Universidad Católica
San Pablo

CIENCIA DE LA COMPUTACIÓN

Computación Paralela y Distribuida

Laboratorio - 2

Rodrigo Alonso Torres Sotomayor

CCOMP 8-1

"El alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo"

Multiplicación de Matrices

Introducción

En este trabajo se analizarán dos maneras de realizar la multiplicación de matrices: la multiplicación clásica y la multiplicación por bloques. Para este análisis se utilizó una máquina virtual utilizando Linux para poder compilar y ejecutar estos códigos y para la obtención de datos recopilados en dichas pruebas se usó valgrind y kcachegrind.

Requisitos de la PC

En este trabajo se utilizó una máquina virtual y las caracteristicas de dicha máquina son:

- Sistema Operativo: Linux.
- Memoria base (RAM): 4 GB
- Procesador: 4CPUs
- Memoria VRAM: 17mb

Implementaciones

```
// Funcion para multiplicar dos matrices por bloques
vector<vector<int>> multiplyMatricesByBlocks(vector<vector<int>>& A, vector<vector<int>>& B,
 ↪  int blockSize) {
    int n = A.size();
    int m = B[0].size();
    int p = A[0].size();

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

    for (int i = 0; i < n; i += blockSize) {
        for (int j = 0; j < m; j += blockSize) {
            for (int k = 0; k < p; k += blockSize) {
                for (int ii = i; ii < min(i + blockSize, n); ii++) {
                    for (int jj = j; jj < min(j + blockSize, m); jj++) {
                        for (int kk = k; kk < min(k + blockSize, p); kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }

    return C;
}

//funcion multplicacion de matrices clasica
vector<vector<int>> multiplicacionClasica(vector<vector<int>>& A, vector<vector<int>>& B)
{
    int n = A.size();
    int m = B[0].size();
    int p = A[0].size();
```

```
    vector<vector<int>> C(n, vector<int>(m, 0));

    for (int filas = 0; filas < n; filas++) {
        for (int columnasC = 0; columnasC < m; columnasC++) {
            for (int columnas = 0; columnas < p; columnas++) {
                C[filas][columnasC] += A[filas][columnas] * B[columnas][filas];
            }
        }
    }

    return C;

}
```

Resultados

0.1 Tiempo

	Clásica	2 Bloques	3 Bloques
100 x 100	0.0406762	0.0254851	0.0554851
1000 x 1000	20.7125	24.6906	23.6659

0.2 Valgrind y Kcachegrind

0.2.1 Trabajando con matrices 100 x 100

Se utilizaron matrices de 100 x 100 para esta prueba. Los resultados de la multplicación de matrices usando el método clásico se aprecia en las imágenes 1 y 2, y el método de bloques, en este caso se utilizó sólo 2 bloques en esta prueba, los resultados se pueden observar en las imágenes 3 y 4.

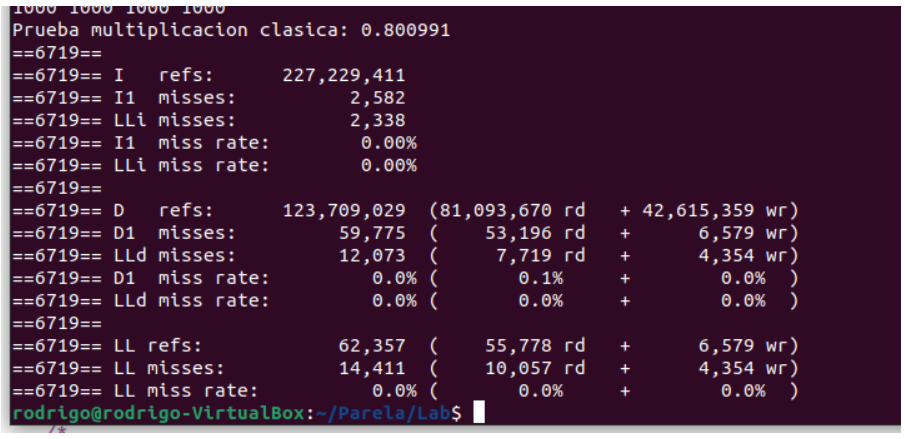

Figure 1: Test 100x100, Clásica, Valgrind

0.2.2 Trabajando con matrices 1000 x 1000

Se utilizaron matrices de 1000 x 1000 para esta prueba. Los resultados de la multplicación de matrices usando el método clásico se aprecia en las imágenes 5 y 6, y el método de bloques, en este caso se utilizaron 2 y 3 bloques en esta prueba, los resultados se pueden observar en las imágenes 7, 8, 9 y 10.

