Pruebas sobre el comportamiento de la memoria caché: MULTIPLICACIÓN DE MATRICES

Patrick Xavier Márquez Choque

1. Introducción

Este trabajo tiene como objetivo la explicación de la implementación, resultados y análisis de la ejecución de la comparación de 2 algoritmos de multiplicación de matrices, la multiplicación de matrices que utiliza 3 bucles FOR's anidados en comparación al algoritmo de la multiplicación de matrices por bloques que utiliza 6 bucles anidados FOR's analizando el movimientos de los datos entre la memoria principal y la memoria caché evaluando su complejidad algoritmica utilizando diferentes tamaños de matrices ejecutando los programas en conjunto con las herramientas valgrind y kcachegrind para obtener una evaluación más precisa de su desempeño en término de cache misses.

2. Implementación

El link al repositorio donde se encuentra implementado los códigos mencionados se encuentra en el siguiente enlace: `https://github.com/patrick03524/Programacion-Paralela-y-Distribuida/tree/main/SegundoTrabajo`

LA implementación de ambos métodos de la multiplicación de matrices necesita de funciones y librerías de la siguiente manera:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <chrono>
#include <iomanip>
using namespace std;
using namespace std::chrono;

#define matrizA 1000
#define matrizB 1000
#define matrizC 1000
double **A, **B, **C;

void initialize();
void print_();

void initialize(){
  /*  Initialize A  */
```


doi:10.1017/xxxxxxx xxxx

```
19    A = new double*[matrizA];
20    for(int i = 0; i<matrizA; ++i){
21      A[i] = new double[matrizB];
22    }
23    int numero_aleatorio;
24    for(int i = 0; i<matrizA; ++i){
25      for(int j = 0; j<matrizB; ++j){
26        numero_aleatorio = 1 + rand() % (101 - 1);  ///numeros entre
      1-100
27        A[i][j] = numero_aleatorio;
28      }
29    }
30    /*  Initialize B  */
31    B = new double*[matrizB];
32    for(int i = 0; i<matrizB; ++i){
33      B[i] = new double[matrizC];
34    }
35    for(int i = 0; i<matrizB; ++i){
36      for(int j =0; j<matrizC; ++j){
37        numero_aleatorio = 1 + rand() % (101 - 1);  ///numeros entre
      1-100
38        B[i][j] = numero_aleatorio;
39      }
40    }
41    /*  Initialize C  */
42    C = new double*[matrizA];
43    for(int i = 0; i<matrizA; ++i){
44      C[i] = new double[matrizC];
45    }
46  }
47
48  void print_(){
49    cout<<"MATRIZ A"<<endl;
50    for(int i = 0; i<matrizA; ++i){
51      for(int j = 0; j<matrizB; ++j){
52        cout<<A[i][j]<<" ";
53      }
54      cout<<endl;
55    }
56    cout<<"ARRAY B"<<endl;
57    for(int i = 0; i<matrizB; ++i){
58      for(int j = 0; j<matrizC; ++j){
59        cout<<B[i][j]<<" ";
60      }
61      cout<<endl;
62    }
63    cout<<"ARRAY RES"<<endl;
64    for(int i = 0; i<matrizA; ++i){
65      for(int j = 0; j<matrizC; ++j){
66        cout<<C[i][j]<<" ";
67      }
68      cout<<endl;
69    }   }
```

Listing 1: Métodos y funciones necesarias para la inicialización

Para la implementación de ambos métodos de multiplicación de matrices es el siguiente:

```
    /*  First pair of loops */

  for(int i = 0; i<matrizA; ++i){
    for(int j = 0; j<matrizC; ++j){
      C[i][j] = 0;
      for(int k =0; k<matrizB; ++k){
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
  /*  Second pair of loops  */
  /* First Inicialize the number of every Block Size */
  blockSize = max(matrizA,matrizC);
  blockSize = min(matrizA,matrizC);
  blockSize = 10;

