Pruebas sobre el comportamiento de la memoria caché: Multiplicación de matrices

Correa Ochoa Wilber *Departamento de Ciencia de la Computación*
Universidad Católica San Pablo
Arequipa, Perú
wilber.correa@ucsp.edu.pe

March 30, 2021

1 Implementación

Primero se generó números aleatorios para las 2 matrices:

```
//random
        static std::uniform_real_distribution <double>
            sample_01_(10000.0, 50000.0);
        std::random_device rd;
        std::mt19937 rng(rd());

    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            matA[i][j] = sample_01_(rng);
        }
    }
        for (int i = 0; i < n; i++){
                for (int j = 0; j < p; j++)
                {
                        matB[i][j] = sample_01_(rng);
                }
        }
```

1.1 Algoritmo 1: Multiplicación 3 bucles:

Esta es la multiplicación de matrices más común:

```
void normal_mult(int **matA, int **matB, int **matC)
{
```

```
        for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                        matC[i][j] = 0.0;
        for (int i = 0; i < m; ++i)
                for (int j = 0; j < p; ++j)
                        for (int k = 0; k < n; ++k)
        {
                                matC[i][j] += matA[i][k] * matB[k][j];
        }
}
```

Este algoritmo tiene una complejidad de n^3 por la cantidad de bucles.

1.2 Algoritmo 2: Multiplicación 6 bucles:

Esta multiplicación va haciéndolo por bloques contiguos, aprovechando así el área local de los datos, así logra menos cache misses que la multiplicación común. La variable blocksize es el tamaño en el que se van a dividir los bloques para agrupar los datos e irlos calculando.

```
void matrix_multiplication(int** matrix_A, int** matrix_B, int** matrix_C){
  for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
                matrix_C[i][j] = 0.0;

 for(int i=0; i<n; i+=block_size){
  for(int j=0; j<n; j+=block_size){
    for(int k=0; k<n; k+=block_size){
      for(int ii=i; ii<i+block_size; ii++){
        for(int jj=j; jj<j+block_size; jj++){
                  for(int kk=k; kk<k+block_size; kk++){
                        matrix_C[ii][jj] += matrix_A[ii][kk]*matrix_B[kk][jj];
                }
        }
     }
    }
  }
 }
}
```

Este algoritmo tiene una complejidad de n^6 por la cantidad de bucles.

2 Resultados

Se ha probado los algoritmos con matrices de 100×100, 200×200 y 600×600. La que dio resultados un poco más notorios fue de 600×600. Los resultados arrojados por valgrind para el algoritmo de 3 bucles se muestran:

```
wilber@DESKTOP-6KKAPV6:/mnt/d/paralelo/multi$ valgrind --tool=callgrind --simulate-cache=yes ./normal
==4792== Callgrind, a call-graph generating cache profiler
==4792== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==4792== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4792== Command: ./normal
==4792==
--4792-- warning: L3 cache found, using its data for the LL simulation.
==4792== For interactive control, run 'callgrind_control -h'.
==4792== error calling PR_SET_PTRACER, vgdb might block


==4792==
==4792== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==4792== Collected : 11170725079 4593531273 465407458 1917 256657466 95437 1822 8577 69955
==4792==
==4792== I   refs:      11,170,725,079
==4792== I1  misses:             1,917
==4792== LLi misses:             1,822
==4792== I1  miss rate:           0.00%
==4792== LLi miss rate:           0.00%
==4792==
==4792== D   refs:       5,058,938,731  (4,593,531,273 rd + 465,407,458 wr)
==4792== D1  misses:       256,752,903  (  256,657,466 rd +      95,437 wr)
==4792== LLd misses:            78,532  (        8,577 rd +      69,955 wr)
==4792== D1  miss rate:            5.1% (          5.6%   +         0.0%  )
==4792== LLd miss rate:            0.0% (          0.0%   +         0.0%  )
==4792==
==4792== LL refs:          256,754,820  (  256,659,383 rd +      95,437 wr)
==4792== LL misses:             80,354  (       10,399 rd +      69,955 wr)
==4792== LL miss rate:             0.0% (          0.0%   +         0.0%  )
```

Figure 1: Resultado de la ejecución del algoritmo 1 usando Valgrind.

Los resultados arrojados por valgrind para el algoritmo de 6 bucles se muestran:

```
wilber@DESKTOP-6KKAPV6:/mnt/d/paralelo/multi$ valgrind --tool=callgrind --simulate-cache=yes ./anormal
==4793== Callgrind, a call-graph generating cache profiler
==4793== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==4793== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4793== Command: ./anormal
==4793==
--4793-- warning: L3 cache found, using its data for the LL simulation.
==4793== For interactive control, run 'callgrind_control -h'.
==4793== error calling PR_SET_PTRACER, vgdb might block
==4793==
==4793== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==4793== Collected : 11758593891 4887660929 487420988 1837 1865112 95400 1739 8302 69917
==4793==
==4793== I   refs:      11,758,593,891
==4793== I1  misses:             1,837
==4793== LLi misses:             1,739
==4793== I1  miss rate:           0.00%
==4793== LLi miss rate:           0.00%
==4793==
==4793== D   refs:       5,375,081,917  (4,887,660,929 rd + 487,420,988 wr)
==4793== D1  misses:         1,960,512  (    1,865,112 rd +      95,400 wr)
==4793== LLd misses:            78,219  (        8,302 rd +      69,917 wr)
==4793== D1  miss rate:            0.0% (          0.0%   +         0.0%  )
==4793== LLd miss rate:            0.0% (          0.0%   +         0.0%  )
==4793==
==4793== LL refs:            1,962,349  (    1,866,949 rd +      95,400 wr)
==4793== LL misses:             79,958  (       10,041 rd +      69,917 wr)
==4793== LL miss rate:             0.0% (          0.0%   +         0.0%  )
```

Figure 2: Resultado de la ejecución del algoritmo 2 usando Valgrind.

3 Análisis de Resultados

El tiempo de ejecución del algoritmo 1 fue menor al del algoritmo 2, pero no por tanto, esto se debe a la complejidad de los algoritmos.

Pero al ejecutarse usando la herramienta valgrind se ve la diferencia entre cache misses de la data de la memoria cache L1, el algoritmo 1 tiene 5.1% de miss rate, mientras que el algoritmo 2 tiene 0.0%. Lo que hace que la complejidad del algoritmo no deje que haya una gran diferencia en los tiempos de ejecución es la cantidad de cache misses, ya que el cache miss, hace que tome mayor tiempo al que debería si todo la data accedida estuviese contigua en los bucles.

También podemos observar que la memoria L3 no tiene miss rate, esto se debe a que en las memorias L1, L2 o L3 se ha encontrado la data, por lo tanto no tiene que seguir buscando. No aparece información sobre la memoria L2, así que no se puede analizar muy bien.

4 Repositorio

Se tiene dos archivos, normal.cpp es el algoritmo 1; anormal.cpp es el algoritmo 2. `https://github.com/wilberever100/Paralelo/tree/main/multi`