presentados en la Tabla I muestran que la primera forma tiene un mejor rendimiento que la segunda, especialmente a medida que el tamaño de la matriz aumenta. Esto se debe principalmente a cómo ambas formas aprovechan la memoria caché. La primera forma tiene un alto número de *cache hits* debido a que accede secuencialmente a la memoria por filas, lo que explota la *localidad espacial* y *temporal*. Esto significa que los datos adyacentes y recientemente utilizados están más frecuentemente disponibles en la caché, reduciendo la necesidad de acceder a la memoria principal.

Por otro lado, la segunda forma accede a la matriz por columnas, lo que provoca un mayor número de *cache misses*

Valor de MAX	Primera Forma (segundos)	Segunda Forma (segundos)
100	4.1969e-05	4.3164e-05
200	0.000173058	0.000200513
400	0.000657152	0.000898779
800	0.00266543	0.00427397
1600	0.010799	0.0286272
3200	0.0436007	0.134383
6400	0.176388	0.566714
12800	0.714516	3.0812
25600	3.40258	17.342

TABLE I
TIEMPO DE EJECUCIÓN EN SEGUNDOS PARA LAS DOS FORMAS DE BUCLES ANIDADOS.

porque los elementos de una columna no están contiguamente almacenados en la memoria. Esto dificulta el aprovechamiento de la *localidad espacial*, resultando en una menor eficiencia en el uso de la caché.

Además, la forma en que la caché maneja las escrituras (*write-through* o *write-back*) y el *cache mapping* también pueden influir, pero el factor determinante aquí es el patrón de acceso a la memoria. La primera forma, al acceder por filas, es más compatible con estos esquemas y resulta en un menor tiempo de ejecución. En sistemas que utilizan memoria virtual, el rendimiento también puede verse afectado, ya que matrices más grandes podrían requerir el uso de disco, lo cual es más costoso en términos de tiempo.

B. *Multiplicación de Matrices*

1) ***Analisis Simple****:* La Tabla II muestra una comparación de los tiempos de ejecución para ambos enfoques de multiplicación de matrices.

Tamaño de Matriz	Multiplicación Clásica (s)	Multiplicación por Bloques (s)
10	6.151e-06	9.809e-06
20	5.0881e-05	7.7514e-05
40	0.000365963	0.000581237
80	0.0030288	0.00471272
160	0.026546	0.0365391
320	0.233593	0.308757
640	2.08351	2.48982
1280	21.7345	19.8072
2560	253.301	156.013

TABLE II
TIEMPOS DE EJECUCIÓN PARA AMBOS MÉTODOS DE MULTIPLICACIÓN DE MATRICES.

La multiplicación de matrices clásica tiene una complejidad algorítmica de $O(N^3)$, donde N es el tamaño de la matriz. Este algoritmo itera a través de cada elemento de la matriz resultante y, para cada uno, realiza una suma de productos sobre la fila de la primera matriz y la columna de la segunda matriz. Debido a su acceso a memoria en un patrón no secuencial (acceso por columnas), este enfoque puede resultar en un alto número de *cache misses*, ya que los datos necesarios pueden no estar disponibles en la caché, obligando al algoritmo a acceder a la memoria principal con frecuencia.

En contraste, la multiplicación de matrices por bloques también tiene una complejidad algorítmica de $O(N^3)$, pero su rendimiento es optimizado al dividir las matrices en submatrices o bloques. Este enfoque mejora la localidad espacial y temporal, permitiendo que los datos accedidos recientemente se mantengan en la caché por más tiempo. Durante la ejecución, los bloques se cargan en la caché, lo que minimiza la cantidad de accesos a la memoria principal y reduce significativamente el número de *cache misses* en comparación con la multiplicación clásica.