  for(int i = 0; i<matrizA; i+=blockSize){
    for(int j = 0; j<matrizC; j+=blockSize){
      for(int k =0; k<matrizB; k+=blockSize){
        for(int ii = i; ii<min(i+blockSize,matrizA); ++ii){
          for(int jj = j; jj<min(j+blockSize,matrizC); ++jj){
            for(int kk = k; kk<min(k+blockSize,matrizB); ++kk){
              C[ii][jj] += A[ii][kk] * B[kk][jj];
            }
          }
        }
      }
    }
  }
```

Listing 2: Algoritmo de la multiplicación de matrices

La implementación del anterior código fue realizado en C++ según la multiplicación de matrices clásica o la versión por bloques.

La **complejidad algoritmica** de ambas métodos de multiplicaciones de matrices se explica de la siguiente manera:

- a: el tamaño del filas de la primera matriz.
- b: tanto el tamaño de columnas de la primera matriz como el tamaño de filas de la segunda matriz según la propiedad de multiplicación de matrices.
- c: el tamaño de columnas de la segunda matriz.
- n: tamaño de cada bloque que se separa dentro de la multiplicación de matrices por bloques.

*La **complejidad algoritmica** de la multiplicación de matrices clásica es de $O(a * b * c)$ en general y $O(a^3)$ en caso de multiplicar 2 matrices cuadradas.*

En cambio *La **complejidad algoritmica** de la multiplicación de matrices por bloques es de*

$$O(\frac{a*b*c}{(\frac{n}{min(a,b,c)})}) \tag{1}$$

en general y en caso de multiplicar 2 matrices cuadradas será:

$$O(\frac{a^3}{(\frac{n}{a})^2}) \tag{2}$$

Para la experimentación y obtención de resultados se realizaron una cantidad considerable de elementos hasta multiplicar dos matrices de $1000x1000$ utilizando la instrucción new para reservar la memoria necesaria en el Heap(Esto se puede comprobar de manera mucho más detalladas en el apartado de Análisis de la Ejecución). Además se utilizó una librería llamada chrono para la evaluación de los tiempos que se analizaran en el apartado de resultados.

El link al repositorio donde se encuentra implementado los códigos mencionados se encuentra en el siguiente enlace: `https://github.com/patrick03524/Programacion-Paralela-y-Distribuida/tree/main/SegundoTrabajo`

3. Resultados

Matriz A		Matriz B		Matriz C		Tiempo Promedio	Resultados en segundos				
Multiplicación de Matrices clásica con 3 FOR's anidados											
100	100	100	100	100	100	0.0073005	0.003996	0.008003	0.007001	0.007005	0.008005
							0.008001	0.007997	0.008002	0.005	0.009995
100	1000	1000	500	100	500	0.6323344	0.548128	0.596736	0.622026	0.594999	0.768032
							0.604002	0.73143	0.665997	0.610996	0.580998
500	1000	1000	100	500	100	0.5413964	0.49899	0.513003	0.505996	0.514994	0.500996
							0.614999	0.594996	0.511995	0.528995	0.629
100	500	500	1000	100	1000	0.5038991	0.533002	0.501996	0.535998	0.356997	0.485001
							0.492996	0.497999	0.618998	0.526006	0.489998
1000	500	500	100	1000	100	0.6826933	0.66	0.695998	0.588	0.682998	0.680956
