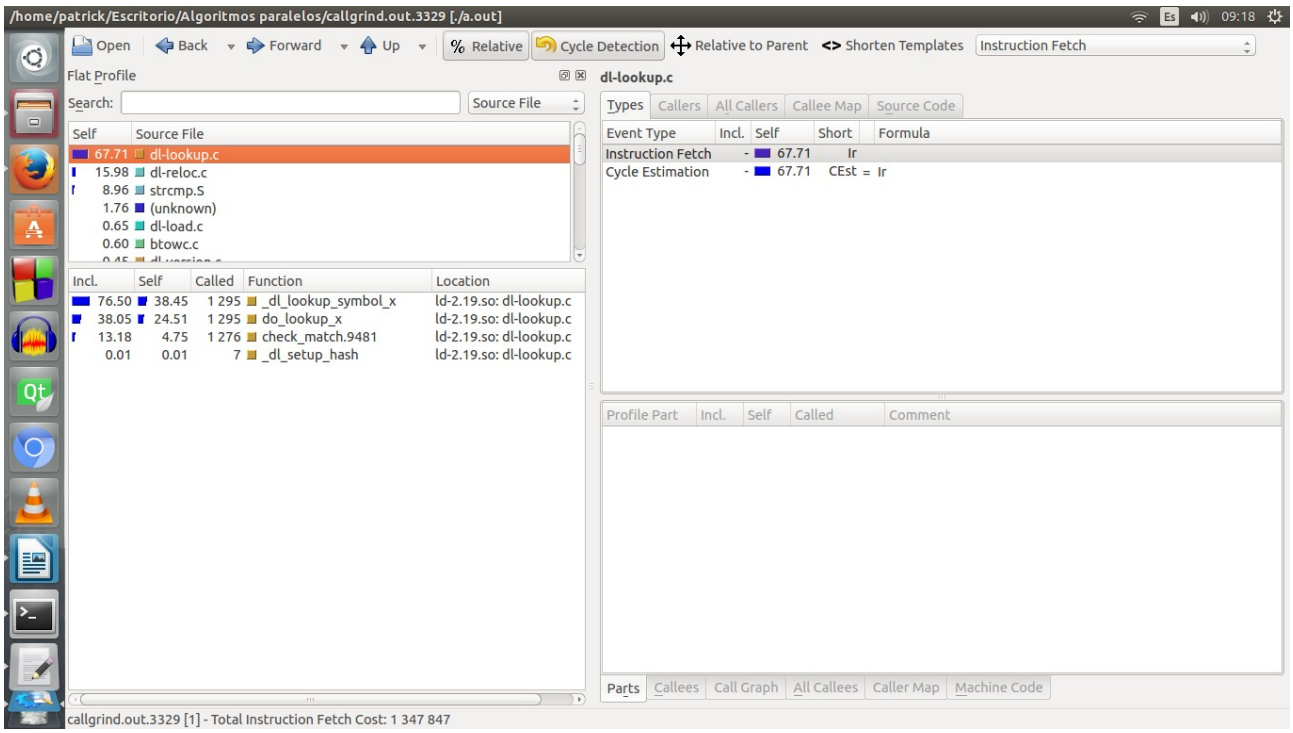
Informe Memoria Caché

Curso: Algoritmos Paralelos
Docente: Mg. Alvaro Mamani Aliaga
Apellidos y Nombres: Harvey Arce, Norman Patrick

- Implement in C the simple three-nested-loop version of the matrix product and try to evaluate its performance for a relatively large matrix size.
- Implement the blocked version with six nested loops to check whether you can observe a significant gain.
- Execute these algorithms step by step to get a good understanding of data movements between the cache and the memory and try to evaluate their respective complexity in term of distant memory access.
- Execute these two versions of the code with valgring and kcachegrind to get a precise evaluation of their performance in term of cache misses.

