Algorítmos Paralelos
Three Nested Loop

Vanessa Santillana

23 Marzo 2017

1 Programa

Primeramente nuestro programa tiene por objetivo de multiplicar 2 matrices, sin embargo existen dos métodos de hacer tal multiplicación. También utilizaremos números randómicos al llenar las matrices.

1.1 Three Nested Loop

El primer método serian la multiplicación de filas por columnas de matrices de tamaños $n * m$.

```
1 void m_filas(int n, int m, int o,  int a[n][o], int b[n][m], int c[
      m][o]){
2     int i, j, k ;
3
4     for (i=0;i<n; i++)
5         for (j = 0; j<o; j++)
6           for (k=0;k<m; k++)
7               a[i][j] += b[i][k]* c[k][j] ;
8 }
```

El segundo método seria la multiplicación por columnas, y este quiere decir que la columna de la matriz A se multiplicara por el primer valor de la fila de B, y se suma a una matriz inicialmente vacía C que tendrá nuestro resultado.

```
1 void m_columnas(int n, int m, int o, int a[n][o], int b[n][m], int
      c[m][o]){
2     int i, j, k ;
3
4     for (i=0;i<n; i++)
5         for (k=0;k<m; k++)
6           for (j = 0; j<o; j++)
7               a[i][j] += b[i][k]* c[k][j];
8 }
```

1.2 Six Nested Loops

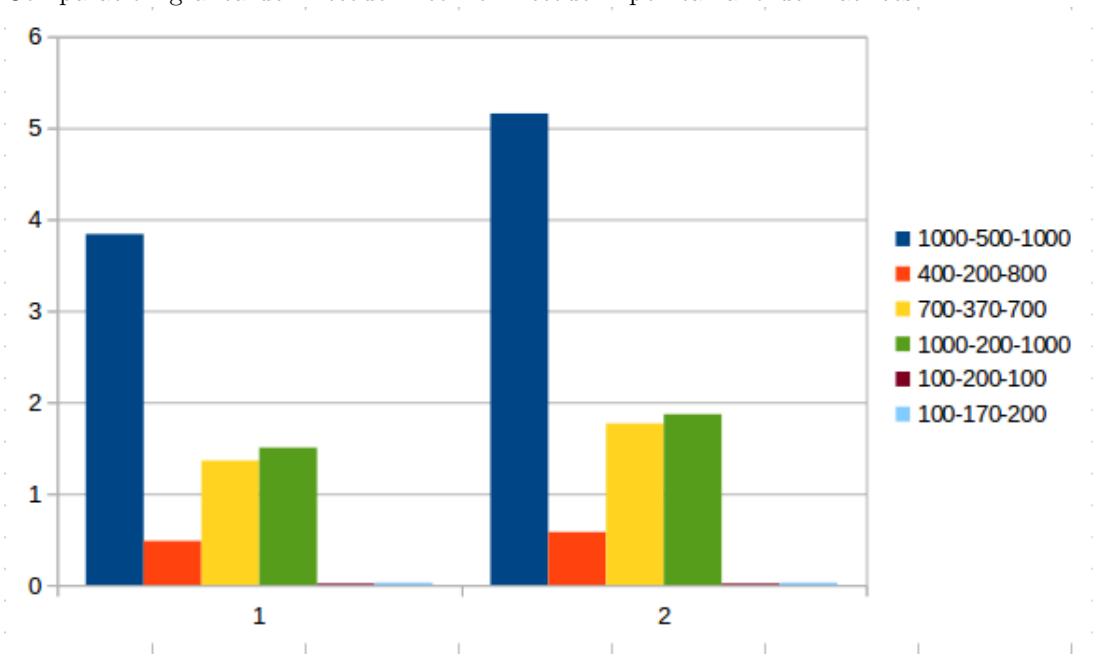
Este algoritmo se encarga de dividir en Blockes, es decir que al realizar la operación, en este caso una multiplicación de fila por columna, dividirá con un tamaño BlockSize la fila y después hará lo mismo con la columna de la segunda matriz, para obtener multiplicaciones mas pequeñas las cuales nos proporcionaran menos costo a la hora de hacer la operación con matrices mucho mas grandes.

```
1  void m_MatrixBlocked(int n, int m, int o, int BlockSize,int a[n][o
       ], int b[n][m],int c[m][o])
2  {
3    int iIter ,jIter ,kIter ,i ,j ,k;
4    for(iIter = 0; iIter < n; iIter+=BlockSize)
5          for(jIter = 0; jIter < m; jIter += BlockSize)
6              for(kIter = 0; kIter < o; kIter += BlockSize)
7                  for(i = iIter; i < MIN(iIter+BlockSize,n); i++)
8                      for(j = jIter; j < MIN(jIter+BlockSize,m); j++)
9                          for(k = kIter; k < MIN(kIter+BlockSize,o);
                               k++)
10                             a[i][j] += b[i][k] * c[k][j];
11 }
```

2 Comparación de los Nested Loops

2.1 Three Nested Loop

Comparación gráfica del método 1 con el método 2 por tamaño de matrices.