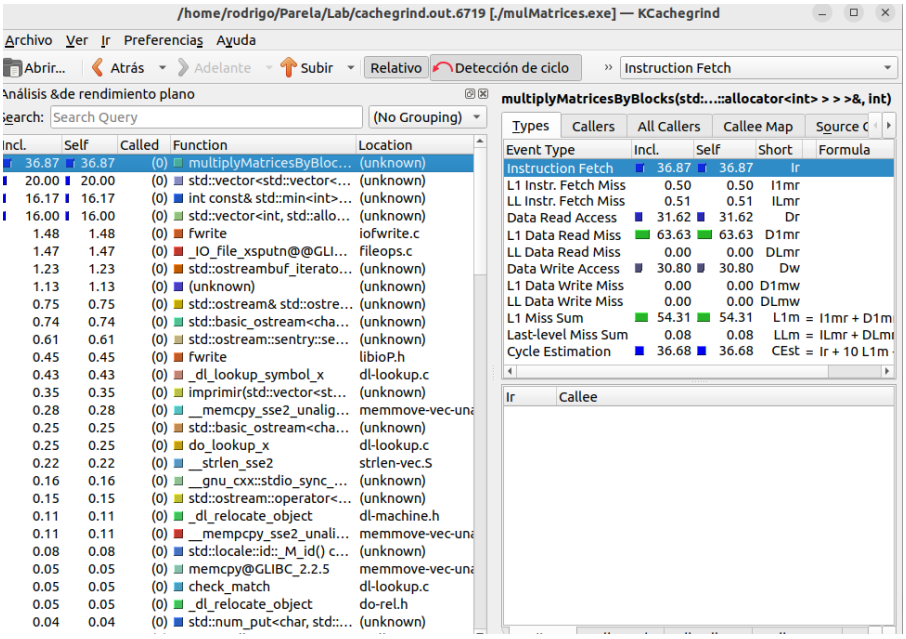

Figure 2: Test 100x100, Clásica, Kcachegrind

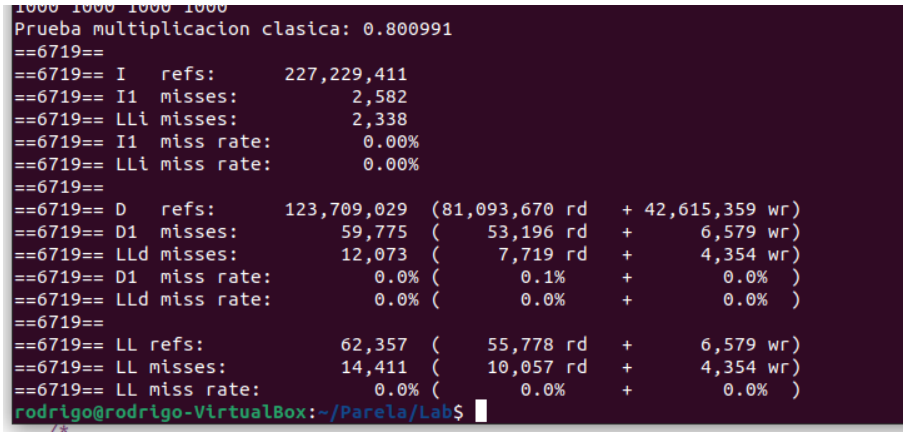

Figure 3: Test 100x100, 2 Bloques, Valgrind

0.2.3 Trabajando con matrices 10000 x 10000

Se utilizaron matrices de 10000 x 10000 para esta prueba. Los resultados de la multplicación de matrices usando el método clásico se aprecia en las imágenes 11 y 12, y el método de bloques, en este caso se utilizaron 2, 3 y 5 bloques en esta prueba al ser matrices de mayor dimensión, los resultados se pueden observar en las imágenes 13, 14, 15, 16, 17 y 18.

Análisis

0.3 Diferencias Algorítmicas

- Multiplicación de Matrices por Bloques:

  Divide las matrices en bloques más pequeños y realiza la multiplicación de matrices por bloques. Utiliza cuatro bucles anidados para iterar sobre los bloques y realizar las multiplicaciones de manera más localizada. El tamaño del bloque (blockSize) es un parámetro importante que puede afectar el rendimiento.