Al ejecutar ambos algoritmos paso a paso, se puede observar que la multiplicación por bloques mejora el movimiento de datos entre la memoria principal y la caché. Esto se traduce en un rendimiento superior, especialmente en matrices de mayor tamaño, donde el acceso secuencial a los bloques reduce la latencia de memoria y optimiza el uso de la caché. En resumen, aunque ambos métodos tienen la misma complejidad algorítmica, la implementación por bloques ofrece una mejora considerable en la eficiencia práctica debido a un mejor manejo de la memoria caché.

2) ***Analisis con herramientas Valgrind y Kcachegrind****:* La Tabla III muestra un resumen de los resultados con una matriz de tamaño 640, estos datos fueron extraídos con las herramientas *Valgrind* y *Kcachegrind* donde la multiplicación por bloques ofrece un manejo superior de la caché, particularmente en el contexto de matrices de gran tamaño. Este algoritmo se beneficia significativamente de un mejor aprovechamiento de la localidad espacial y temporal, lo cual se manifiesta en un número considerablemente menor de *cache misses* en L1 para lectura de datos (**D1mr**). Esto sugiere que los datos necesarios están más frecuentemente disponibles en la caché, reduciendo la necesidad de acceder a la memoria principal y mejorando así el rendimiento general.

Por otro lado, la versión clásica de la multiplicación de matrices presenta un número mucho mayor de *cache misses* en L1, lo que penaliza su desempeño al trabajar con matrices de gran tamaño. Este incremento en los *cache misses* se debe a un menor aprovechamiento de la localidad temporal y espacial, lo que obliga al algoritmo a realizar más accesos a la memoria principal, incrementando los tiempos de ejecución.

En cuanto a la gestión del ciclo de escritura, ambos algoritmos muestran una eficiencia notable, ya que no experimentan *cache misses* en la caché de escritura (**D1mw**). Esto indica un manejo adecuado de las políticas de escritura, como *write-through* y *write-back*, que aseguran que los datos se escriban de manera eficiente en la memoria.

Aunque la multiplicación por bloques ejecuta un mayor número de instrucciones, su mejor manejo de la caché mitiga el impacto negativo que estas podrían tener en los tiempos de ejecución. El costo adicional en términos de operaciones requeridas para manejar los bloques es compensado por la reducción en los *cache misses* y la consecuente mejora en el rendimiento.

IV. CONCLUSIONES

En el análisis del intercambio de índices para acceso a matrices, se concluye que la primera forma de acceder a los elementos por filas es significativamente más eficiente que la

Evento	multMatrizClasica	multMatrizBloque
Instruction Fetch (Ir)	400,361,816	625,237,123
Data Read Access (Dr)	168,160,804	249,675,557
Data Write Access (Dw)	16,080,406	61,047,228
L1 Instr. Fetch Miss (I1mr)	5	7
L1 Data Read Miss (D1mr)	534,971	68,520
L1 Data Write Miss (D1mw)	0	0
LL Instr. Fetch Miss (IL1mr)	0	7
LL Data Read Miss (DL1mr)	0	0
Cycle Estimation (CEst)	405,711,976	625,923,093

TABLE III
COMPARACIÓN DE EVENTOS EN MULTIPLICACIÓN DE MATRICES CON UN TAMAÑO DE 640.

segunda forma, que accede por columnas. Esto se debe al mejor aprovechamiento de la localidad espacial y temporal, lo que resulta en un mayor número de *cache hits* y, por ende, en un menor tiempo de ejecución.

En cuanto a la multiplicación de matrices, el uso de un algoritmo de bloques demuestra una clara ventaja en términos de eficiencia en el manejo de la caché. A pesar de ejecutar más instrucciones, su capacidad para reducir los *cache misses* le permite superar a la multiplicación clásica, especialmente con matrices de gran tamaño. La optimización del uso de la memoria caché es, por lo tanto, un factor clave para mejorar el rendimiento en operaciones matriciales complejas.

REFERENCES

[1] Justo Perez. Repositorio de github. https://dtellogaete.medium.com/regresi%C3%B3n-lineal-en-python-y-r-machine-learning-01-ff9d9077f8f, 2024.