

Laboratorio Sesión 08: Investigando la jerarquía de memoria (II)

Objetivo

El objetivo de la sesión es observar el efecto que puede tener la jerarquía de memoria en el rendimiento de un programa.

El precompilador de C y la compilación condicional

Antes de ejecutar el compilador de C siempre se llama al precompilador. El precompilador de C se llama cpp. El precompilador se encarga básicamente de:

- La inclusión de los ficheros de cabeceras (p.e. #include <stdio.h>).
- La expansión de macros.
- La compilación condicional. Usando algunas directivas especiales es posible incluir o excluir partes del programa original en función de condiciones varias.

En los programas de la sesión de hoy se utiliza la compilación condicional. En la parte incial de cada programa encontramos lo siguiente:

```
#ifndef N
#define N 256
#endif
/* Dimensión por defecto */
```

Estas tres sentencias indican que si la constante N no está definida, entonces toma el valor 256. La forma de darle un valor diferente a esta constante es al compilar:

```
$> gcc mm-ijk.c tiempo.c -DN=64 -o IJK64
```

La opción -DN=64, es la forma de inicializar N con el valor 64. Este mecanismo es muy útil para redefinir los parámetros de un programa en tiempo de compilación, sin necesidad de editar el programa de nuevo.

¿Influye la jerarquía de memoria en el tiempo de ejecución de los programas?

La mejor forma de comprobar la influencia de la jerarquía de memoria en el tiempo de ejecución de los programas es mediante un ejemplo. En el paquete de programas de esta sesión tenéis 3 programas: mm-ijk.c, mm-jki.c y mm-ki.j.c. Estos programas son 3 de las 6 formas ijk del producto de matrices. En el programa mm-ijk.c se evalúa la siguiente porción de código (ijk denota que el bucle más externo es el bucle controlado por la variable i y el más interno el controlado por k):

```
t1 = GetTime();

for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];

t2 = GetTime();
total = t2 - t1;
```

El conjunto de los tres bucles anidados realiza el producto de matrices $C = A * B$. Como en las prácticas anteriores, las llamadas a las rutinas `GetTime()` sirven para calcular el tiempo de ejecución de este código. La rutina `GetTime()` nos da el tiempo empleado por el programa hasta ese instante; la diferencia ($t_2 - t_1$) nos da el tiempo de ejecución del bucle medido en milisegundos. Este es el esquema básico a utilizar para medir el tiempo de ejecución de una porción de código. En el programa `mm-jki.c`, el código a evaluar es el siguiente:

```
t1 = GetTime();  
  
for (j=0; j<N; j++)  
    for (k=0; k<N; k++)  
        for (i=0; i<N; i++)  
            C[i][j] = C[i][j] + A[i][k] * B[k][j];  
  
t2 = GetTime();  
total = t2 - t1;
```

La única diferencia con respecto al código anterior es la ordenación de los bucles y, en consecuencia, el orden en que se accede a los elementos de las matrices (algo similar ocurre para el programa `mm-ki.j.c`). Un detalle fundamental es que los tres programas realizan, de distinta forma, la misma operación: el producto de matrices.

Para compilar y ejecutar estos programas en Linux hay que hacer lo siguiente:

```
$> gcc mm-ijk.c tiempo.c -DN=64 -o IJK64  
$> ./IJK64
```

Donde `-DN=64` indica que el tamaño de las matrices es $N=64$ (si no se pone nada el valor por defecto es $N=256$); y `IJK64` es el nombre del fichero ejecutable.

El ejecutable se comporta de forma diferente según el tamaño de la matriz. Si la matriz tiene un tamaño $N=6$ (o menor), la aplicación vuelca por pantalla el contenido de la matriz resultante. Esto es útil para comprobar que los tres programas tienen el mismo resultado. Para tamaños mayores la aplicación devuelve el tiempo de ejecución medido en tics de reloj.

Una optimización adicional

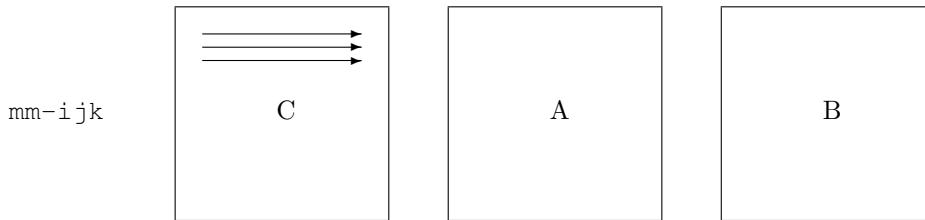
Una posible optimización del producto de matrices podría ser el siguiente:

```
t1 = GetTime();  
  
for (i=0; i<N; i++)  
    for (j=0; j<N; j++) {  
        tmp = C[i][j];  
        for (k=0; k<N; k++)  
            tmp = tmp + A[i][k] * B[k][j];  
        C[i][j] = tmp;  
    }  
  
t2 = GetTime();  
total = t2 - t1;
```

Esta optimización se puede aplicar a las tres formas anteriores (nota importante: la optimización se aplica de forma diferente en los tres códigos). Este código optimizado lo podéis encontrar en el fichero `mm-ijk2.c`.

