MASTER SIANI ULPGC
Computación Paralela
Paseo por las optimizaciones
Optimización basada en paralelismo de datos

Sergio Marrero Marrero
Universidad de Las Palmas de Gran Canaria

14/03/2016

Índice

1. Ejercicio 2. Paralelismo de datos

1.1. Paralelismo de datos en nuestras matrices

Para entender este tipo de optimización, visualicemos la siguiente multiplicación de matrices con $n = 3$.

$$C_{n\times n} = B_{n\times n}A_{n\times n} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix},$$

Esto es:

$$\begin{aligned} c_{11} &= b_{11}a_{11} + b_{12}a_{21} + b_{13}a_{31} \\ c_{21} &= b_{21}a_{11} + b_{22}a_{21} + b_{23}a_{31} \\ c_{31} &= b_{31}a_{11} + b_{32}a_{21} + b_{13}a_{31} \\ c_{12} &= b_{11}a_{12} + b_{12}a_{22} + b_{13}a_{32} \\ c_{22} &= b_{21}a_{12} + b_{22}a_{22} + b_{23}a_{32} \\ c_{32} &= b_{31}a_{12} + b_{32}a_{22} + b_{33}a_{32} \\ c_{13} &= b_{11}a_{13} + b_{12}a_{23} + b_{13}a_{33} \\ c_{13} &= b_{21}a_{13} + b_{22}a_{23} + b_{23}a_{33} \\ c_{13} &= b_{31}a_{13} + b_{32}a_{23} + b_{33}a_{33} \end{aligned}$$

Ahora vamos a expresar las tres primeras sumas anteriores(que corresponden con la primera columna de C)de una forma que hará visible donde se pueden encontrar el método de optimización:

$$\begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} = b_{11} \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} + b_{21} \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} + b_{31} \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix}$$

Se puede observar que para obtener la primera fila de C, utilizamos toda la matriz A y solamente la primera fila de B. El objetivo será aprovechar esta característica. En lugar de calcular c_{11} de manera individual, para luego calcular c_{21} y volver a buscar valor previamente utilzado $b11$, paralelizamos estas operaciones y calculamos todas las que el procesador sea capaz, en

paralelo. En el caso que hemos puesto, paralelizando las tres operaciones agotarían las posibilidades, ya que nuestra matriz es de n=3. Sin embargo, dependiendo de las características del procesador la paralelización de datos varía.

A partir de los procesadores Intel Pentium III se introduce este tipo de paralelización, denominada 'Single-Instruction Multiple Data'(SIMD). Se añaden aquí 79 instrucciones vectoriales, pudiendose utilizara únicamente para datos de simple precisión, 32 bits. En nuestro caso, el procesador permite calcular cuatro elementos en paralelo, es decir: $4x32$ bit simple-precision

Para poder utilziar este tipo de optimizaciones hay que hacerlo a través de los *intrinsics*. Un *intrinsic* es una función C/C++ conocida por el compilador que hace corresponder una sentencia a una secuencia de una o varias instrucciones vectoriales. Por ejemplo, el *intrinsic* que nos permite vectorizar la suma es el siguiente:

```
c0 = _mm_add_ps (__m128 a, __m128 b);
```

Como se verá a continuación, las operaciones que vayamos a realizar tendremos que adaptarlas con el *intrinsic* adecuado, que permita que el compilador lo reconozca y pueda vectorizarlo sin problemas.

El código que se va añadir al código anterior para poder implementar la parelización de datos es el siguiente:

```
void dgemm (int n, float* A, float* B, float* C)
{
 int i,j,k;
 for (i = 0; i < n; i+=4)
   for (j = 0; j < n; ++j)
   {
        __m128 c0=_mm_load_ps(C+i+j*n);
    float cij = C[i+j*n]; /* cij = C[i][j] */
    for(k = 0; k < n; k++ )
     c0 = _mm_add_ps(c0, /* c0 += A[i][k]*B[k][j] */
        _mm_mul_ps(_mm_load_ps(A+i+k*n),
        _mm_load_ps1(B+k+j*n) ));
```

```
_mm_store_ps(C+i+j*n, c0); /* C[i][j] = c0 */

    }
}
```

Se observa lo siguiente. Con respecto al código anterior, aparecen las siguientes 'lineas extrañas':

```
__m128 c0=_mm_load_ps(C+i+j*n);

c0 =_mm_add_ps(c0,_mm_mul_ps(_mm_load_ps(A+i+k*n),
_mm_load_ps1(B+k+j*n) ));

_mm_store_ps(C+i+j*n, c0);
```

La primera instruccion indica que se carguen 4 elementos de memoria ($4x32 = 128bits$) para la matriz C. La siguiente instrucción es permite realizar la suma entre las distintas multiplicaciones de los elementos de las matrices A y B. Estos elementos tendrán que multiplicarse entre sí, y por ello también estarán incluidos dentro de la instrucción *mult* y a su vez tendrán que ser cargados con la función *load*.Finalmente, el resultado de estas operaciones se almacenarán en las memorias que se reservaron al principio con la funcion *store*

1.2. Compilación del código y analisis

Para que el código funcione, hay incluir en el código fuente la siguiente librería:

```
#include <xmmintrin.h>
```

Se compilará el código con la siguiente instrucción:

```
gcc -msse -o matrix2vectorizado4096 matrix2vectorizado4096.c -O3
```

Respecto de la forma de compilación anterior, vemos dos cambios. En primer lugar aparace la palabra *-msse* haciendo referencia al método de optimización que quiere usarse *Streaming SIMD Extension* y además se ha aumentado al nivel -O3.

El nivel de optimización -O3 es el nivel más alto del compilador GCC. En este nivel se pone más énfases en la velocidad que en el tamaño del código. Este nivel de optimización incluye todas las optimizaciones incluidas en el nivel -O2. El nivel -O3 puede producir un aumento en el tamaño del código ejecutable pudiendo generar relentizaciones indeseables debido a un desbordamiento en la memoria cache.

A continuación se exponen los resultados obtenidos, comparandolo con la mejor versión secuencial, la cual se utilizará de referencia, como ya se apuntó anteriormente:

Optimización	-gcc	tiempo(s)	MFlops/s	Archivo (Kb)	Speed up
Sin optimizar	—	1284.4 (21.408)	107.0	8.7	—-
-O2	—	739.6 (12.327)	185.8	8.7	1
Par. datos	—	174.9 (12.327)	785.7	8.7	4.2287

Cuadro 1: Ejecución con paralelización de datos

Se observa que el fichero ejecutable no ha aumentado su tamaño.