LABORATORIO: Pruebas sobre el comportamiento de la memoria caché

Cristhian David Huanca Olazabal

15 de septiembre de 2025

1. Implementar y comparar los 2-bucles anidados FOR presentados en el cap. 2 del libro, pag 22.

La implementación se hará usando C++ y algunas modificaciones que pide el código de la pagina 22 principalmente en la forma en la que asignamos valores a las matrices y vectores.

```cpp
#include <iostream>
using namespace std;
int main()
{
  double A[MAX][MAX], y[MAX], x[MAX];
  for (int i = 0; i < MAX; i++)
  {
    x[i] = (double)rand() / RAND_MAX;
    y[i] = 0.0;
    for (int j = 0; j < MAX; j++)
    {
      A[i][j] = (double)rand() / RAND_MAX;
    }
  }

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

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

En comparación, podemos concluir que el primer bucle será más eficiente y veloz que el segundo, debido a que aprovecha mejor las características de la memoria caché del CPU. Como se explicó en teoría, la memoria caché trabaja por bloques de datos llamados cache lines o bloques de caché.

Cuando accedemos a un dato, por ejemplo A[0][0], no solo se carga ese elemento, sino también un segmento contiguo de memoria en la caché. Este comportamiento se conoce como localidad espacial.

Por lo tanto, en el primer bucle habrá una menor cantidad de cache misses, ya que recorremos la matriz de forma contigua en memoria (por filas), aprovechando los datos ya cargados en la caché. En cambio, en el segundo bucle accedemos a la matriz por columnas, lo cual rompe la localidad espacial en C/C++ (donde los arreglos se almacenan por filas), generando más cache misses y menor eficiencia.

2. Implementar en C/C++ la multiplicación de matrices clásica, la versión de tres bucles anidados y evaluar su desempeño considerando diferentes tamaños de matriz.

Para solucionar este problema, se definen 3 matrices, la última será la matriz de la respuesta. Para poder evaluar su desempeño se hace uso de un cronómetro que se encargará de tomar el tiempo. De

primera mano sabemos que la versión a fuerza bruta para resolver este problema no es eficiente y es:

$$O(n^3)$$

Esto debido a que tenemos que recorrer todos n veces cada una de las n filas y n columnas para poder hallar la multiplicación de matrices.

```cpp
#include <iostream>
#include <chrono>
#include <ctime>
using namespace std;
using namespace std::chrono;

#define MAX 800

int A[MAX][MAX], B[MAX][MAX], Y[MAX][MAX];

void print(int Y[MAX][MAX])
{
  for(int i = 0; i < MAX; i++)
  {
    for(int j = 0; j < MAX; j++)
    {
      cout << Y[i][j] << "␣";
    }
    cout << endl;
  }
}
int main()
{
  for(int i = 0; i < MAX; i++)
  {
    for(int j = 0; j < MAX; j++)
    {
      A[i][j] = rand() % 5;
      B[i][j] = rand() % 5;
      Y[i][j] = 0;
    }
  }
  auto start = high_resolution_clock::now();

  for(int i = 0; i < MAX; i++)
  {
    for(int j = 0; j < MAX; j++)
    {
      for(int k = 0; k < MAX; k++)
      {
        Y[i][j] += A[i][k] * B[k][j];
      }
    }
  }
  auto end = high_resolution_clock::now();

  auto time = duration_cast<milliseconds>(end - start).count();
  cout << time << "␣" << MAX;

  return 0;
}
```

Con la ejecución del código obtenemos la siguiente figura con diferentes configuraciones de n.

Cuadro 1: Tiempo de ejecución de la multiplicación de matrices clásica

Tamaño n	**Tiempo (ms)**
100	4
400	274
800	2398
1200	8232
1600	21890

3. Implementar la versión por bloques (investigar en internet), seis bucles anidados, evaluar su desempeño y compararlo con la multiplicación de matrices clásica.

En este caso tenemos la versión por bloques. Según se puede ver en la teoría, es necesario dividir la matriz en una cantidad de bloques que sea múltiplo de sus dimensiones. De esta forma podemos aprovechar mejor las ventajas de la memoria caché.

La idea, en el caso del código que estoy mostrando, es la implementación con 6 bucles anidados: los 3 primeros corresponden a los bloques de la matriz y los otros 3 recorren el interior de dichos bloques. En otras palabras, convertimos la matriz original en una más pequeña compuesta por elementos que, en realidad, son bloques. Estos bloques, a su vez, son submatrices. Es por eso que necesitamos los 6 bucles.