Estudio Previo

1. Dibujad, para cada una de las formas `ijk`, en qué orden se recorren las matrices A , B y C . Por ej. (sólo se incluye el dibujo de la dirección en que se recorre la primera matriz):



2. Suponiendo que cada elemento de las matrices ocupa 4 bytes y que nuestra cache de datos tiene líneas de tamaño 64 bytes calculad, para cada una de las 3 formas ijk, cuántos bloques de memoria se han de mover de Memoria Principal a Memoria Cache para ejecutar completamente el bucle más interno 1 vez. Por ejemplo, en la forma ijk:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

tenéis que calcular cuántos bloques de memoria se han de mover de MP a MC para ejecutar:

recorre toda 1 fila \Rightarrow En un bloque (línea) caben 64 bytes / 4 bytes = 16 elementos \Rightarrow N=256 \Rightarrow 256/16 = 16
recorre toda 1 columna \Rightarrow cada elemento al que accedemos es un **jalo**. Esto es debido a que la matriz está ordenada en memoria por filas.

```
for (k=0; k<N; k++)
    C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

1 fallo (el inicial), debido a que en el bucle interno i y j no cambian

Esto es equivalente a calcular los fallos de cache, suponiendo que la MC es de tamaño infinito y completamente asociativa.

3. Suponiendo que cada elemento de las matrices ocupa 4 bytes, y que el tamaño de página de nuestro sistema es de 4 Kbytes calculad, para cada una de las formas ijk, cuántas páginas de memoria virtual se utilizan al ejecutar completamente el bucle más interno 1 vez (como en el apartado anterior). *Cuando se recorre la matriz por columnas* \Rightarrow $N = 256 \Rightarrow 256^2 \text{ elem.} / 1024 \text{ elem/pag} = 64$
 $N = 512 \Rightarrow 512^2 \text{ elem.} / 1024 \text{ elem/pag} = 256$
 $N = 1024 \Rightarrow 1024^2 \text{ elem.} / 1024 \text{ elem/pag} = 1024$

Trabajo a realizar durante la Práctica

1. Compilad y ejecutad los tres programas para un tamaño N=6. Comprobad que los 3 programas dan el mismo resultado.
2. Rellenad la tabla de la hoja de respuestas indicando para cada forma del producto de matrices cuánto tiempo (segundos) tarda en ejecutarse y a cuántos MFLOPS se ejecuta. Os aconsejamos el uso de una hoja de cálculo para agilizar vuestro trabajo.
3. Teniendo en cuenta lo que habéis hecho en los apartados anteriores y en el trabajo previo, explicad la razón de las diferencias de rendimiento en estos tres programas.
4. Aplicad la optimización adicional a las otras dos aplicaciones que no la tienen. Compilad y ejecutad los tres programas para un tamaño N=6 y comprobad que los 3 programas dan el mismo resultado.
5. Rellenad la tabla de la hoja de respuestas indicando para cada forma del producto de matrices con la optimización adicional, cuánto tiempo (segundos) tarda en ejecutarse y a cuántos MFLOPS se ejecuta. Os aconsejamos el uso de una hoja de cálculo para agilizar vuestro trabajo.