							0.720998	0.679994	0.755995	0.664996	0.696998
1000	1000	1000	1000	1000	1000	16.6825	16.348	16.036	18.269	16.657	15.807
							16.544	15.798	17.04	18.253	16.073
Multiplicación de Matrices por Bloques con 6 FOR's anidados											
100	100	100	100	100	100	0.0117994	0.011003	0.010999	0.012002	0.011	0.012001
							0.012991	0.011	0.015	0.012	0.009998
100	1000	1000	500	100	500	0.4548095	0.446001	0.470029	0.467995	0.45503	0.473999
							0.442026	0.455994	0.426029	0.444995	0.465997
500	1000	1000	100	500	100	0.4235902	0.423031	0.439031	0.429996	0.433999	0.438995
							0.365003	0.430862	0.430995	0.420996	0.422994
100	500	500	1000	100	1000	0.4326059	0.381999	0.436993	0.438026	0.432008	0.441008
							0.439998	0.451008	0.429993	0.436027	0.438999
1000	500	500	100	1000	100	0.4760487	0.449999	0.473454	0.490997	0.470996	0.41
							0.486993	0.493029	0.501025	0.486996	0.496998
1000	1000	1000	1000	1000	1000	7.86089	7.49303	8.14702	8.38507	7.603	7.76329
							7.69599	7.58907	8.03174	8.234	7.66669

Figura 1: Cuadro comparativo entre los resultados de la ejecución del programa

El hardware utilizado para las ejecuciones de los algoritmos es el siguiente:

- Intel I5 G4560 con 3.50GHz de velocidad promedio del procesador
- Una memoria RAM de 12.0GB
- Una caché de 3 niveles en primer lugar L1 con tamaño de 128kb, L2 con 512kb y L3 con 3.0MB

Se hicieron varias ejecuciones de ambos algoritmos de multiplicación de matrices y estas se pueden organizar en la Figura 1 realizando 10 ejecuciones de cada algoritmo utilizando diferentes tamaños (*Se realizaron 60 ejecuciones multiplicando 6 diferentes pares de matrices de tamaños:* $100x100 * 100x100$, $100x1000 * 1000x500$, $500x1000 * 1000x100$, $100x500 * 500x1000$, $1000x500 * 500x100 y 1000x1000 * 1000x1000$) de las matrices para obtener un promedio del tiempo de ejecución obtenido y asi obtener mejores conclusiones.

En el caso del rendimiento del algoritmo de multiplicación de matrices por bloques depende directamente de la cantidad del tamaño del bloque elegido para ejecutar el programa. Esta característica influye en nuestro rendimiento de diferentes maneras:

- Según la complejidad del algoritmo de multiplicación de matrices por bloques se puede llegar al **peor de los casos, donde se multipliquen matrices cuadradas y el tamaño de bloque será del mismo tamaño del número de filas y columnas de nuestras matrices, esto hace que la complejidad del algoritmo sea n^3 volviendolo similar al algoritmo de multiplicación de matrices clásica.**
- También dependiendo de que tan pequeño sea el tamaño del bloque entonces es donde la lógica ”Divide y Vencerás” se viene afectada ya que, de manera análoga de un computador y la caché que se utiliza, tiene que tener un equilibrio entre el tamaño total de la memoria principal en comparación de la memoria caché. Esto se puede entender ya que si se tiene un tamaño de bloque muy pequeño entonces la mejora en el tiempo en comparación sería minima al del primer algoritmo de multiplicación de matrices.

