Práctica I [Cache Misses]

Percy Maldonado Quispe
Ciencia de la Computación
Arequipa-UCSP

I. Introducción

En esta ocasión pondremos a prueba dos algoritmos que satisfacen la multiplicación de matrices.

1. Multiplicación normal, tres for anidados.
2. Multiplicación por bloques, seis for anidados.

Se realizará experimentos para ver cual solución es más rápida en cuanto tiempo, y ver el porcentaje de Misses Cache.

II. Hardware

- Fabricante: TOSHIBA
- Model: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
- Arquitectura: x86_64.
- CPU(s): 4.

III. Complejidad Algoritmica

- Multiplicación Normal. Matriz cuadrada de tamaño n. $\mathcal{O}(n^3)$
- Multiplicación por Bloques.

IV. Comparación por Tiempo

En este experimento tomaremos el tiempo (en segundos) en que demora el algoritmo clásico con el de bloques. Sucede un fenomeno al realizar el experimento. En matrices pequeñas (128x128 ó 256x256) la multiplicación clásica debería ser más rapida, y efectivamente lo es, pero lo extraño es con el optimizador de GCC, los resultados son totalmente distintos, por bloques es más rápida.

Compilador: gcc (Ubuntu 5.4.0-6ubuntu1 16.04.10) 5.4.0 20160609

IV-A. Sin Optimización GCC

Al ver los tiempos sin optimización, vemos que la multiplicación clásica es súper rápida en comparación al de bloques, muy por encima, pero cuando llegamos a matrices grandes tales como 1024x1024 el tiempo comienza a disminuir, vemos claramente que con una matriz de 2048x2048 es más rápida que una clásica, con bloques de 512.

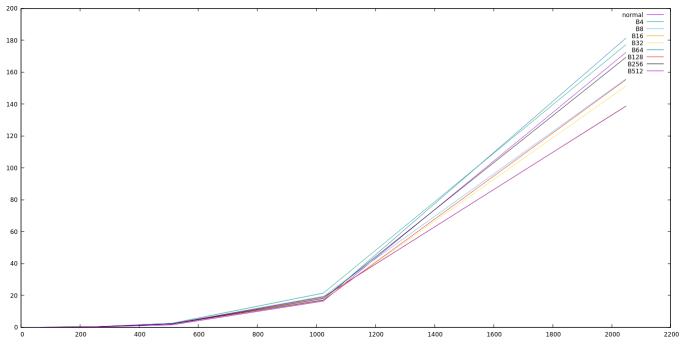

Figura 1. GCC Normal

IV-B. Optimización GCC -O2

GCC realiza casi todas las optimizaciones admitidas que no implican una compensación de velocidad de espacio. En comparación con -O, esta opción aumenta el tiempo de compilación y el rendimiento del código generado [GCC,].

En contra parte con el de sin optimización, con O2 se ve que el producto por bloques comienza a superar a la clásica desde matrices de tamaño 512x512.

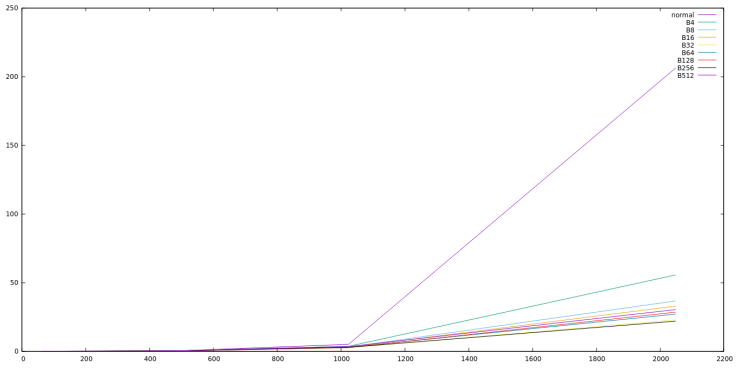

Figura 2. GCC -O2

IV-C. Optimización GCC -O3

Optimizar aún más. -O3 activa todas las optimizaciones especificadas por -O2 y también activa indicadores de optimización nuevos.

Con la optimización del compilador de GCC -O3 el algoritmo por bloques comienza a ser más rápida a partir de matrices de 128x128.