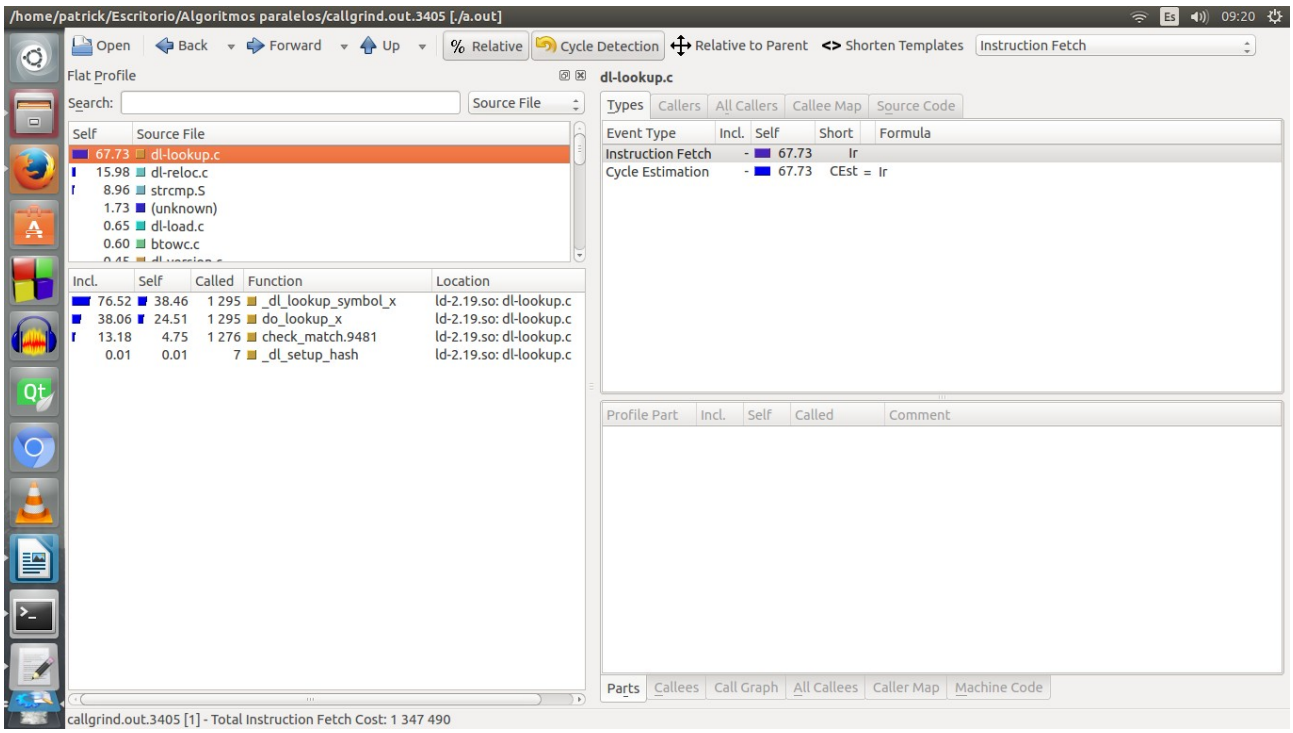
Para una matriz pequeña de 3 bucles

```
patrick@patrick-desktop:~/Escritorio/Algoritmos paralelos$ g++ 3-nested-loops.cpp
patrick@patrick-desktop:~/Escritorio/Algoritmos paralelos$ valgrind --tool=cachegrind ./a.out
==20200== I   refs:      1,347,200
==20200== I1  misses:        1,251
==20200== LLi misses:        1,224
==20200== I1  miss rate:      0.09%
==20200== LLi miss rate:      0.09%
==20200==
==20200== D   refs:        450,662  (329,215 rd   + 121,447 wr)
==20200== D1  misses:       10,351  (  8,770 rd   +   1,581 wr)
==20200== LLd misses:        6,405  (  5,211 rd   +   1,194 wr)
==20200== D1  miss rate:       2.2% (    2.6%     +     1.3%  )
==20200== LLd miss rate:       1.4% (    1.5%     +     0.9%  )
==20200==
==20200== LL refs:          11,602  ( 10,021 rd   +   1,581 wr)
==20200== LL misses:         7,629  (  6,435 rd   +   1,194 wr)
==20200== LL miss rate:        0.4% (    0.3%     +     0.9%  )
```



Para una matriz pequeña de 6 bucles

```
patrick@patrick-desktop:~/Escritorio/Algoritmos paralelos$ g++ 6-nested-loops.cpp
patrick@patrick-desktop:~/Escritorio/Algoritmos paralelos$ valgrind --tool=cachegrind ./a.out
==20955== I   refs:      1,346,843
==20955== I1  misses:        1,250
==20955== LLi misses:        1,223
==20955== I1  miss rate:      0.09%
==20955== LLi miss rate:      0.09%
==20955==
==20955== D   refs:        450,504  (329,075 rd   + 121,429 wr)
==20955== D1  misses:       10,351  (  8,770 rd   +   1,581 wr)
==20955== LLd misses:        6,405  (  5,211 rd   +   1,194 wr)
==20955== D1  miss rate:       2.2% (    2.6%     +     1.3%  )
==20955== LLd miss rate:       1.4% (    1.5%     +     0.9%  )
==20955==
==20955== LL refs:          11,601  ( 10,020 rd   +   1,581 wr)
==20955== LL misses:         7,628  (  6,434 rd   +   1,194 wr)
==20955== LL miss rate:        0.4% (    0.3%     +     0.9%  )
```



Para C = AB (A: $n \times m$, B: $m \times p$, entonces $C : n \times p$)

El algoritmo de 3 bucles toma Θ (nmp). Una simplificación común para el propósito del análisis de algoritmos es asumir que las entradas son todas las matrices cuadradas de tamaño n × n, en cuyo caso el tiempo de ejecución es Θ (n^3), es decir, cúbico.