- Multiplicación de Matrices Clásica:

  Utiliza tres bucles anidados para realizar la multiplicación de matrices de manera clásica. No hay una división en bloques; trabaja directamente con los elementos individuales de las matrices.

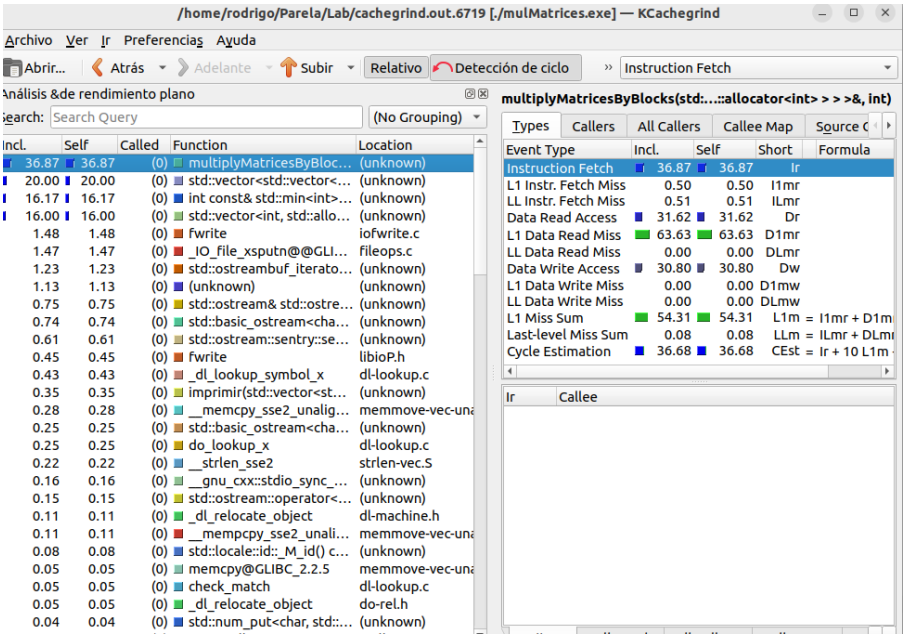

Figure 4: Test 100x100, 2 Bloques, Kcachegrind

```
==6810==    by 0x109A80: main (in /home/rodrigo/Parela/Lab/mulMatrices.exe)
==6810==
==6810== I   refs:      459,493,341
==6810== I1  misses:          2,010
==6810== LLi misses:          1,931
==6810== I1  miss rate:        0.00%
==6810== LLi miss rate:        0.00%
==6810==
==6810== D   refs:      252,981,826  (162,332,876 rd   + 90,648,950 wr)
==6810== D1  misses:      5,330,703  (  5,138,410 rd   +    192,293 wr)
==6810== LLd misses:        199,777  (      8,631 rd   +    191,146 wr)
==6810== D1  miss rate:         2.1% (        3.2%     +        0.2%  )
==6810== LLd miss rate:         0.1% (        0.0%     +        0.2%  )
==6810==
==6810== LL refs:         5,332,713  (  5,140,420 rd   +    192,293 wr)
==6810== LL misses:         201,708  (     10,562 rd   +    191,146 wr)
==6810== LL miss rate:          0.0% (        0.0%     +        0.2%  )

rodrigo@rodrigo-VirtualBox:~/Parela/Lab$
```

Figure 5: Test 1000x1000, Clásica, Valgrind

La multiplicación de matrices por bloques es generalmente más eficiente en términos de tiempo de ejecución, especialmente cuando se trata de matrices grandes. Esto se debe a que reduce la cantidad de acceso a memoria y explora más eficazmente la localidad de referencia espacial. Al dividir las matrices en bloques más pequeños, es más probable que los datos se mantengan en la caché, lo que reduce los fallos de caché y mejora el rendimiento en comparación con la multiplicación clásica de matrices.

0.4 Memoria Caché

- Multiplicación de Matrices por Bloques:

  La división en bloques puede ayudar a aprovechar mejor la memoria caché, ya que trabaja con submatrices más pequeñas que son más propensas a caber en la caché. Los bucles anidados en la multiplicación de bloques están diseñados para explorar la localidad de referencia espacial y minimizar los fallos de caché. La elección del tamaño de bloque (blockSize) es crucial para optimizar la eficiencia de la caché y puede requerir ajustes específicos según la arquitectura de la caché.