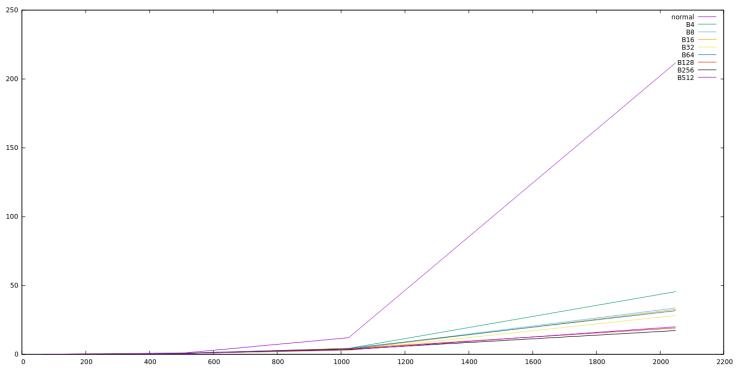

Figura 3. GCC -O3

Cuadro I
COMPARACIÓN TIEMPO VS TAMAÑO [GCC NORMAL]

N	normal	B4	B8	B16	B32	B64	B128	B256	B512
64	0.0020000	0.0050000	0.0050000	0.0040000	0.0060000	0.0060000	0.0060000	0.0060000	0.0050000
128	0.0180000	0.0420000	0.0370000	0.0330000	0.0340000	0.0320000	0.0320000	0.0310000	0.0320000
256	0.1660000	0.3510000	0.3000000	0.2770000	0.2790000	0.2720000	0.2660000	0.2690000	0.2770000
512	1.4870000	2.5300000	2.2450000	2.1240000	2.2540000	2.1450000	2.0320000	2.0570000	2.2090000
1024	16.3760000	21.3830000	19.4060000	18.7560000	17.2190000	17.0930000	16.9020000	18.0650000	18.9610000
2048	172.3020000	177.1650000	155.6790000	138.7660000	151.1500000	181.2660000	155.2010000	169.1920000	138.5510000

Cuadro II
COMPARACIÓN TIEMPO VS TAMAÑO [GCC -O2]

N	normal	B4	B8	B16	B32	B64	B128	B256	B512
64	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
128	0.0030000	0.0060000	0.0070000	0.0040000	0.0040000	0.0050000	0.0040000	0.0030000	0.0040000
256	0.0390000	0.0480000	0.0400000	0.0390000	0.0380000	0.0360000	0.0340000	0.0360000	0.0370000
512	0.3930000	0.3850000	0.3330000	0.3090000	0.3130000	0.2970000	0.2930000	0.3010000	0.3610000
1024	4.9660000	3.5490000	2.9350000	2.8220000	2.5310000	2.7390000	2.5600000	2.7750000	3.3290000
2048	206.0630000	55.4380000	36.5430000	32.7560000	22.7040000	26.9900000	28.3700000	21.8160000	30.2420000

Cuadro III
COMPARACIÓN TIEMPO VS TAMAÑO [GCC -O3]

N	normal	B4	B8	B16	B32	B64	B128	B256	B512
64	0.0000000	0.0010000	0.0010000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
128	0.0080000	0.0090000	0.0090000	0.0070000	0.0080000	0.0070000	0.0080000	0.0070000	0.0070000
256	0.0770000	0.0790000	0.0700000	0.0650000	0.0620000	0.0620000	0.0610000	0.0650000	0.0650000
512	0.8710000	0.7210000	0.5750000	0.5200000	0.4940000	0.5100000	0.5000000	0.5210000	0.6090000
1024	11.8840000	4.1470000	3.5280000	3.2610000	2.6730000	4.1080000	3.8810000	3.3890000	3.0410000
2048	211.3960000	45.3310000	33.4710000	32.3610000	28.0240000	31.4440000	18.9790000	17.0430000	19.8100000

V. ANÁLISIS CON VALGRIND

V-A. M. Normal

Al ver con Valgrind los Caches Misses, vemos que en total se hace en el bloque D de datos, una referencia de 3 106 204 714 de acceso, tanto escritura como lectura. Los Misses en D son 158 688 109 [Lectura y Escritura]. El porcentaje de Cache Misses es de 5.1 % con respector a la cantidad de datos accedida.

```
time normal_mult 52.2590000000 s
==19282==
==19282== I   refs:      5,817,638,208
==19282== I1  misses:            1,788
==19282== LLi misses:            1,725
==19282== I1  miss rate:          0.00%
==19282== LLi miss rate:          0.00%
==19282==
==19282== D   refs:      3,106,204,714  (2,967,023,199 rd   + 139,181,515 wr)
==19282== D1  misses:      158,688,109  (  158,636,051 rd   +      52,058 wr)
==19282== LLd misses:           65,049  (       13,866 rd   +      51,183 wr)
==19282== D1  miss rate:           5.1% (          5.3%     +         0.0%  )
==19282== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==19282==
==19282== LL refs:         158,689,897  (  158,637,839 rd   +      52,058 wr)
==19282== LL misses:            66,774  (       15,591 rd   +      51,183 wr)
==19282== LL miss rate:            0.0% (          0.0%     +         0.0%  )
maldonado@maldonado-sarah ~/Paralelos $
```