```
#include <iostream>
#include <chrono>
#include <ctime>
using namespace std;
using namespace std::chrono;

#define MAX 1600
#define BLOCK_SIZE 160

int A[MAX][MAX], B[MAX][MAX], Y[MAX][MAX];

void print(int Y[MAX][MAX])
{
  for(int i = 0; i < MAX; i++)
  {
    for(int j = 0; j < MAX; j++)
    {
      cout << Y[i][j] << " ";
    }
    cout << endl;
  }
}

int main()
{
  for(int i = 0; i < MAX; i++)
  {
    for(int j = 0; j < MAX; j++)
    {
      A[i][j] = rand() % 5;
      B[i][j] = rand() % 5;
      Y[i][j] = 0;
    }
  }

  auto start = high_resolution_clock::now();

  for(int ii = 0; ii < MAX; ii += BLOCK_SIZE)
  {
    for(int jj = 0; jj < MAX; jj += BLOCK_SIZE)
    {
      for(int kk = 0; kk < MAX; kk += BLOCK_SIZE)
      {
        for(int i = ii; i < ii + BLOCK_SIZE; i++)
        {
          for(int j = jj; j < jj + BLOCK_SIZE; j++)
          {
            for(int k = kk; k < kk + BLOCK_SIZE; k++)
            {
              Y[i][j] += A[i][k] * B[k][j];
            }
          }
        }
      }
    }
  }

  auto end = high_resolution_clock::now();
```

```
  auto time = duration_cast<milliseconds>(end - start).count();
  cout << time << " " << MAX << " (Block size: " << BLOCK_SIZE << ")";

  return 0;
}
```

Con la ejecución del código podemos apreciar que la mejora es más evidente a medida que aumenta el tamaño de la matriz. También es importante mencionar que la cantidad de elementos por bloque tiene un impacto significativo en el desempeño. Si el bloque es demasiado pequeño, el rendimiento puede ser muy similar al de la versión clásica. Del mismo modo, si el bloque es demasiado grande o cercano al tamaño total de la matriz, se pierde la ventaja del enfoque por bloques.

Por tal motivo, para una matriz de 1600×1600 de enteros se eligió una configuración de 160 elementos por bloque, es decir, un 10 % del tamaño de la matriz. De esta forma se obtuvo una mejora relevante en comparación con el método clásico. Para evidenciar estas mejoras se presenta la tabla de comparación con los tiempos respectivos.

Cuadro 2: Comparación de tiempos de ejecución entre multiplicación clásica y por bloques (10 %)

Tamaño n	**Tiempo clásico (ms)**	**Tiempo (bloques 10 %)**
100	4	5
400	274	315
800	2398	2312
1200	8232	7803
1600	21890	18203

Ambos casos mantienen una complejidad de $O(n^3)$. Sin embargo, la versión por bloques tiene la ventaja de reutilizar valores previamente cargados en la memoria caché, reduciendo la cantidad de *cache misses*. En cambio, en la versión clásica se producen fallos de caché en casi todos los accesos. Esta diferencia explica la variación en los tiempos de ejecución entre ambos métodos.

4. Ejecutar ambos algoritmos paso a paso, y analizar el movimiento de datos entre la memoria principal y la memoria cache. Hacer una evaluación de acuerdo a la complejidad algorítmica.

En el caso del algoritmo clásico con complejidad $O(n^3)$, tenemos el siguiente código que se encarga de multiplicar las matrices.

```
for(int i = 0; i < MAX; i++)
{
  for(int j = 0; j < MAX; j++)
  {
    for(int k = 0; k < MAX; k++)
    {
      Y[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Esta ejecución no es óptima debido a que tenemos muchos accesos a memoria RAM, es decir, tenemos cache miss. Analicemos esto con mayor detalle: Cuando:

- **Iteración $k = 0$:** $Y[0][0] += A[0][0] \times B[0][0]$
  - Acceso $A[0][0]$: **CACHE MISS** → Carga línea de caché desde memoria
  - Acceso $B[0][0]$: **CACHE MISS** → Carga línea de caché desde memoria
  - Acceso $Y[0][0]$: **CACHE MISS** → Carga línea de caché desde memoria
- **Iteración $k = 1$:** $Y[0][0] += A[0][1] \times B[1][0]$
  - Acceso $A[0][1]$: **CACHE HIT** (mismo bloque que $A[0][0]$)

- Acceso $B[1][0]$: **CACHE MISS** (diferente fila de B)
- Acceso $Y[0][0]$: **CACHE HIT**

▪ **Iteración** $k = 2$**:** $Y[0][0] += A[0][2] \times B[2][0]$

- Acceso $A[0][2]$: **CACHE HIT**
- Acceso $B[2][0]$: **CACHE MISS** (nueva fila de B)
- Acceso $Y[0][0]$: **CACHE HIT**

No es eficiente ya que tenemos que cargar nuevas líneas que no se usarán en el acceso a la matriz B. Esto es debido a que este se mueve por filas y no por columnas como si se hace en la matriz A.