- Multiplicación de Matrices Clásica:

  La multiplicación clásica de matrices no tiene la ventaja de la división en bloques, por lo que los patrones de acceso a la memoria pueden no ser tan eficientes. Puede generar más fallos de caché, ya que accede a elementos individuales de manera secuencial.

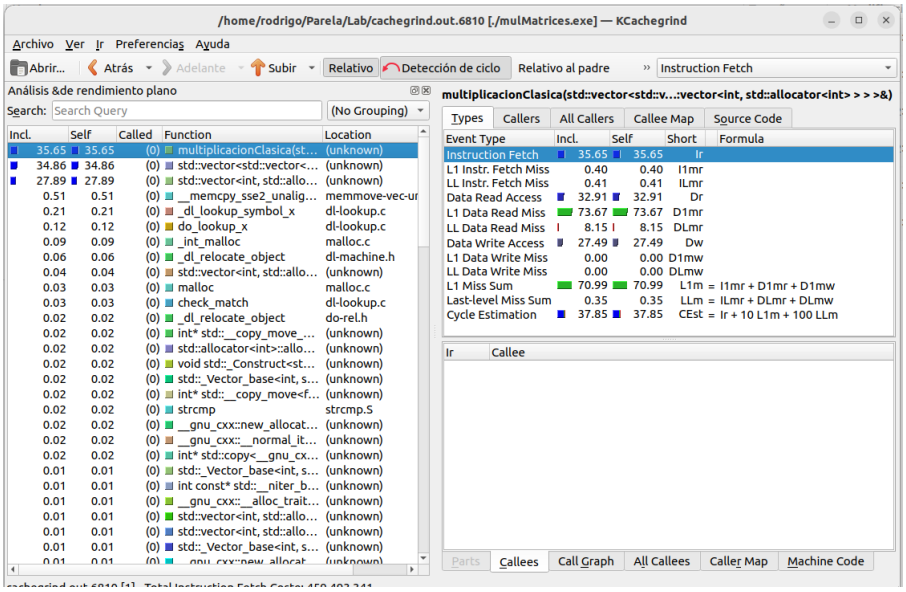

Figure 6: Test 1000x1000, Clásica, Kcachegrind

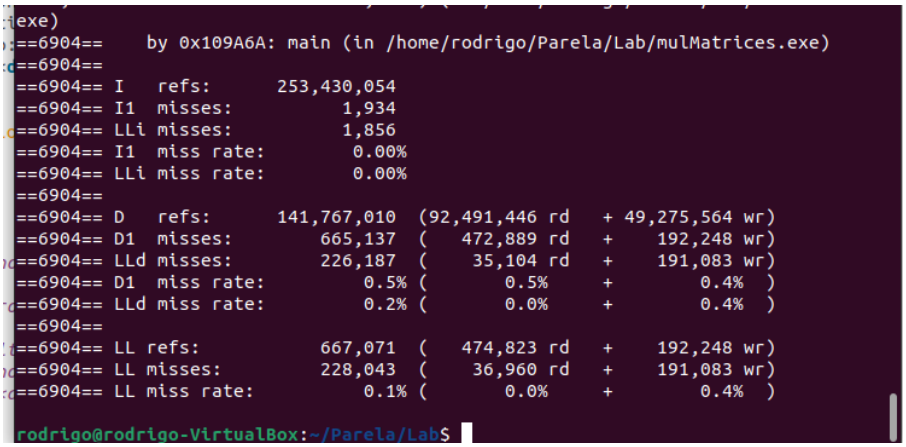

Figure 7: Test 1000x1000, 2 Bloques, Valgrind

En resumen, la multiplicación de matrices por bloques suele ser más eficiente en términos de rendimiento y uso de la memoria caché en comparación con la multiplicación de matrices clásica, especialmente cuando se trata de matrices grandes. Sin embargo, la elección del tamaño de bloque es importante y debe ajustarse en función de la arquitectura de hardware específica y el tamaño de las matrices que estés multiplicando.