Los tres bucles en la multiplicación de matrices pueden ser arbitrariamente intercambiados entre sí sin un efecto sobre la exactitud o tiempo de ejecución. Sin embargo, el orden puede tener un impacto considerable en el rendimiento práctico debido al acceso a la memoria y el uso de la memoria caché del algoritmo; cuál orden es mejor también depende de si las matrices se almacenan en orden por filas, en orden por columnas, o una mezcla de ambos. Lo que resultaría en mayor o menor uso de la memoria.

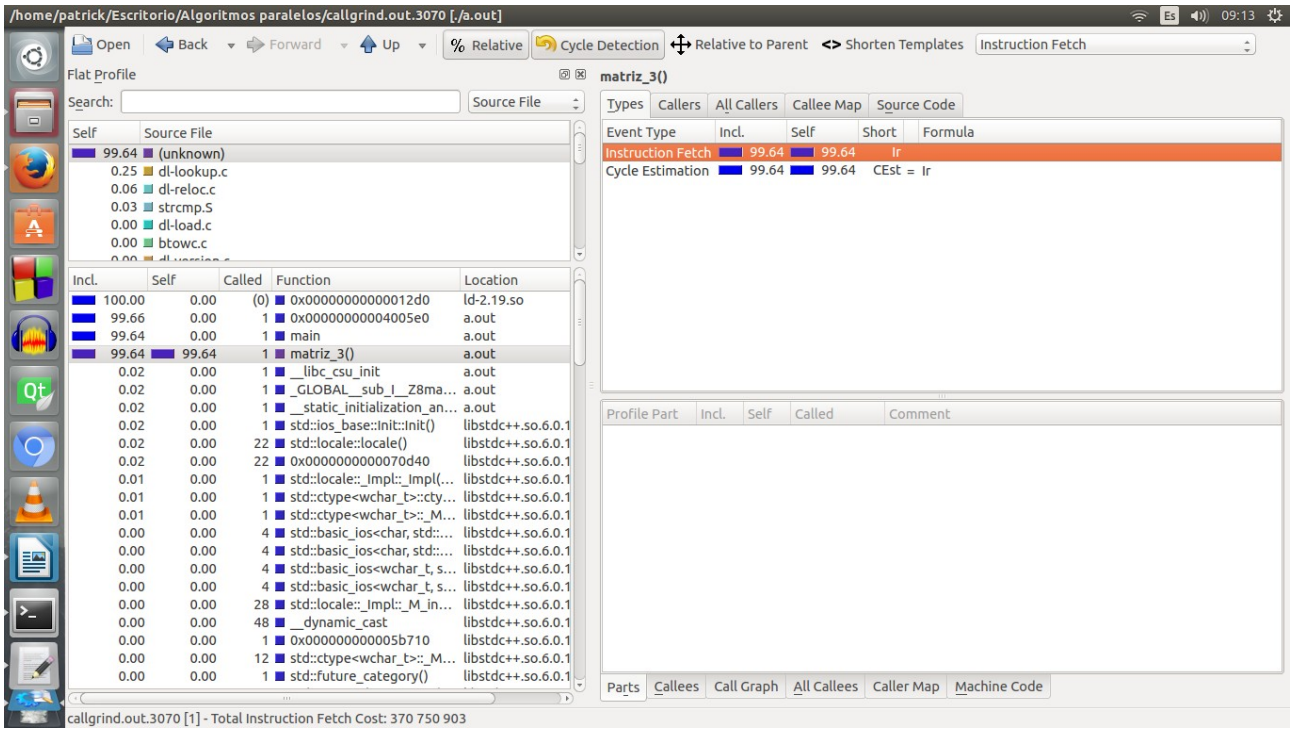
La variante óptima del algoritmo de 3 bucles para A y B es una versión de 6 bucles anidados, donde la matriz está implícitamente dividida en tamaños cuadrados de tamaño √m por √m (algoritmo de 6 bucles anidado).