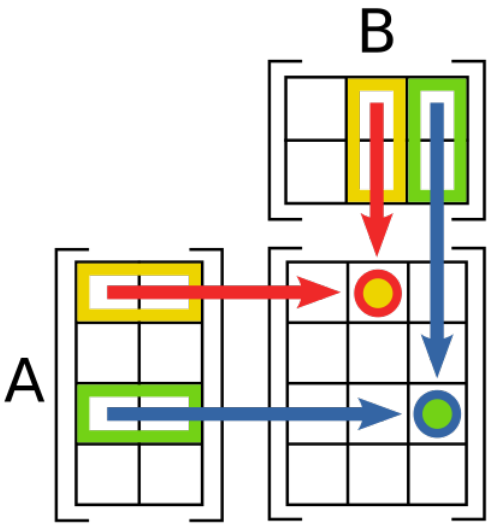

Figura 1: Multiplicación de matrices clásica

En el caso de la versión con bloques, tenemos que la complejidad sigue siendo $O(n^3)$. Sin embargo, este será menor cuando tenemos matrices grandes en dimensión.

```
for(int ii = 0; ii < MAX; ii += BLOCK_SIZE)
{
  for(int jj = 0; jj < MAX; jj += BLOCK_SIZE)
  {
    for(int kk = 0; kk < MAX; kk += BLOCK_SIZE)
    {
      for(int i = ii; i < ii + BLOCK_SIZE; i++)
      {
        for(int j = jj; j < jj + BLOCK_SIZE; j++)
        {
          for(int k = kk; k < kk + BLOCK_SIZE; k++)
          {
            Y[i][j] += A[i][k] * B[k][j];
          }
        }
      }
    }
  }
}
```

Esto se debe a que abordamos el problema del método clásico. Ahora estamos manejando bloques y podemos reutilizar los elementos que ya estén cargados en la memoria caché. De esta forma, la configuración de los bloques tendrá un gran efecto en la ejecución del código, ya que bloques muy grandes o muy pequeños no aprovecharán de forma eficiente las ventajas de la caché. En el caso de mi ejemplo, coloqué a los bloques con un tamaño equivalente al 10 % de n.

Volviendo al ejemplo, con una dimensión de $n = 1600$, teníamos que el tamaño del bloque sería de 160 valores. Es decir, tendríamos la siguiente configuración:

Bloque	Rango de Filas	Rango de Columnas	Tamaño
$A_{0,0}$	$[0, 159]$	$[0, 159]$	160×160
$A_{0,1}$	$[0, 159]$	$[160, 319]$	160×160
$A_{0,2}$	$[0, 159]$	$[320, 479]$	160×160
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$A_{1,0}$	$[160, 319]$	$[0, 159]$	160×160
$A_{1,1}$	$[160, 319]$	$[160, 319]$	160×160
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$A_{9,9}$	$[1440, 1599]$	$[1440, 1599]$	160×160