Link del repositorio

https://github.com/RodATS/ComputacionParalela.git

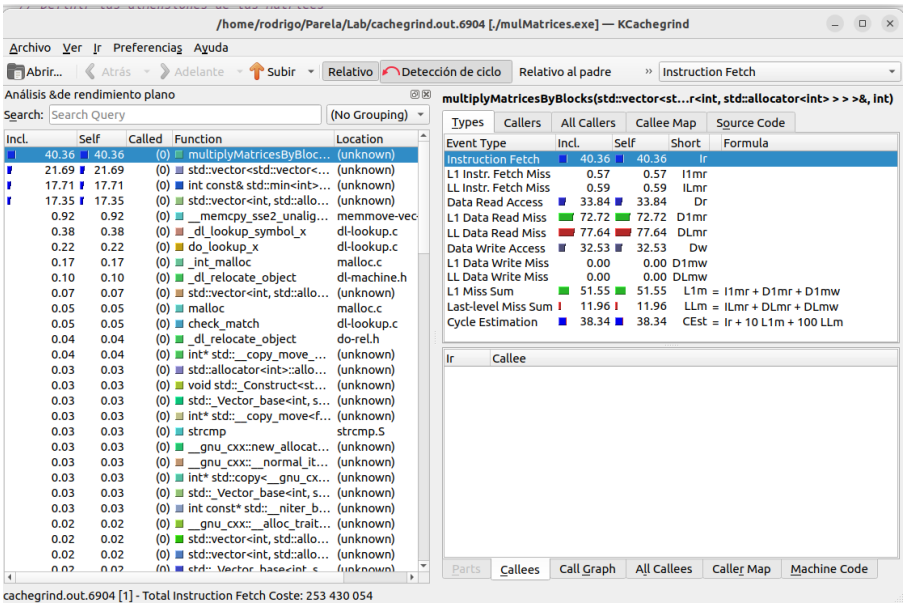

Figure 8: Test 1000x1000, 2 Bloques, Kcachegrind

```
==6958==    by 0x109A6A: main (in /home/rodrigo/Parela/Lab/mulMatrices.exe)
==6958==
==6958== I   refs:      310,113,542
==6958== I1  misses:          1,934
==6958== LLi misses:          1,856
==6958== I1  miss rate:        0.00%
==6958== LLi miss rate:        0.00%
==6958==
==6958== D   refs:      172,592,159  (111,889,098 rd   + 60,703,061 wr)
==6958== D1  misses:        522,427  (    330,179 rd   +    192,248 wr)
==6958== LLd misses:        223,406  (     32,323 rd   +    191,083 wr)
==6958== D1  miss rate:         0.3% (        0.3%     +        0.3%   )
==6958== LLd miss rate:         0.1% (        0.0%     +        0.3%   )
==6958==
==6958== LL refs:           524,361  (    332,113 rd   +    192,248 wr)
==6958== LL misses:         225,262  (     34,179 rd   +    191,083 wr)
==6958== LL miss rate:          0.0% (        0.0%     +        0.3%   )

rodrigo@rodrigo-VirtualBox:~/Parela/Lab$
```