La tasa de error de caché de la multiplicación de matrices (3 bucles) es la misma que la de una versión iterativa (6 bucles), pero a diferencia de ese algoritmo, el algoritmo recursivo es ajeno al cache, esto quiere decir que no hay parámetro de ajuste necesario para obtener un rendimiento óptimo de caché, y se comporta bien en un entorno de multiprogramación, donde los tamaños de caché son efectivamente dinámicos debido a otros procesos que ocupan espacio en el caché. El algoritmo de 3 bucles es ajeno a la caché también, pero mucho más lento en la práctica si el diseño de la matriz no está adaptado para el algoritmo.

Matriz grande (200x200) de 3 bucles (3 segundos)

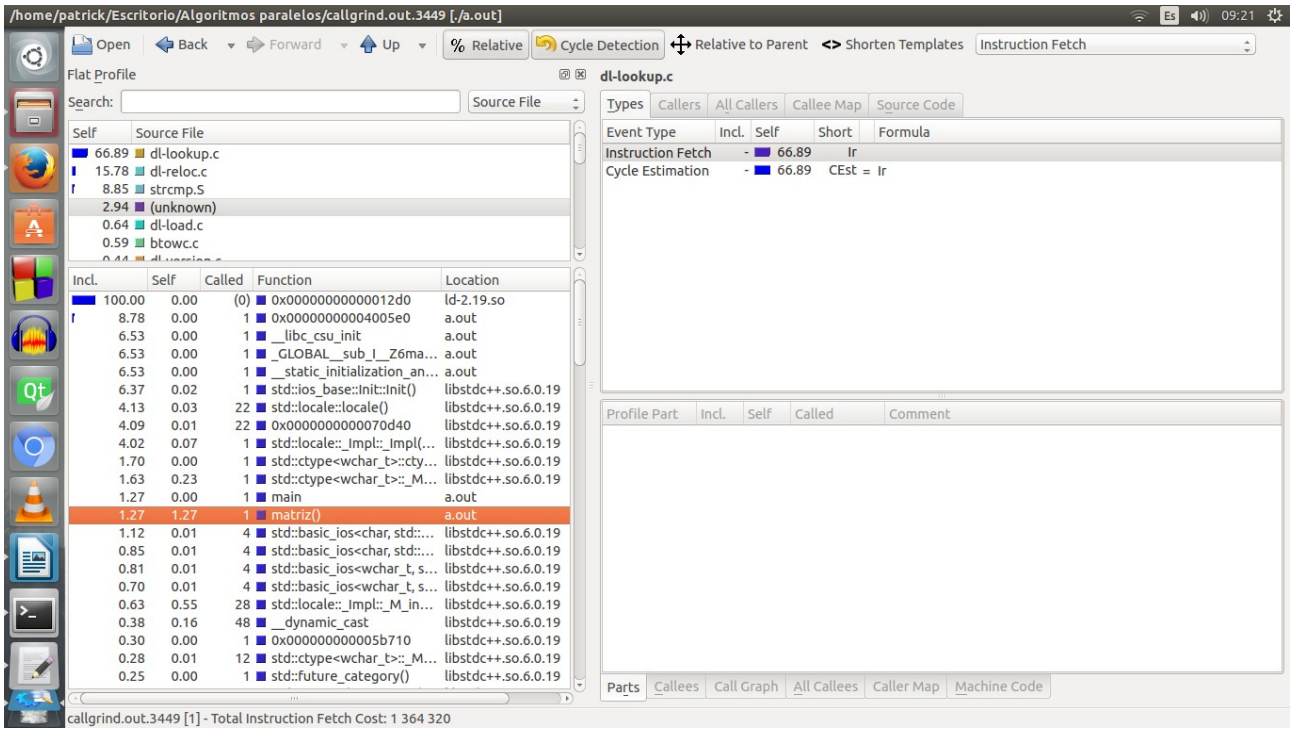
```
==13137== I   refs:      370,750,256
==13137== I1  misses:          1,251
==13137== LLi misses:          1,224
==13137== I1  miss rate:        0.00%
```

```
==13137== LLi miss rate:        0.00%
==13137==
==13137== D   refs:      153,212,810  (144,970,987 rd   + 8,241,823 wr)
==13137== D1  misses:        520,666  (    514,073 rd   +     6,593 wr)
==13137== LLd misses:         13,855  (      7,711 rd   +     6,144 wr)
==13137== D1  miss rate:         0.3% (        0.3%     +       0.0%  )
==13137== LLd miss rate:         0.0% (        0.0%     +       0.0%  )
==13137==
==13137== LL refs:           521,917  (    515,324 rd   +     6,593 wr)
==13137== LL misses:          15,079  (      8,935 rd   +     6,144 wr)
==13137== LL miss rate:          0.0% (        0.0%     +       0.0%  )
```