En la siguiente tabla se puede apreciar la diferencia de realizar la operación de multiplicación de fila por columna y la de multiplicar columna por columna almacenando las sumas que requiere ésta.

Métodos	1000-500-1000	1000-200-1000	700-370-700	400-200-800	100-170-200	100-200-100
Filas	0.3837597	0.1501527	0.1360849	0.0481088	0. 025742	0.0015253
Columnas	0.5155229	0.1868259	0.1766762	0.0578829	0.0026034	0.0015364

Table 1: Comparación de Three Nested Loop por columnas y por filas.

2.2 Six Nested Loop

Tamaño de Matriz	Tamaño del Bloque	Three Nested Loop	Six Nested Loops
100-1500-100	50	0.113	0.108
200-500-200	50	0.143	0.137
300-500-300	70	0.343	0.291
350-550-350	70	0.503	0.443
350-550-350	100	0.503	0.429
100-2000-100	100	0.142	0.126

Table 2: Comparación de Three Nested Loop y Six Nested Loops.

Tamaño de Matriz	Tamaño del Bloque	Three Nested Loop	Six Nested Loops
200-1000-200	1	0.286	0.600
200-1000-200	50	0.286	0.258
200-1000-200	100	0.286	0.279

Table 3: Comparación de Three Nested Loop y Six Nested Loops por tamaño de Bloque.

3 Pruebas con Valgrind

Three Nested Loop

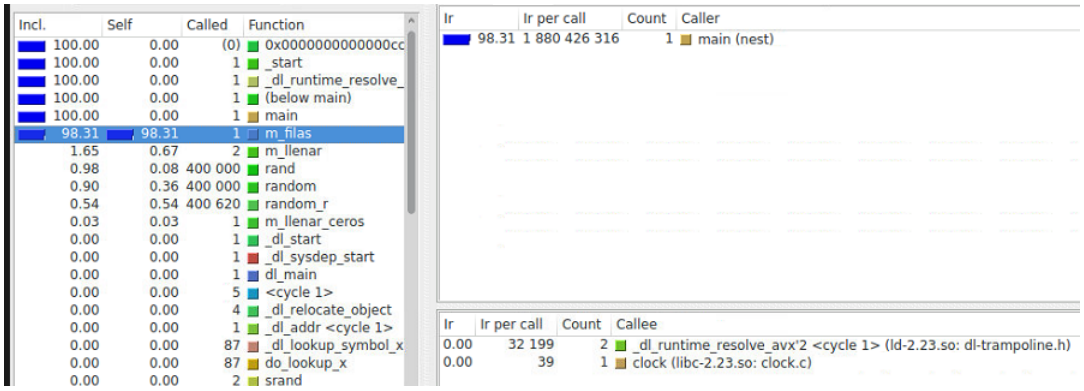
```
==13235== Cachegrind, a cache and branch-prediction profiler
==13235== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==13235== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==13235== Command: ./nest
==13235==
--13235-- warning: L3 cache found, using its data for the LL simulation.
13102.118000
==13235==
==13235== I   refs:      1,912,702,504
==13235== I1  misses:            1,075
==13235== LLi misses:            1,062
==13235== I1  miss rate:          0.00%
==13235== LLi miss rate:          0.00%
==13235==
==13235== D   refs:        774,190,737  (730,093,627 rd   + 44,097,110 wr)
==13235== D1  misses:       42,553,111  ( 42,524,977 rd   +     28,134 wr)
==13235== LLd misses:           29,967  (      1,953 rd   +     28,014 wr)
==13235== D1  miss rate:           5.5% (        5.8%     +        0.1%  )
==13235== LLd miss rate:           0.0% (        0.0%     +        0.1%  )
==13235==
==13235== LL refs:          42,554,186  ( 42,526,052 rd   +     28,134 wr)
==13235== LL misses:            31,029  (      3,015 rd   +     28,014 wr)
==13235== LL miss rate:            0.0% (        0.0%     +        0.1%  )
```

Six Nested Loop

```
==13089== Cachegrind, a cache and branch-prediction profiler
==13089== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==13089== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==13089== Command: ./nest
==13089==
--13089-- warning: L3 cache found, using its data for the LL simulation.
14000.744000
==13089==
==13089== I   refs:      2,128,666,661
==13089== I1  misses:            1,075
==13089== LLi misses:            1,063
==13089== I1  miss rate:          0.00%
==13089== LLi miss rate:          0.00%
==13089==
==13089== D   refs:        862,929,327  (818,056,011 rd   + 44,873,316 wr)
==13089== D1  misses:          221,904  (    193,768 rd   +     28,136 wr)
==13089== LLd misses:           29,967  (      1,953 rd   +     28,014 wr)
==13089== D1  miss rate:           0.0% (        0.0%     +        0.1%  )
==13089== LLd miss rate:           0.0% (        0.0%     +        0.1%  )
==13089==
==13089== LL refs:             222,979  (    194,843 rd   +     28,136 wr)
==13089== LL misses:            31,030  (      3,016 rd   +     28,014 wr)
==13089== LL miss rate:            0.0% (        0.0%     +        0.1%  )
```

