MASTER SIANI ULPGC

Computación Paralela

Paseo por las optimizaciones

Optimización basada en el uso de la cache

Sergio Marrero Marrero

Universidad de Las Palmas de Gran Canaria

14/03/2016

Índice

1. Ejercicio 4. Gestionando la memoria. Blocking.

1.1. Los niveles de la memoria

Existen diferentes niveles de memoria. Lo ideal sería que el procesador tuviera acceso a una memoria infinita en un tiempo mínimo, sin embargo esto no es posible. El precio del aumento de la capacidad a la memoria es el incremento del tiempo que hace falta para acceder a esta. Por esta razón, la memoria se divide en niveles. Cuanto más grande es el nivel, mayor es la cantidad de memoria a la que se tiene acceso, pero mayor es el tiempo necesario para ello. La siguiente imagen ofrece una visión bastante simplifcada de lo que acabamos de decir.

Figura 1: Niveles de memoria

Por otro lado, la memoria caché se relaciona con cada procesador de la siguiente forma:

Nivel	pulso(clk)	Tamaño
L1	1	32 Kb
L2	10	256 Kb
L3	20	3 Mb
RAM	200	100 Mb
Disco Duro	miles	Giga/Tera

Cuadro 1: Pulsos de reloj y capacidad en memorias

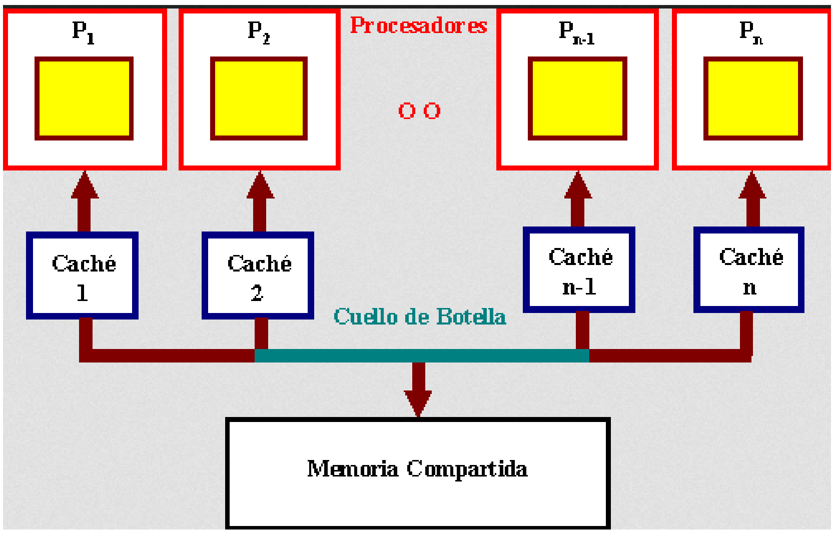

Figura 2: Niveles de memoria propia de cada procesador

Como se ve en esta nueva imagen, la memoria caché se divide en subniveles. Los niveles privados de cada procesador, que en la imagen esta representados como $cache_i$ se dividen en L_1 y L_2. Estos dos subniveles de la caché son de uso particular de cada procesador. El nivel L_3 de la caché ya es compartido por todos. Después de la memoria caché vendría la ram y después de esto, la memoria del disco.

A continuación se presenta en una tabla los tiempos de acceso en pulsos de reloj y el tamaño de cada nivel de memoria.

Teniendo en cuenta esta características, queda claro cual va a ser el procedimiento de optimización. Se intentará en todo lo posible que el procesador nunca tenga salir de la memoria L1 a buscar los datos. Sin embargo tenemos un problema y es el siguiente. Siguiendo con las matrices que hemos manejado hasta ahora podemos ver como se hace necesario acceder a la memoria RAM:

$$tam = 4096 * 4096 * 3 * 4 = 192MB$$

Teniendo en cuenta que L1 tiene 32 KB se puede ver donde está la dificultad de este asunto. Nada más ni nada menos que $192 * 1024/32 = 6144$ veces más contenido que continente.

La técnica que se va a plantear es la de dividir la multiplicación de matrices en bloques que quepan dentro de la L1. Hasta que no se termina todo el bloque, no se vuelve a salir de la L1 a buscar más datos. De esta forma conseguimo minimizar al máximo el número de salidas de dicha memoria. Simplemente se dividirá la operación en 3 bloques de 32 x 32, es decir: $3 * 32 * 32 * 4 = 12KB$ que es menor al tamaño de la L1.

1.2. Compilación del código y análisis

El código que se deberá implementar es el siguiente:

```
void dgemm_blocking (int n, int si , int sj , int sk , float* A, float* B
int i ,j ,k,x;
for ( i = si ; i < si+BLOCKSIZE; i+=UNROLL*4 )
        for ( j = sj ; j < sj+BLOCKSIZE; j++ ) {
        __m128 c[4];
        for ( x = 0; x < UNROLL; x++ )
                c[x] = _mm_load_ps(C+i+x*4+j*n);
                for( k = sk; k < sk+BLOCKSIZE; k++ ) {
                        __m128 b = _mm_load_ps1(B+k+j*n); /* replica 4
                        for ( x = 0; x < UNROLL; x++ )
                        c[x] = _mm_add_ps(c[x], /* c[x] += A[i+x*4][k]
                        _mm_mul_ps(_mm_load_ps(A+i+x*4+k*n), b ));
                        }
                        for ( x = 0; x < UNROLL; x++ )
                        _mm_store_ps(C+i+x*4+j*n, c[x]); /* C[i][j] =
                        }
                }
```

```
void dgemm (int n, float* A, float* B, float* C){
int i,j,k;

for ( j = 0; j < n; j += BLOCKSIZE )
        for ( i = 0; i < n; i += BLOCKSIZE )
                for ( k = 0; k < n; k += BLOCKSIZE)
                dgemm_blocking (n, i, j, k, A, B, C);
                }
```

Como se puede observar en este código, se definen dos funciones. La función *dgemm* recorre las matrices en saltos de 32, es decir en saltos del tamaño del bloque que hemos elegido *BLOCKSIZE=32*. Por otro lado, la función *dgemmblocking* recorre el interior de cada bloque.

Para que esta funcione hay que definir:

```
#define BLOCKSIZE 32
```

.

La forma de compilar esta función será la siguiente:

```
gcc -msse -o matrix5blocking4096 matrix5blocking4096.c -O3
```

Los resultados de la ejecución del código son los siguientes:

Optimización	-gcc	tiempo(s)	MFlops/s	Archivo (Kb)	Speed up
Sin optimizar	-O	1284.4 (21.408)	107.0	8.7	—-
-O2	-O2	739.6 (12.327)	185.8	8.7	1
Par. datos	msse,-O3	174.9 (2.92)	785.7	8.7	4.2287
Segmentación	-O3	65.1 (1.09)	2111.5	8.7	11.361
Blocking	-O3	18	7627.7	8.7	41.089

Cuadro 2: Ejecución minimizando las salidas de L1. Blocking

Se observa que el speed-up es bastante grande. **Se ha conseguido optimizar enormemente la multiplicación de matrices sin haber tenido**

la necesidad de utilizar más de un procesador. Se puede decir ahora, que una vez se ha conseguido que un solo procesador funcione al máximo rendimiento, tiene sentido comenzar a utilizar una paralelización de otros procesadores.