Figura 4. Valgrind Normal

V-B. M. Bloques

Al igual que en analisis del normal con valgrind, veremos el porcentaje de Cache Misses en el de Bloques. Para ello tenemos que el total de acceso en lectura y escritura es de 5 135 157 187. En Cache Misses tenemos un total de 778 905. Por lo tanto el porcentaje de Cache Misses es de aproximadamente 0.0 %.

```
time blocked_mult 79.5240000000 s
==19617==
==19617== I   refs:      9,287,517,763
==19617== I1  misses:            1,794
==19617== LLi misses:            1,731
==19617== I1  miss rate:          0.00%
==19617== LLi miss rate:          0.00%
==19617==
==19617== D   refs:      5,135,157,187  (4,277,545,693 rd   + 857,611,494 wr)
==19617== D1  misses:        778,905  (       726,866 rd   +      52,039 wr)
==19617== LLd misses:         65,162  (        13,979 rd   +      51,183 wr)
==19617== D1  miss rate:         0.0% (           0.0%     +         0.0%  )
==19617== LLd miss rate:         0.0% (           0.0%     +         0.0%  )
==19617==
==19617== LL refs:           780,699  (       728,660 rd   +      52,039 wr)
==19617== LL misses:          66,893  (        15,710 rd   +      51,183 wr)
==19617== LL miss rate:          0.0% (           0.0%     +         0.0%  )
maldonado@maldonado-sarah ~/Paralelos $
```

Figura 5. Valgrind Bloques

VI. ANÁLISIS CON KCACHEGRIND

KCachegrind no es más que solo una interfaz gráfica para interpretar lo generado por Valgrind.

VI-A. M. Normal

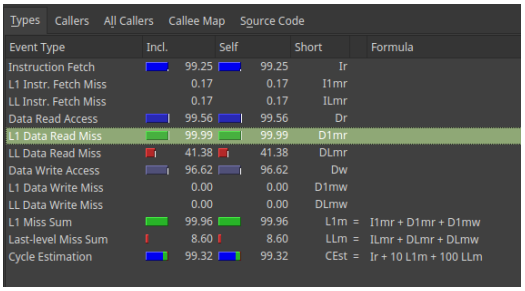

Figura 6. KCachegrind Normal

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	79.53	79.53	Ir	
L1 Instr. Fetch Miss	0.33	0.33	I1mr	
LL Instr. Fetch Miss	0.35	0.35	ILmr	
Data Read Access	76.32	76.32	Dr	
L1 Data Read Miss	98.08	98.08	D1mr	
LL Data Read Miss	41.86	41.86	DLmr	
Data Write Access	49.47	49.47	Dw	
L1 Data Write Miss	0.00	0.00	D1mw	
LL Data Write Miss	0.00	0.00	DLmw	
L1 Miss Sum	91.31	91.31	L1m	= I1mr + D1mr + D1mw
Last-level Miss Sum	8.76	8.76	LLm	= ILmr + DLmr + DLmw
Cycle Estimation	79.49	79.49	CEst	= Ir + 10 L1m + 100 LLm

Figura 7. KCachegrind Bloques

VII. Conclusión

Bueno, es muy complicado tener una afirmación clara, para ello tendremos en cuenta primero la ejecución sin optimizaciones de parte del Compilador. Entonces dicho esto, vemos que la multiplicación clásica es mejor para matrices pequeñas, con pequeñas me refiero menores a 2000x2000, una vez pasado por ese tamaño, en los experimentos realizados vemos que la M. por bloques comienza a tener mejor desempeño.

Ahora pasemos con la optimización de GCC, defrente con -O3, vemos que el desempeño de la M. por bloques es superior a la M. clásica. desde matrices 'pequeñas' (64x64), entonces podemos afirmar que si ponemos en producción un código para la multiplicación en modo Release (-O3), el producto por bloques es mucho mejor que la clásica.

En cuanto a Cache Misses, en todo momento $[> 32]$ se ve que la Multiplicación por Bloques genera menos Cache Misses, y ¿esto por qué?, se estara pregutando, la respuesta es sencilla, el numero de bloques a utilizar es pequeño, y esto nos asegura que no vamos a estar cargando gran cantidad de datos a la memoria cache, e incluso llenarla, con el proceso de bloques se llega a un equilibrio entre que cantidad de memoria cache puedo soportar, y solo cargar esa parte.

Referencias

[GCC,] Options that control optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [Web, accedido el 27-08-18].

Índice de figuras

Índice de cuadros

Índice