Como podemos apreciar, los bloques se pueden reutilizar,

- **Iteración de bloques:** $ii = 0,\ jj = 0,\ kk = 0$
  - Procesamos el bloque $A[0 : \text{BLOCK\_SIZE}][0 : \text{BLOCK\_SIZE}]$, $B[0 : \text{BLOCK\_SIZE}][0 : \text{BLOCK\_SIZE}]$ y actualizamos $Y[0 : \text{BLOCK\_SIZE}][0 : \text{BLOCK\_SIZE}]$.
- **Iteración interna** $k = 0$**:** $Y[0][0] += A[0][0] \times B[0][0]$
  - Acceso $A[0][0]$: **CACHE MISS** $\rightarrow$ carga la primera línea del bloque de A.
  - Acceso $B[0][0]$: **CACHE MISS** $\rightarrow$ carga la primera línea del bloque de B.
  - Acceso $Y[0][0]$: **CACHE MISS** $\rightarrow$ carga línea de Y.
- **Iteración interna** $k = 1$**:** $Y[0][0] += A[0][1] \times B[1][0]$
  - Acceso $A[0][1]$: **CACHE HIT** (está en el mismo bloque de A).
  - Acceso $B[1][0]$: **CACHE HIT** (pertenece al bloque cargado de B).
  - Acceso $Y[0][0]$: **CACHE HIT**.
- **Iteración interna** $k = 2$**:** $Y[0][0] += A[0][2] \times B[2][0]$
  - Acceso $A[0][2]$: **CACHE HIT** (mismo bloque).
  - Acceso $B[2][0]$: **CACHE HIT** (aún en el bloque cargado de B).
  - Acceso $Y[0][0]$: **CACHE HIT**.
- **Conclusión del bloque:**
  - Los accesos a A y B generan **pocos cache misses** iniciales.
  - Los siguientes accesos dentro del mismo bloque son en su mayoría **cache hits**, gracias a la localidad espacial y temporal.
  - Se aprovecha mejor la memoria caché comparado con la versión clásica.

5. Ejecutar ambos algoritmos utilizando las herramientas valgrind y kcachegrind para obtener una evaluación mas precisa de su desempeño en términos de .

Con la aplicación de valgrind con los siguientes parámetros: con $n = 800$, 10 % de tamaño de los bloques en su versión. Se tienen los siguientes resultados:

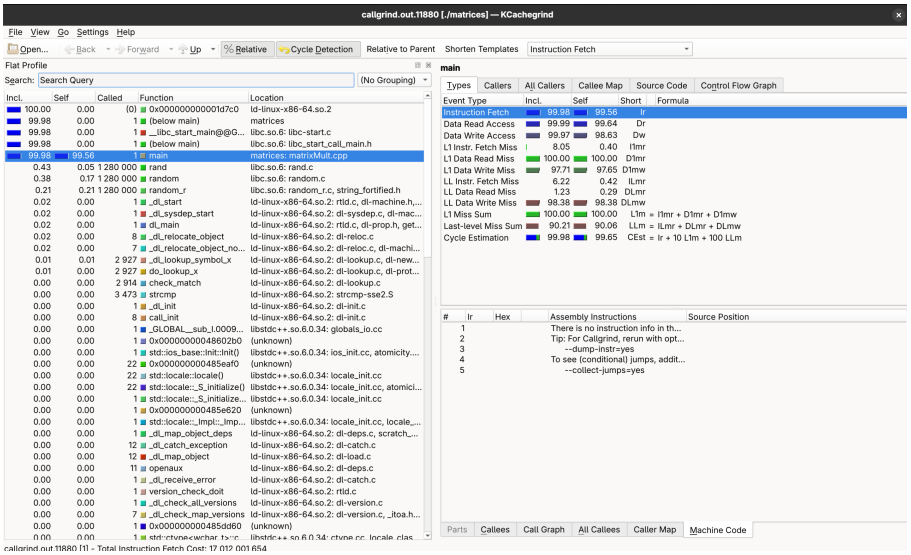

Figura 3: Kcachegrind para la versión clásica

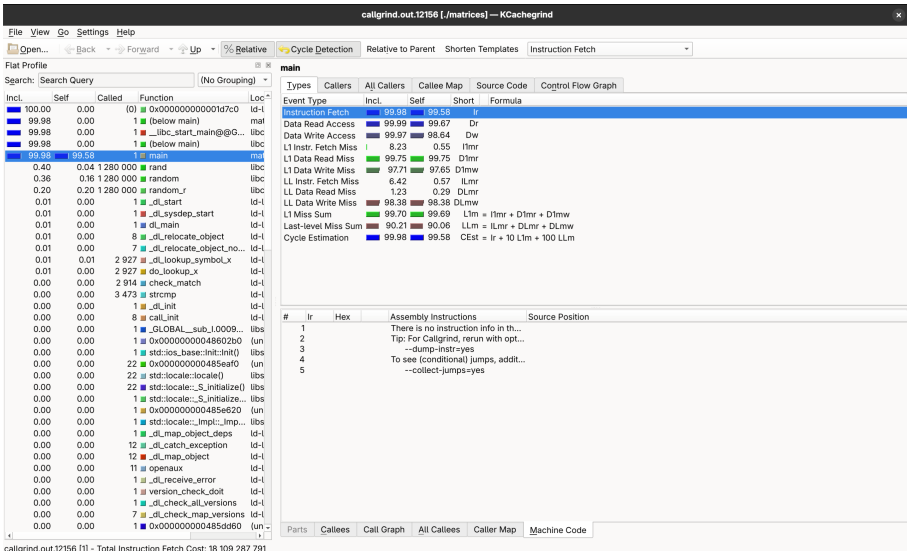

Figura 2: Kcachegrind para la versión con bloques

```
chuancao26@fedora:~/Documents/Unsa2025B/paralela/lab1$ g++ -O0 -g matrixMult.cpp -o matrices
chuancao26@fedora:~/Documents/Unsa2025B/paralela/lab1$ valgrind --tool=cachegrind --cache-sim=yes ./matrices
==11773== Cachegrind, a high-precision tracing profiler
==11773== Copyright (C) 2002-2024, and GNU GPL'd, by Nicholas Nethercote et al.
==11773== Using Valgrind-3.25.1 and LibVEX; rerun with -h for copyright info
==11773== Command: ./matrices
==11773==
--11773-- warning: L3 cache found, using its data for the LL simulation.
70878 800==11773==
==11773== I refs:        17,012,012,283
==11773== I1  misses:             2,015
==11773== LLi misses:             1,916
==11773== I1  miss rate:           0.00%
==11773== LLi miss rate:           0.00%
==11773==
==11773== D refs:         7,216,981,458  (6,689,366,720 rd   + 527,614,738 wr)
==11773== D1  misses:       528,909,112  (  528,786,451 rd   +     122,661 wr)
==11773== LLd misses:           131,372  (        9,614 rd   +     121,758 wr)
==11773== D1  miss rate:            7.3% (          7.9%     +         0.0%  )
==11773== LLd miss rate:            0.0% (          0.0%     +         0.0%  )
==11773==
==11773== LL refs:          528,911,127  (  528,788,466 rd   +     122,661 wr)
==11773== LL misses:            133,288  (       11,530 rd   +     121,758 wr)
==11773== LL miss rate:             0.0% (          0.0%     +         0.0%  )
chuancao26@fedora:~/Documents/Unsa2025B/paralela/lab1$
```

Figura 4: Valgrind para Versión Clásica

```
chuancao26@fedora:~/Documents/Unsa2025B/paralela/lab1$ g++ -O0 -g matrixBloques.cpp -o matrices
chuancao26@fedora:~/Documents/Unsa2025B/paralela/lab1$ valgrind --tool=cachegrind --cache-sim=yes ./matrices
==11705== Cachegrind, a high-precision tracing profiler
==11705== Copyright (C) 2002-2024, and GNU GPL'd, by Nicholas Nethercote et al.
==11705== Using Valgrind-3.25.1 and LibVEX; rerun with -h for copyright info
==11705== Command: ./matrices
==11705==
--11705-- warning: L3 cache found, using its data for the LL simulation.
72056 800 (Block size: 80)==11705==
==11705== I refs:        18,109,298,420
==11705== I1  misses:             2,019
==11705== LLi misses:             1,920
==11705== I1  miss rate:           0.00%
==11705== LLi miss rate:           0.00%
==11705==
==11705== D refs:         7,771,785,087  (7,238,329,906 rd   + 533,455,181 wr)
==11705== D1  misses:         8,184,491  (    8,061,828 rd   +     122,663 wr)
==11705== LLd misses:           131,373  (        9,614 rd   +     121,759 wr)
==11705== D1  miss rate:            0.1% (          0.1%     +         0.0%  )
==11705== LLd miss rate:            0.0% (          0.0%     +         0.0%  )
==11705==
==11705== LL refs:            8,186,510  (    8,063,847 rd   +     122,663 wr)
==11705== LL misses:            133,293  (       11,534 rd   +     121,759 wr)
==11705== LL miss rate:             0.0% (          0.0%     +         0.0%  )
chuancao26@fedora:~/Documents/Unsa2025B/paralela/lab1$
```

Figura 5: Valgrind para versión con bloques

- Ambos métodos ejecutan un número similar de instrucciones, consistente con la complejidad $O(n^3)$.
- En la versión clásica se observa una tasa de fallos en caché de primer nivel de datos (**D1 miss rate**) de aproximadamente 7,3 %, mientras que en la versión por bloques disminuye a solo 0,1 %.
- La cantidad de fallos en caché de último nivel (LL) se mantiene casi constante, pero el acceso por bloques aprovecha mejor la localidad espacial y temporal, reduciendo drásticamente los fallos en L1.
- Esto confirma que la estrategia por bloques mejora la eficiencia en el uso de caché, aunque mantiene la misma complejidad algorítmica.