Figure 9: Test 1000x1000, 3 Bloques, Valgrind

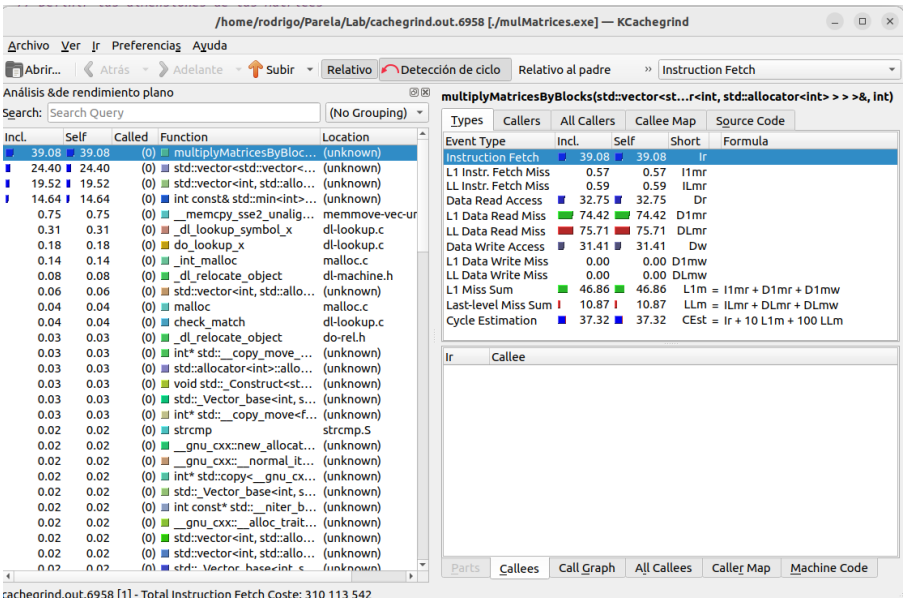

Figure 10: Test 1000x1000, 3 Bloques, Kcachegrind

```
 /home/rodrigo/Parela/Lab/mulMatrices.exe)
==7152==
==7152== I   refs:      110,193,803
==7152== I1  misses:          1,910
==7152== LLi misses:          1,833
==7152== I1  miss rate:        0.00%
==7152== LLi miss rate:        0.00%
==7152==
==7152== D   refs:       70,772,131  (35,872,692 rd   + 34,899,439 wr)
==7152== D1  misses:     16,269,803  ( 8,305,815 rd   +  7,963,988 wr)
==7152== LLd misses:      7,921,303  (     7,706 rd   +  7,913,597 wr)
==7152== D1  miss rate:        23.0% (      23.2%     +       22.8%  )
==7152== LLd miss rate:        11.2% (       0.0%     +       22.7%  )
==7152==
==7152== LL refs:        16,271,713  ( 8,307,725 rd   +  7,963,988 wr)
==7152== LL misses:       7,923,136  (     9,539 rd   +  7,913,597 wr)
==7152== LL miss rate:          4.4% (       0.0%     +       22.7%  )

rodrigo@rodrigo-VirtualBox:~/Parela/Lab$
```

Figure 11: Test 10000x10000, Clásica, Valgrind

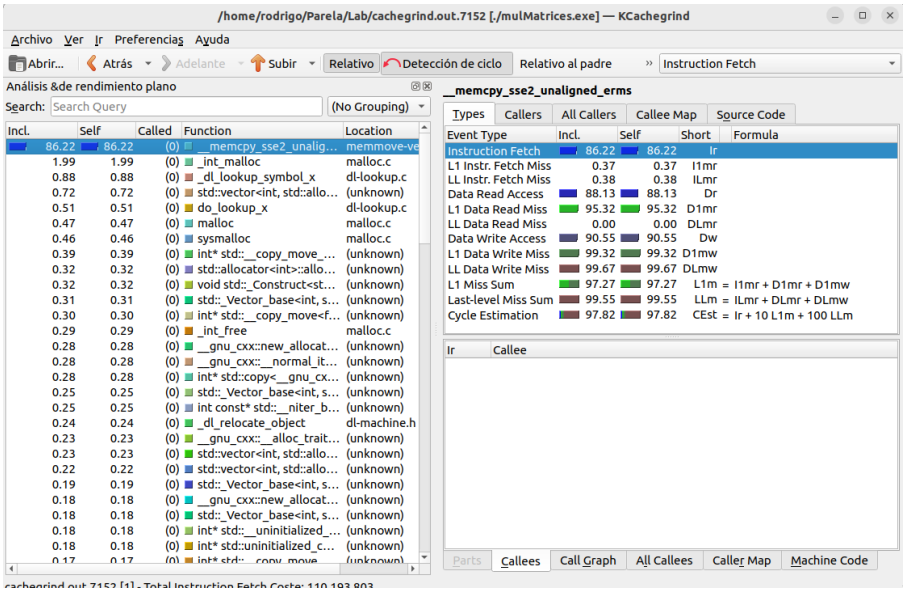

Figure 12: Test 10000x10000, Clásica, Kcachegrind

```
cator<int> >*, unsigned long, std::vector<int, std::allocator<int> > const&) (in
 /home/rodrigo/Parela/Lab/mulMatrices.exe)
==7203==
==7203== I   refs:      97,774,111
==7203== I1  misses:         1,913
==7203== LLi misses:         1,836
==7203== I1  miss rate:       0.00%
==7203== LLi miss rate:       0.00%
==7203==
==7203== D   refs:      62,715,711  (31,813,360 rd   + 30,902,351 wr)
==7203== D1  misses:    14,396,111  ( 7,349,495 rd   +  7,046,616 wr)
==7203== LLd misses:     7,009,573  (     7,706 rd   +  7,001,867 wr)
==7203== D1  miss rate:       23.0% (      23.1%     +       22.8%  )
==7203== LLd miss rate:       11.2% (       0.0%     +       22.7%  )
==7203==
==7203== LL refs:       14,398,024  ( 7,351,408 rd   +  7,046,616 wr)
==7203== LL misses:      7,011,409  (     9,542 rd   +  7,001,867 wr)
==7203== LL miss rate:         4.4% (       0.0%     +       22.7%  )

rodrigo@rodrigo-VirtualBox:~/Parela/Lab$
```

