Práctica 4: Rellenado de arrays, partición en bloques (tiling) y distribución de datos en estructuras

En esta sesión estudiaremos tres nuevas técnicas orientadas a la mejora del rendimiento caché a través de la reestructuración de un código dado.

1. En el campus virtual tenéis un código escrito en C en el que la aplicación de cada técnica es beneficiosa. Aplica la técnica correspondiente sobre cada uno de ellos. El código proporcionado incluye comandos que permiten medir su tiempo de ejecución.
2. Medir el tiempo de ejecución de los códigos. Deberán compararse los tiempos de ejecución del código antes y después de aplicar la optimización. Sugerimos que en las compilaciones se use al menos el flag `-O` o `-O2` para que el compilador aplique algunas optimizaciones básicas.

   Recuerda que como la ejecución con Cachegrind es bastante más lenta que la ejecución normal deberás ajustar en tiempo de compilación los tamaños de los bucles y estructuras de datos a través de la macro `N`, y el número de repeticiones de los bucles que se usan para medir los tiempos de ejecución a través de la macro `NUM_REPS`. Ejemplo: `gcc -g -O3 -DN=100 -DNUM_REPS=1 miejemplo.c`.
3. Ejecutar estos códigos con Cachegrind para comparar la tasa de fallos obtenida antes y después de aplicar la optimización. Encontrar una configuración de la caché (tamaño de la caché, tamaño de bloque/línea y asociatividad) para la que se aprecie la mejora obtenida al aplicar cada técnica.

Finalmente, responder a las preguntas que aparecen en el cuestionario `campus virtual` asociado a esta sesión de prácticas.

1. Rellenado de arrays

Cuando no es posible acceder de forma secuencial a los contenidos de un array, puede darse el caso de que debido a las dimensiones del mismo los accesos sobre él generen fallos de conflicto de forma sistemática. Un ejemplo típico de esta situación es el acceso por columnas a una matriz bidimensional en C (puesto que en C los arrays bidimensionales se almacenan por filas) que tiene un tamaño de fila que es una potencia de dos. En esta situación muchas o incluso todas las líneas que contienen los elementos de cada columna de la matriz recaen en el mismo conjunto de la caché, haciendo imposible su reuso para cuando se va a acceder a la siguiente columna. La Figura 1 ilustra este problema para una caché de correspondencia directa con 4 líneas en la que cada línea puede contener dos de los elementos del array `m[4][8]`. Como puede verse en la ilustración de la izquierda, en esta situación las líneas de memoria que contienen los datos de una columna dada del array están todas mapeadas a la misma línea de la caché, con lo que cada vez que se va acceder a un nuevo elemento de una columna, la línea asociada al elemento precedente debe ser expulsada de la caché para hacerle sitio. De esta forma, cuando se va a acceder a la siguiente columna, es imposible reusar ninguna de las líneas que se trajeron durante el procesamiento de la columna precedente y deben volver a traerse todas de la memoria. Este problema puede resolverse rellenando el array aumentando el tamaño de la fila de la matriz, de forma que las líneas de cada columna se distribuyan entre más conjuntos. En la ilustración de la derecha hemos pasado a declarar `m` como una matriz `m[4][10]` aunque realmente sólo sigamos usando sus 8 primeras columnas. El cometido de las dos columnas de relleno es simplemente que, como puede apreciarse, gracias a ellas ahora cada línea de una columna de la matriz va a parar a una línea diferente de la caché. De esta forma es posible reusar todas cuando se va a proceder a procesar la siguiente columna de `m`. La Figura 2 tiene el código correspondiente.

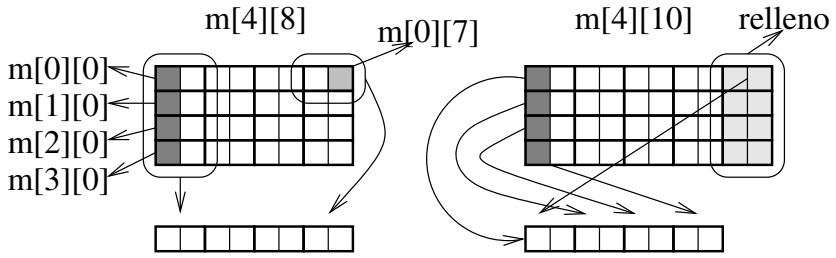

Figura 1: Problemática que resuelve el rellenado de arrays

```
/*Declaracion original de una matriz con una fila
 con un tamaño multiplo o divisor del de la cache */
float m[4][8];

/*bucle que recorre el array por columnas*/

for(j=0; j<8;j++)
  for(i=0; i<4; i++)
    q = q * 1.2 + m[i][j];

/*Declaracion nueva de un array cuyas filas no
  tienen un tamaño multiplo o divisor del de la cache  */
float m[4][10];
```

Figura 2: Ejemplo de rellenado de arrays

2. Partición en bloques (Tiling)

La partición en bloques es una técnica general de reestructuración de un programa para reutilizar bloques de datos que se encuentran en la caché antes de que sean reemplazados. Imaginemos un programa con varios arrays, algunos accedidos por columnas y otros por filas. Almacenando los datos por filas o columnas (según la característica del compilador) no resolvemos nada, pues tanto filas como columnas son utilizadas en cada iteración. Por esta misma razón, el intercambio de bucles tampoco nos sirve en este caso.