6. Comparad los resultados obtenidos con los obtenidos antes de optimizar los programas, y sacad conclusiones de dicha comparación.

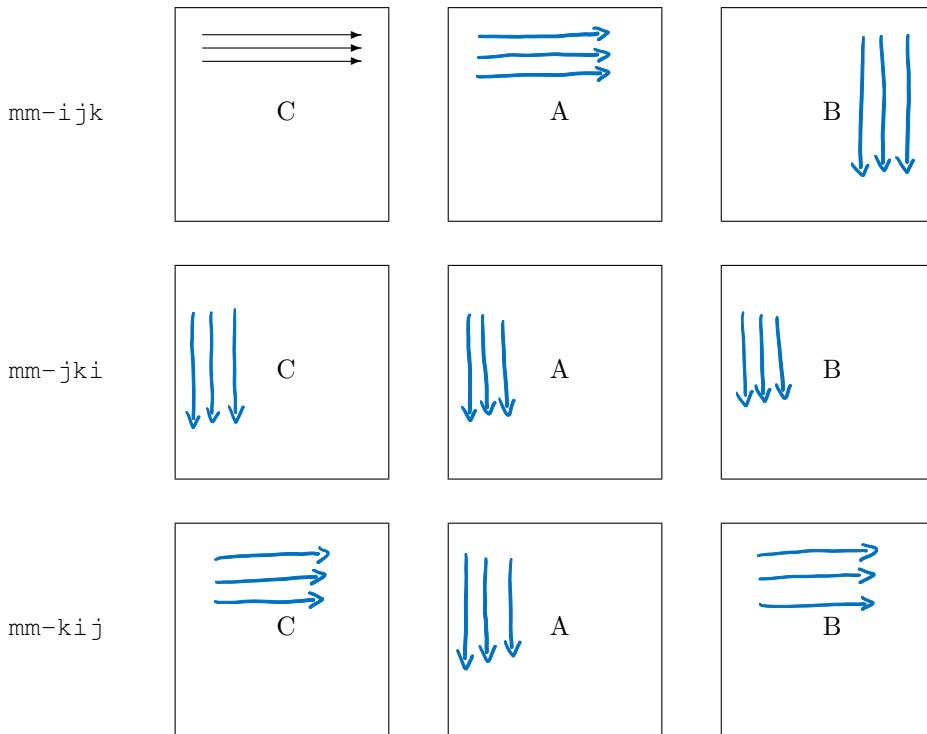
Nombre: _____

Grupo: _____

Nombre: _____

Hoja de respuesta al Estudio Previo

1. Dibujad, para cada una de las formas ijk, en qué orden se recorren las matrices A, B y C:



2. Calculad los fallos de cache, suponiendo que la MC es de tamaño infinito y completamente asociativa. Escribid los resultados en la siguiente tabla:

N	mm-ijk			mm-jki			mm-kij		
	matriz A	matriz B	matriz C	matriz A	matriz B	matriz C	matriz A	matriz B	matriz C
256	16	256	1	256	1	256	1	16	16
512	32	512	1	512	1	512	1	32	32
1024	64	1024	1	1024	1	1024	1	64	64

3. Calculad cuántas páginas de memoria virtual se utilizan al ejecutar completamente el bucle más interno 1 vez. Escribid los resultados en la siguiente tabla:

N	mm-ijk			mm-jki			mm-kij		
	matriz A	matriz B	matriz C	matriz A	matriz B	matriz C	matriz A	matriz B	matriz C
256	1	64	1	64	1	64	1	1	1
512	1	256	1	256	1	256	1	1	1
1024	1	1024	1	1024	1	1024	1	1	1

Nombre: _____

Grupo: _____

Nombre: _____

Hoja de respuestas de la práctica

1. Compilad y ejecutad los tres programas para un tamaño N=6. Comprobad que los 3 programas dan el mismo resultado.
2. Rellenad la siguiente tabla:

N	Tiempo ejecución (en seg.)			MFLOPS		
	mm-ijk	mm-jki	mm-kij	mm-ijk	mm-jki	mm-kij
256	0'049	0'060	0'037	684'78	559'2	906'88
512	0'488	0'521	0'281	550'07	515'23	955'29
1024	3'725	8'383	2'211	576'51	256'17	971'27

$$\text{MFLOPS} = \frac{2 * N^3}{t_{ej} * 10^6}$$

3. Teniendo en cuenta lo que habéis hecho en los apartados anteriores y en el trabajo previo, explicad la razón de las diferencias de rendimiento en estos tres programas.

La diferencia de rendimiento está en la manera en la que se accede al vector, si accedemos por filas el rendimiento es mejor que un acceso por columnas, el cual produce más fallos de cache. Esto es debido a que la matriz está almacenada en memoria en el orden que siguen las filas.

4. Aplicad la optimización adicional a las otras dos aplicaciones. Compilad y ejecutad los tres programas para un tamaño N=6. Comprobad que los 3 programas dan el mismo resultado.

5. Rellenad la siguiente tabla:

N	Tiempo ejecución (en seg.)			MFLOPS		
	mm-ijk	mm-jki	mm-kij	mm-ijk	mm-jki	mm-kij
256	0'039	0'052	0'035	860'37	645'28	958'7
512	0'325	0'481	0'273	825'96	558'08	983'28
1024	2'491	8'218	2'121	862'1	261'31	1012'48

6. Comparad los resultados obtenidos con los obtenidos antes de optimizar los programas, y sacad conclusiones de dicha comparación.

Las optimizaciones hechas dan como resultado menos accesos a cache, por lo que mejoran el tiempo de ejecución.