

Algoritmos Paralelos / Laboratorio

Joseline Maria Zevallos Aliaga
joseline.zevallos@ucsp.edu.pe
Universidad Católica San Pablo

1 Multiplicación de Matrices

- Implementando la multiplicación de matrices con 3 bucles se hizo las pruebas con matrices de 200×200 , 400×400 , 600×600 , 800×800 y 1000×1000 , como se muestra en la Figura 1.

```
Matriz de 200x200
tiempo: 0.033175

real    0m0.039s
user    0m0.035s
sys      0m0.002s
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ g++ MatMul3b.cpp -o 3b
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ time ./3b
Matriz de 400x400

tiempo: 0.425989

real    0m0.495s
user    0m0.428s
sys      0m0.006s
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ g++ MatMul3b.cpp -o 3b
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ time ./3b
Matriz de 600x600

tiempo: 1.4363

real    0m1.486s
user    0m1.439s
sys      0m0.009s
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ g++ MatMul3b.cpp -o 3b
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ time ./3b
Matriz de 800x800

tiempo: 4.60301

real    0m4.645s
user    0m4.596s
sys      0m0.024s
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ g++ MatMul3b.cpp -o 3b
MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ time ./3b
Matriz de 1000x1000

tiempo: 22.14

real    0m22.258s
user    0m22.047s
sys      0m0.115s
```

Figure 1:

- Implementando la multiplicación de matrices con 6 bucles se hizo las pruebas con matrices de 200×200 , 400×400 , 600×600 , 800×800 y 1000×1000 como se muestra en la Figura 2.

Cuando se hace la prueba de la multiplicación de matrices de 1000×1000 se ve que la multiplicación de 6 bucles toma un tiempo de 13.4071 a comparación de la multiplicación de 3 bucles que toma un tiempo mayor a ese que es de 22.14.

En la multiplicación de 3 bucles debido a la localidad espacial de la cache ocurren muchos cache misses, debido a que se acceden a partes de memoria que la cache no tiene almacenada. En cambio la multiplicación de 6 bucles que se hace por bloques, al hacer esta separación por bloques se va en un orden mas favorable para la cache, produciendo menos cache misses.

```

Matriz de 200x200

tiempo: 0.069378

real    0m0.079s
user    0m0.071s
sys      0m0.003s
|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ g++ MatMul6b.cpp -o 6b
|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ time ./6b
Matriz de 400x400

tiempo: 0.584316

real    0m0.597s
user    0m0.585s
sys      0m0.007s
|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ g++ MatMul6b.cpp -o 6b
|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ time ./6b
Matriz de 600x600

tiempo: 1.96061

real    0m1.993s
user    0m1.955s
sys      0m0.017s
|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ g++ MatMul6b.cpp -o 6b
|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ time ./6b
Matriz de 800x800

tiempo: 4.697

real    0m4.794s
user    0m4.678s
sys      0m0.037s
|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ g++ MatMul6b.cpp -o 6b
|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ time ./6b
Matriz de 1000x1000

tiempo: 9.31696

real    0m9.512s
user    0m9.229s
sys      0m0.110s

```

Figure 2:

2 Usando Valgrind y Cachegrind

Probando con Valgrind con la opción de Cachegrind se obtuvo lo siguiente:

- Con la multiplicación de matrices de 3 bucles, como se muestra en la Figura3.:

```

|MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ valgrind --tool=cachegrind ./3b
==10940== Cachegrind, a cache and branch-prediction profiler
==10940== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==10940== Using Valgrind-3.14.0.SVN and LibVEX; rerun with -h for copyright info
==10940== Command: ./3b
==10940==
--10940-- warning: L3 cache found, using its data for the LL simulation.
--10940-- run: /usr/bin/dsymutil "/.3b"
warning: no debug symbols in executable (-arch x86_64)
Matriz de 200x200

tiempo: 2.03401
==10940==
==10940== I refs:      234,795,397
==10940== I1 misses:    5,198
==10940== L1i misses:   2,924
==10940== I1 miss rate:  0.00%
==10940== L1i miss rate: 0.00%
==10940==
==10940== D refs:      196,220,878 (178,906,985 rd + 17,313,893 wr)
==10940== D1 misses:    538,088 ( 527,949 rd +   10,139 wr)
==10940== L1d misses:   17,462 (   8,206 rd +   9,256 wr)
==10940== D1 miss rate:  0.3% (   0.3% +   0.1% )
==10940== L1d miss rate: 0.0% (   0.0% +   0.1% )
==10940==
==10940== LL refs:      543,286 ( 533,147 rd +   10,139 wr)
==10940== LL misses:    20,386 (  11,130 rd +   9,256 wr)
==10940== LL miss rate:  0.0% (   0.0% +   0.1% )

```

Figure 3:

- Con la multiplicación de matrices de 6 bucles, como se muestra en la Figura4.:

```

[MacBook-Pro--de-Joss:matrixMult3b-6b Joss$ valgrind --tool=cachegrind ./6b
==11023== Cachegrind, a cache and branch-prediction profiler
==11023== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==11023== Using Valgrind-3.14.0.SVN and LibVEX; rerun with -h for copyright info
==11023== Command: ./6b
==11023==
--11023-- warning: L3 cache found, using its data for the LL simulation.
--11023-- run: /usr/bin/dsymutil "-.6b"
warning: no debug symbols in executable (-arch x86_64)
Matriz de 200x200

tiempo: 4.13469
==11023==
==11023== I refs:      463,897,554
==11023== I1 misses:    5,309
==11023== L1i misses:   2,954
==11023== I1 miss rate:  0.00%
==11023== L1i miss rate: 0.00%
==11023==
==11023== D refs:      382,417,686 (280,707,165 rd + 101,710,521 wr)
==11023== D1 misses:    78,480 ( 70,854 rd + 7,626 wr)
==11023== L1d misses:   17,472 ( 10,722 rd + 6,750 wr)
==11023== D1 miss rate:  0.0% (  0.0% +  0.0% )
==11023== L1d miss rate: 0.0% (  0.0% +  0.0% )
==11023==
==11023== LL refs:      83,789 ( 76,163 rd + 7,626 wr)
==11023== LL misses:    20,426 ( 13,676 rd + 6,750 wr)
==11023== LL miss rate:  0.0% (  0.0% +  0.0% )

```

Figure 4:

El BlockSize representa el tamaño del bloque que debe calcularse para cada iteración. Esto no debería ser más grande que el tamaño de la matriz. Así también se hizo la prueba con diferentes tamaño de bloque:

- BlockSize = 60 con una matriz de 200x200.

```

tiempo: 1.85002
==4573==
==4573== I refs:      610,497,297
==4573== I1 misses:    1,916
==4573== L1i misses:   1,758
==4573== I1 miss rate:  0.00%
==4573== L1i miss rate: 0.00%
==4573==
==4573== D refs:      296,654,527 (246,001,465 rd + 50,653,062 wr)
==4573== D1 misses:    399,338 ( 391,281 rd + 8,057 wr)
==4573== L1d misses:   16,961 ( 10,180 rd + 6,781 wr)
==4573== D1 miss rate:  0.1% (  0.2% +  0.0% )
==4573== L1d miss rate: 0.0% (  0.0% +  0.0% )
==4573==
==4573== LL refs:      401,254 ( 393,197 rd + 8,057 wr)
==4573== LL misses:    18,719 ( 11,938 rd + 6,781 wr)
==4573== LL miss rate:  0.0% (  0.0% +  0.0% )

```

Figure 5:

- BlockSize = 80 con una matriz de 200x200.

```

tiempo: 1.84107
==4651==
==4651== I refs:      608,985,449
==4651== I1 misses:    1,918
==4651== L1i misses:   1,760
==4651== I1 miss rate:  0.00%
==4651== L1i miss rate: 0.00%
==4651==
==4651== D refs:      295,287,539 (245,090,106 rd + 50,197,433 wr)
==4651== D1 misses:    440,403 ( 432,346 rd + 8,057 wr)
==4651== L1d misses:   16,961 ( 10,180 rd + 6,781 wr)
==4651== D1 miss rate:  0.1% (  0.2% +  0.0% )
==4651== L1d miss rate: 0.0% (  0.0% +  0.0% )
==4651==
==4651== LL refs:      442,321 ( 434,264 rd + 8,057 wr)
==4651== LL misses:    18,721 ( 11,940 rd + 6,781 wr)
==4651== LL miss rate:  0.0% (  0.0% +  0.0% )

```

Figure 6:

- BlockSize = 100 con una matriz de 200x200.

Se nota una diferencia como primero en el tiempo, ya que mientras va aumentando el tamaño del bloque el tiempo va ir reduciendo ya que no al momento de realizar la multiplicación de matrices de 6 bucles va ir incrementando el tamaño de bloques y se va a poder repartir mejor los datos. En la parte de los cache misses se nota que va ir aumentando cada que se va ir aumentando el tamaño de bloques.

```

tiempo: 1.8351
==4727==
==4727== I   refs:      605,023,753
==4727== I1  misses:        1,915
==4727== LLi misses:        1,757
==4727== I1  miss rate:      0.00%
==4727== LLi miss rate:    0.00%
==4727==
==4727== D   refs:      293,934,110 (244,187,802 rd + 49,746,308 wr)
==4727== D1  misses:        531,118 ( 523,061 rd +   8,057 wr)
==4727== LLd misses:        16,961 (  10,180 rd +   6,781 wr)
==4727== D1  miss rate:      0.2% (   0.2% +   0.0% )
==4727== LLd miss rate:    0.0% (   0.0% +   0.0% )
==4727==
==4727== LL refs:        533,033 ( 524,976 rd +   8,057 wr)
==4727== LL  misses:         18,718 (  11,937 rd +   6,781 wr)
==4727== LL  miss rate:    0.0% (   0.0% +   0.0% )

```

Figure 7:

3 Conclusiones

En la memoria cache al utilizarla con una gran cantidad de datos tiene un efecto de mejora muy significativa, optimizando el tiempo de ejecución del programa. Así, se puede ver que la diferencia de cache misses de la multiplicación de 6 bucles que es en bloques con respecto a la multiplicación de 3 bucles corresponde a casi mas de la mitad.