Entonces se puede concluir que mientras se utilicen matrices más grandes **el segundo algoritmo de Multiplicacion de Matrices por Bloques con 6 FOR’s anidados es mucho más rápido en cuestión de tiempos a comparación del primer algoritmo de Multiplicación de Matrices clásica con 3 FOR’s anidados**.

4. Análisis de la Ejecución

La interpretación para que el segundo algoritmo sea más rapido que el primero se debe a la característica de utilización de bloques dentro de las matrices, este detalles hace que a cantidades mucho más grandes de matrices esto permite divir la matriz en pequeñas cantidades siguiendo un poco la lógica de ”divide y vencerás” ya que

es mucho menos costoso multiplicar por partes matrices más pequeñas que hacerlo directamente con las matrices principales ya que son realmente grandes; esto también se puede concluir debido a la complejidad de cada uno es diferente, esto se puede apreciar en la siguiente Figura:

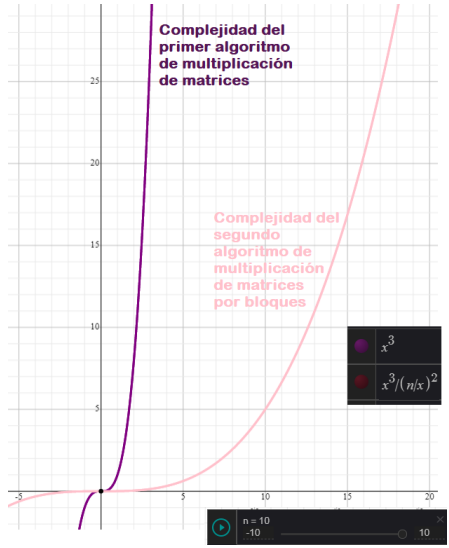

Figura 2: Funcion de Complejidad de cada algoritmo

Se puede observar entonces que el primer algoritmo de multiplicación de matrices es una función cubica a comparación del segundo algoritmo que puede disminuir su complejidad dependiendo del tamaño de bloques ”n” haciendo que en el peor caso donde el tamaño del bloque sea el mismo que las matrices y a la vez estas matrices sean cuadradas. Esto es importante tenerlo en cuenta ya que esta mejora del primer algoritmo es escencial para obtener mejores tiempos y es necesario entender bien para determinar una cantidad para el tamaño de los bloques que pueda mejorar el rendimiento en nuestro código.

4.1. Análisis utilizando las Herramientas Valgrind y KCachégrind

Pero para un mejor análisis de los movimientos de los datos entre la memoria principal y la memoria cache, la cantidad de cache misses y el ratio de los mismos es necesario utilizar 2 herramientas llamadas ”Valgrind” y ”Kcachegrind” para obtener una evaluación más precisa del desempeño del código.

En primer lugar con la herramienta ”Valgrind” es una herramienta de depuración de código que analizará estos algoritmos de una manera mucho más detallada y la herramienta ”KCacheGrind” permite analizar estos datos a nivel de la cache hasta de una manera gráfica de la siguiente manera:

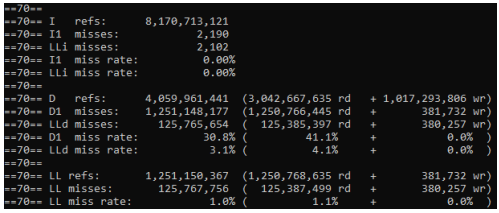

Figura 3: Resultado de la herramienta Valgrind del primer algoritmo.

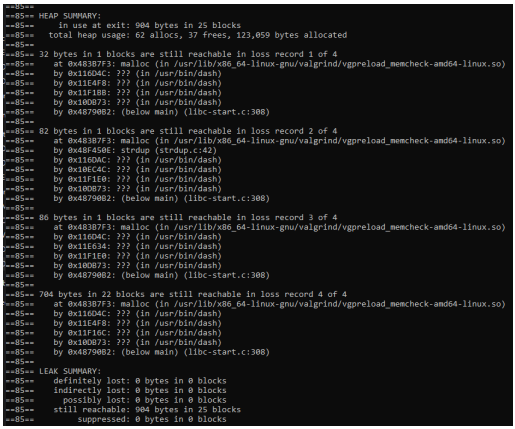

Figura 4: Resultados más precisos de la herramienta Cachegrind del primer algoritmo.

Podemos interpretar estos resultados de tal manera que **hubo alrededor de un 30.8 % de caché miss rate en la primera caché del ordenador L1, y un porcentaje de 1.0 % de caché miss rate hasta el último nivel de la caché**(Last Level Caché).