Figure 13: Test 10000x10000, 2 Bloques, Valgrind

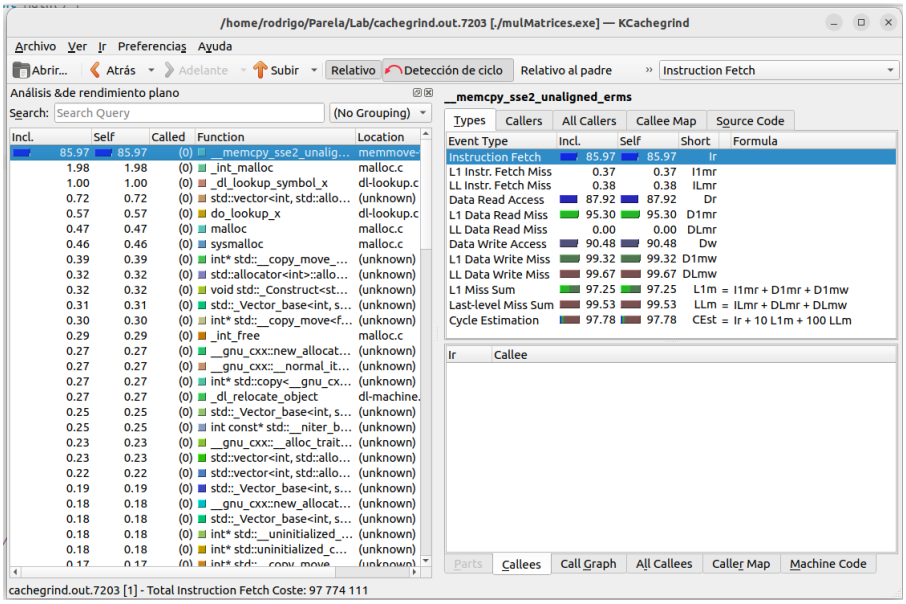

Figure 14: Test 10000x10000, 2 Bloques, Kcachegrind

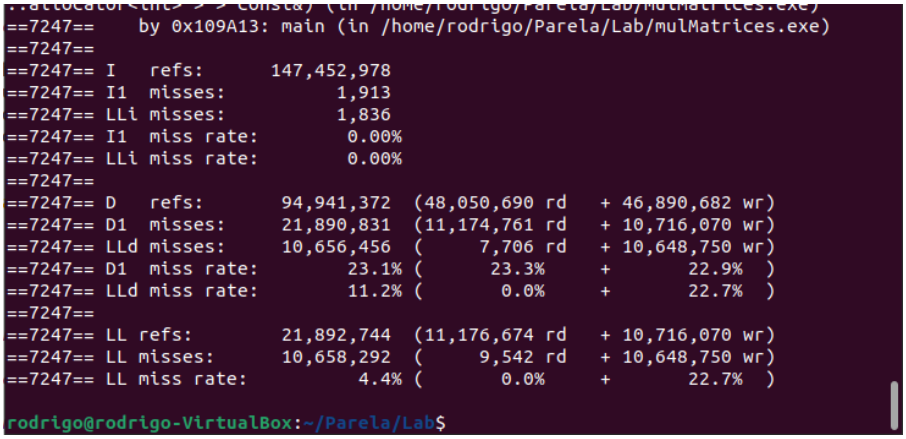

Figure 15: Test 10000x10000, 3 Bloques, Valgrind

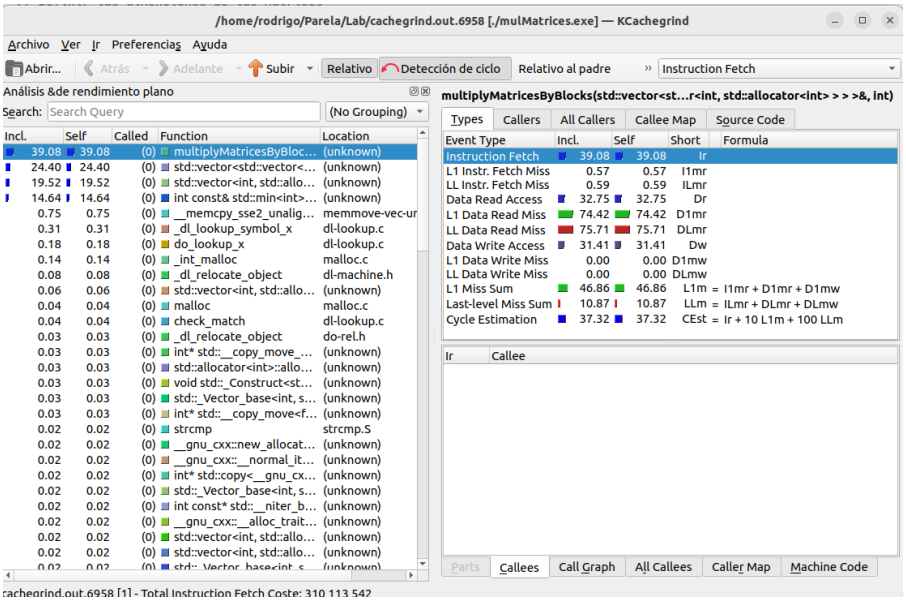

Figure 16: Test 10000x10000, 3 Bloques, Kcachegrind

```
cator<int> >*, unsigned long, std::vector<int, std::allocator<int> > const&) (in
 /home/rodrigo/Parela/Lab/mulMatrices.exe)
==7294==
==7294== I   refs:      87,838,259
==7294== I1  misses:         1,913
==7294== LLi misses:         1,836
==7294== I1  miss rate:       0.00%
==7294== LLi miss rate:       0.00%
==7294==