Matriz grande (200x200) de 6 bucles (1 segundo)

```
==11985==
==11985== I   refs:      27,461,571
==11985== I1  misses:         1,241
==11985== LLi misses:         1,212
==11985== I1  miss rate:       0.00%
==11985== LLi miss rate:       0.00%
==11985==
==11985== D   refs:      14,527,807  (14,395,877 rd   + 131,930 wr)
==11985== D1  misses:        10,560  (     8,978 rd   +   1,582 wr)
==11985== LLd misses:         6,605  (     5,411 rd   +   1,194 wr)
==11985== D1  miss rate:        0.0% (       0.0%     +     1.1%  )
==11985== LLd miss rate:        0.0% (       0.0%     +     0.9%  )
==11985==
==11985== LL refs:           11,801  (    10,219 rd   +   1,582 wr)
==11985== LL misses:          7,817  (     6,623 rd   +   1,194 wr)
==11985== LL miss rate:         0.0% (       0.0%     +     0.9%  )
```



Conclusiones

Se utilizaron las herramientas Valgrind y KcacheGrind que nos ayudan al momento de la depuración y la creación de perfiles mediante una interfaz.

Para matrices pequeñas la diferencia no es notable ya que no requiere de mucho esfuerzo computacional, sin embargo cada vez que van creciendo las matrices, el algoritmo de 3 bucles se vuelve cada vez más lento por que hace unso innecesario de memoria y por que su diseño no es óptimo. A diferencia del algoritmo de 6 bucles anidados que por su apariencia pareciera más lento es todo lo contrario.

Es notable la diferencia que existe entre las referencias del algoritmo de 3 bucles (370,750,256) y de 6 bucles (27,461,571) para matrices de tamaño considerable. Incluso el ratio de cache miss en el algoritmo de 6 bucles es 0% a diferencia del bucle de 3 algoritmos donde su porcentaje de cache miss es de 0.3%. Y el tiempo también es diferente en ambos casos: 3 segundos (3 bucles) y 1 segundo (6 bucles).

Para este tipo de problemas pareciera que un programa de 6 bucles demoraría más que otro programa de 3 bucles, pero al momento de analizar el comportamiento de la caché en memoria vemos que los 3 bucles son más ineficientes e incluso demoran más tiempo que los otros 6.

Comandos Utilizados

valgrind --tool=cachegrind ./a.out

valgrind --tool=callgrind --dump-instr=yes --collect-jumps=yes ./a.out

valgrind --tool=memcheck --leak-check=yes ./a.out

G_SLICE=always-malloc G_DEBUG=gc-friendly valgrind -v --tool=memcheck --leak-check=full --num-callers=40 --log-file=valgrind.log ./a.out

Código Utilizado

En las siguientes páginas está el código utilizado para analizar el comportamiento de la memoria, para ambos códigos simplemente se cambió el tamaño de las matrices.

- Para 3 bucles:
  Variable : Tam
- Para 6 bucles:
  Variable : Tam y Lim

Código Matriz 3 bucles

```
#include <iostream>

using namespace std;

void matriz_3(){

int tam=2;
int a[tam][tam],b[tam][tam],c[tam][tam];

for(int i=0;i<tam;i++)
for(int j=0;j<tam;j++)
   {
    a[i][j]=i;
    b[i][j]=i;
   }

for (int i = 0; i < tam; i++)
      for (int k = 0; k < tam; k++)
          for (int j = 0; j < tam; j++)
              c[i][j] = c[i][j] + a[i][k]*b[k][j];

}


int main()
{
matriz_3();
return 0;
}
```

Código Matriz 6 Bucles Anidados

```
#include <iostream>

using namespace std;

void matriz()
{
int tam=200;
int lim=200;
int a[tam][tam],b[tam][tam],c[tam][tam];

for(int j=0;j<tam;j+= lim)
  {
      for(int k=0; k<tam; k+= lim)
          {
            for(int i=0; i<tam; i++)
               {
                  for(int j = j; j<((j+lim)>tam?tam:(j+lim)); j++)
                       {
                         int temp = 0;
                         for(int k = k; k<((k+lim)>tam?tam:(k+lim)); k++)
                              {
                                temp += a[i][k]*b[k][j];
                              }
                         c[i][j] += temp;
                       }
               }
          }
  }
}


int main()
{
matriz();
return 0;
}
```