Este programa en especifico tiene una gran cantidad de caché misses relacionados al primer nivel de la caché en compración del ultimo nivel pero esto significaria un gasto de tiempo significante para la comparación de los tiempos de ejecución.

En cambio al utilizar estas herramientas en el segundo algoritmo los resultados son los siguientes:

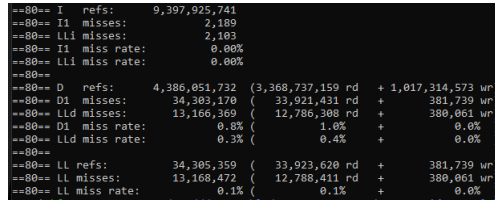

Figura 5: Resultado de la herramienta Valgrind del segundo algoritmo.

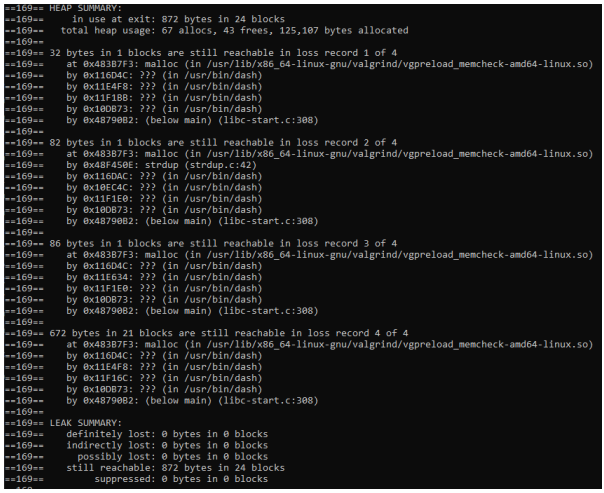

Figura 6: Resultados más precisos de la herramienta Cachegrind del segundo algoritmo.

Podemos interpretar estos resultados de tal manera que **hubo alrededor de un 0.8 % de caché miss rate, que es mucho más pequeño que el del primer algoritmo y también se entiende en los tiempos ya que al utilizar un tamaño de bloques pequeño que sea divisor del tamaño total de las matrices, por ejemplo un tamaño de 10, hace que las operaciones que se esten realizando puedan ser más eficientes y por lo tanto se pueda evitar la cantidad de cache misses dentro de los niveles de nuestra caché. También se tiene un porcentaje de 0.1 % de caché miss rate hasta el último nivel dela caché, este es mínimo ya que como no ocurrieron muchos cache misses dentro del primer nivel de la caché esto permitió que no sea necesario utilizar los otros niveles de la caché mucho más alla de lo necesario y por lo tanto obtener mejores resultados de la ejecución de los programas**.

Para hacer esta comparación mucho más exacta que se pueda interpretar estos cambios y movimientos de datos a nivel de la caché entonces se obtuvo la siguiente imagen de la herramienta para Windows ”QCachegrind”, que tiene cierta similitud con los resutlados obtenidos ya que se obtienen una interfaz para los datos de nuestros resultados.

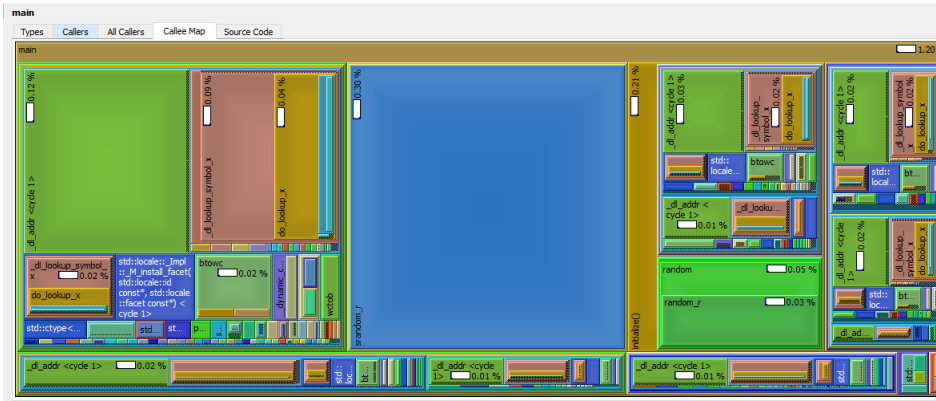

Figura 7: Resultados gráficos de la caché de la multiplicación de matrices

Este software nos permite observar la organización de nuestros algoritmos de la multiplicación de matrices y como se mueven y distribuyen los datos dentro de la memoria caché, podemos observar que todo se encierra dentro de la ejecución del main de nuestro programa. Además que, según los parametros del código, es como se realiza una gran cantidad de operaciones que se repiten y que se almacenan dentro de nuestra memoria caché. Por ejemplo vemos que el gran rectángulo azul tiene la instrucción para obtener números aleatorios, vemos que como tal no almacena el número pero en un porcentaje 0.3 % guarda la información con la que probablemente obtenga estos números aleatorios la librería ”stdlib.h” (tal vez alguna semmila de la cual se obtienen los pseudo-números aleatorios). También se puede apreciar bastantes cuadros con el nombre de ”_dl_addr” estas podrían representar datos que se repiten frecuentemente al hacer uso de la unidad lógica ALU ya que nuestro programa se basa fundamentalmente en operaciones matemáticas de multiplicación y suma de valores numéricos.