==7294== D   refs:      56,270,545  (28,565,874 rd   + 27,704,671 wr)
==7294== D1  misses:    12,897,167  ( 6,584,439 rd   +  6,312,728 wr)
==7294== LLd misses:     6,280,198  (     7,706 rd   +  6,272,492 wr)
==7294== D1  miss rate:       22.9% (      23.1%     +       22.8%  )
==7294== LLd miss rate:       11.2% (       0.0%     +       22.6%  )
==7294==
==7294== LL refs:       12,899,080  ( 6,586,352 rd   +  6,312,728 wr)
==7294== LL misses:      6,282,034  (     9,542 rd   +  6,272,492 wr)
==7294== LL miss rate:         4.4% (       0.0%     +       22.6%  )
```

Figure 17: Test 10000x10000, 5 Bloques, Valgrind

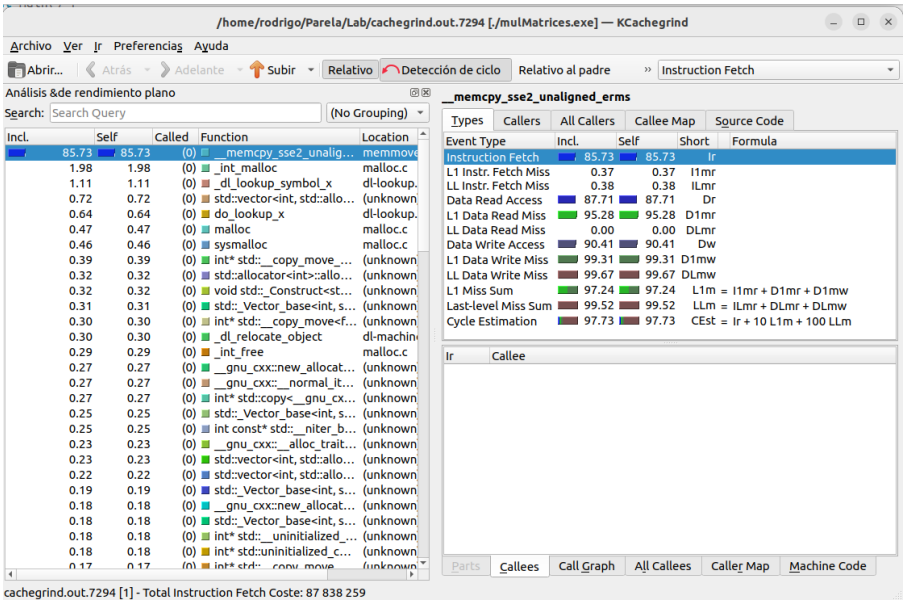

Figure 18: Test 10000x10000, 5 Bloques, Kcachegrind