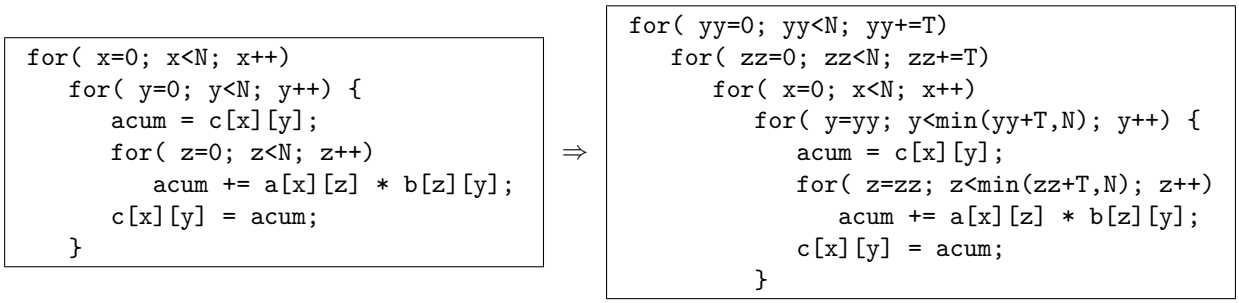

Figura 3: Aplicación de la técnica de partición en bloques al producto de matrices

```
 /* Particion en bloques optimizada*/
for( yy=0; yy<N; yy+=T) {
    lim_y = min(yy+T, N);
    for( zz=0; zz<N; zz+=T) {
        lim_z = min(zz+T, N);
        for( x=0; x<N; x++)
            for( y=yy; y<lim_y; y++) {
                acum = c[x][y];
                for( z=zz; z<lim_z; z++)
                    acum += a[x][z] * b[z][y];
                c[x][y] = acum;
             }
    }
}
```

Figura 4: Código con partición en bloques optimizado

Para asegurar que los elementos que están siendo accedidos están en la caché, el código original debe de ser modificado para trabajar con una sub-matriz de tamaño T×T, de modo que los dos bucles internos realicen los cálculos en pasos de tamaño T en vez de recorrer de principio a fin los arrays A y B. T es el llamado factor de bloqueo. La Figura 3 muestra un código para multiplicar matrices antes y después de aplicarle esta técnica. Como puede apreciarse, la partición en bloques conlleva el aumento del tamaño del código así como del número de cálculos a efectuar. Así, dependiendo del valor de T puede suceder que si bien el número de fallos de caché se reduzca, el tiempo total de ejecución del programa aumente debido a los ciclos de CPU adicionales requeridos. Por ello debe intentarse escribir el código generado por esta técnica de la manera más óptima posible, como muestra la Figura 4.

3. Distribución de datos en estructuras

Las estructuras, y los objetos en el caso de lenguajes orientados a objetos, son ampliamente utilizadas por los programadores al permitir reunir toda la información relativa a un elemento en un mismo contenedor de datos. Desde el punto de vista de la jerarquía de memoria, es importante tener en cuenta que los compiladores de la mayoría de los lenguajes ubican los distintos campos de cada estructura u objeto en la memoria en el orden en que los define el programador. Por otra parte, es posible que algunos campos se usen en muy contadas ocasiones, mientras que otros pueden usarse con mucha frecuencia. De ser éste el caso, sería beneficioso declarar consecutivamente aquellos campos de la estructura que se van a usar conjuntamente para maximizar la localidad espacial. Así, por ejemplo el código a la izquierda de la Figura 5 tiene un bucle que se ejecuta con mucha frecuencia que utiliza los campos `coef` y `v` de la `struct datos`, mientras que los restantes campos se usan menos veces. El acceso es salteado a través de una indirección lo que disminuye aún más la localidad del código. Si al menos definimos los dos campos juntos como podemos ver en la versión del código de la derecha de la figura, hay muchas posibilidades de que con frecuencia ambos vengan en la misma línea de memoria a la caché en vez de en dos diferentes, con el consecuente impacto positivo en el tiempo de ejecución.

```
struct datos {
  float coef, b, c;
  char id[20];
  float posicion[3];
  double v;
};

struct datos ar[N];

for(i=0; i<M; i++)
  r += ar[d[i]].coef * ar[d[i]].v;
```

⇒

```
struct datos {
  double v;
  float coef, b, c;
  char id[20];
  float posicion[3];
};

struct datos ar[N];

for(i=0; i<M; i++)
  r += ar[d[i]].coef * ar[d[i]].v;
```

Figura 5: Código con una estructura en la que optimizamos la ubicación de sus campos para poner adyacentes lo que se usan conjuntamente

```
struct datos1 {
  double v;
  float coef;
};

struct datos2 {
  float b, c;
  char id[20];
  float posicion[3];
};

struct datos1 ar[N];
struct datos2 ar2[N];

for(i=0; i<M; i++)
  r += ar[d[i]].coef * ar[d[i]].v;
```

Figura 6: Código con partición de la estructura en dos: una para los campos usados con frecuencia, y otra para los usados infrecuentemente

Podría obtenerse un grado superior de mejora partiendo la estructura en dos partes. Por un lado tendríamos los campos que se usan mucho, y por otro los que se usan con poca frecuencia. La figura 6 muestra esta disposición. De esta forma, todos los campos que se usan con mucha frecuencia estarían más cercanos y podrían caber los de más estructuras en la caché, ya que se evitarían los huecos de los campos de la estructura que no se emplean en el cálculo frecuente.