En el primer Debug con Valgrid se observa los D1 misses 42,553,111, lo que significa que no se encontro los datos en la cache, pero si vemos c el caso de Six Nested Loop el cual tiene una cantidad considerable de menos misses con 221,904.

4 KCachegrind



Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000000cc
100.00	0.00	1	_start
100.00	0.00	1	_dl_runtime_resolve_
100.00	0.00	1	(below main)
100.00	0.00	1	main
98.31	98.31	1	m_filas
1.65	0.67	2	m_llenar
0.98	0.08	400 000	rand
0.90	0.36	400 000	random
0.54	0.54	400 620	random_r
0.03	0.03	1	m_llenar_ceros
0.00	0.00	1	_dl_start
0.00	0.00	1	_dl_sysdep_start
0.00	0.00	1	dl_main
0.00	0.00	5	<cycle 1>
0.00	0.00	4	_dl_relocate_object
0.00	0.00	1	_dl_addr <cycle 1>
0.00	0.00	87	_dl_lookup_symbol_x
0.00	0.00	87	do_lookup_x
0.00	0.00	2	srand

Ir	Ir per call	Count	Caller
98.31	1 880 426 316	1	main (nest)

Ir	Ir per call	Count	Callee
0.00	32 199	2	_dl_runtime_resolve_avx'2 <cycle 1> (ld-2.23.so: dl-trampoline.h)
0.00	39	1	clock (libc-2.23.so: clock.c)

Self	Function	Location
98.48	m_MatrixBlocked	(unknown)
0.58	m_llenar	(unknown)
0.49	random_r	random_r.c
0.32	random	random.c
0.08	rand	rand.c
0.03	m_llenar_ceros	(unknown)
0.02	(unknown)	(unknown)
0.00	_dl_addr	dl-addr.c
0.00	do_lookup_x	dl-lookup.c
0.00	_dl_lookup_symbol_x	dl-lookup.c
0.00	_dl_relocate_object	dl-machine.h
0.00	_dl_relocate_object	do-rel.h

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	98.48	98.48	Ir	
L1 Instr. Fetch Miss	0.93	0.93	I1mr	
LL Instr. Fetch Miss	0.94	0.94	ILmr	
Data Read Access	98.79	98.79	Dr	
L1 Data Read Miss	98.67	98.67	D1mr	
LL Data Read Miss	0.05	0.05	DLmr	
Data Write Access	90.96	90.96	Dw	
L1 Data Write Miss	0.02	0.02	D1mw	
LL Data Write Miss	0.00	0.00	DLmw	
L1 Miss Sum	85.75	85.75	L1m	= I1mr + D1mr + D1mw
Last-level Miss Sum	0.04	0.04	LLm	= ILmr + DLmr + DLmw
Cycle Estimation	98.32	98.32	CEst	= Ir + 10 L1m + 100 LLm

5 Resultados

Si usamos el segundo método, podemos observar que el recorrido es por columna en la matriz A, entonces cuando quiere obtener el siguiente dígito de la columna tendrá que buscar directamente en memoria, ya que los datos no se encuentran en cache por ser una variable temporal, finalmente este recorrido no seria el mas efectivo.
Usando el primer método, nos damos cuenta que al hacer un recorrido por fila en la matriz A, se convierte en una variable espacial, la cual carga toda la fila y al hacer la operación de multiplicación no necesitara ir hasta memoria, ya que los datos se encuentra en la cache, de tal manera que es más efectivo.

En comparación del Three Nested Loop y Six Nested Loop, podemos observar en la tabla 2 que tiene una mejora en tiempos el Six Nested Loop, debido a que al dividir la matriz en bloques mas pequeños se obtiene que los datos sean más rápidamente leídos de la memoria cache, y evitar leer del disco duro.

6 Conclusión

Según las pruebas hechas, la multiplicación de matriz por filas, es decir el primer método es la mejor opción con el Three Nested Loop, ya que el tiempo que necesita para obtener los datos es menor gracias a la memoria cache.

En el caso de Six Nested Loop, como se ha ido comprobando es mucho más efectivo para matrices bastante grandes, pero cabe decir que depende bastante del tamaño del bloque en el que se dividirá la matriz, sin embargo no es recomendable usar con un tamaño pequeño como de 1 porque de esta manera tarda mas que el método normal Three Nested Loop.