Estudio de los efectos de la localidad en la caché

Pablo Souto Soneira, Nicolás Vilela Pérez

Laboratorio de Arquitectura de Computadores
Grupo 4
{pablo.souto.soneira, nicolas.vilela}@rai.usc.es

17 de marzo de 2021

Resumen

Se toma como objeto de estudio los ciclos de reloj necesarios para la ejecución de un programa con unas características determinadas. Este programa se basa en el acceso a los datos de una matriz empleando dos metodologías diferentes: acceso aleatorio y acceso secuencial. El principal factor a estudiar es el efecto que producen los diferentes niveles de memoria caché según se va avanzando por ellos. El estudio general refleja una clara relación entre la caché accedida y la latencia, incrementando según se va accediendo a memorias más lentas.

Palabras clave: caché, ciclos, matriz, jerarquía, acceso a memoria, principio de localidad, latencia

I. Introducción

La memoria caché es un elemento muy importante a la hora de determinar el coste temporal de un programa. Esto es debido a que los accesos a datos a memoria caché son mucho más rápidos que los accesos a memoria principal. En este informe se realiza un estudio sobre cómo varía este coste temporal cambiando diferentes parámetros en el patrón de acceso y el tamaño del conjunto de datos.

La máquina usada para este estudio tiene las siguientes características:

- **Procesador:** Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- **Tamaño caché L1d por CPU:** 32 KiB
- **Tamaño caché L2 por CPU:** 256 KiB
- **Tamaño caché L3:** 12 MiB
- **Tamaño línea caché:** 64 B

En este informe se evaluarán dos tipos de mediciones: la primera consiste en el acceso al primer elemento de cada fila de forma aleatoria y la segunda accede al primer elemento de la matriz y a los siguientes de la fila distanciados entre sí por el número de datos de la matriz que caben en una línea de la caché.

II. Preparación de las mediciones

Para llevar a cabo el estudio ha sido necesaria la creación de un programa en C que utiliza una matriz como medio sobre el que realizar la medición.

Antes de realizar el estudio es necesario establecer los valores de 3 parámetros que son los que marcarán la diferencia entre unas mediciones y otras:

- **C:** se corresponde con el número de columnas de la matriz y podrá tomar los valores 4, 8, 20 y 40.

- **L:** representa el número de líneas caché diferentes a las que se accederá. Este tomará los 8 valores siguientes: 0.5*S1, 1.5*S1, 0.5*S2, 1.5*S2, 0.5*S3, 0.75*S3, 2*S3 y 4*S3, siendo S1 el número de líneas caché que caben en la caché L1 de datos (es importante comprobar que sea la de datos ya que es en esta donde se encuentra la matriz), S2 el número de líneas caché que caben en la caché L2 y S3 lo mismo para la caché L3.

- **F:** se corresponde con el número de filas de la matriz y su valor depende del de **L**, ya que las lecturas de los valores de la matriz tienen que hacerse sobre **L** líneas caché diferentes.

Cabe destacar que la compilación del código requiere de la opción `-O0` para evitar optimizaciones del compilador y no alterar los datos.

III. Resultados

Los valores presentados fueron obtenidos a partir del acceso a la matriz usando dos metodologías de acceso distintas explicadas más adelante. Primero comenzaremos comentando los puntos comunes a ambas mediciones.

Las mediciones en los accesos se repiten un total de 11 veces, una primera para precargar la caché y así evitar datos atípicos en la primera medición, y las 10 restantes son las que se contabilizan. En cada repetición se miden los ciclos de reloj necesarios para la ejecución del código y, en nuestro caso, nos quedaremos con la mediana como dato representativo. Al final obtenemos un total de 32 valores para cada estudio, 8 tamaños de fila por cada valor que toman las columnas. Por último mencionar dos consideraciones a tener en cuenta: es necesario la impresión de los valores obtenidos en los accesos a la matriz para evitar que el compilador los ignore y que, debido a resultados atípicos en las gráficas de accesos, se nos recomendó realiazar la medición de las 10 repeticiones juntas y luego hacer la media para así alargar el tiempo de ejecución y permitir que los accesos se estabilizasen.

En el **primer estudio** la metodología empleada para el acceso a los datos consiste en el acceso aleatorio al primer elemento de cada fila. Esta aleatoriedad en el acceso se consigue mediante la utilización de un array con los índices de las filas barajado (M[ind[i]][0]). El valor de **F** en este primer estudio se igual al resultado de esta operación: `(LINESIZE * L) / (C * sizeof(double))`. Sin embargo, esto no se cumple para **C** igual a 20 y 40 por lo que debemos igualar **F** a **L** para así tener el número de accesos que buscamos y no menos.

En el **segundo estudio** se accede a los elementos de una fila, de forma secuencial, que estén distanciados por el número de doubles que quepan en una línea caché, en nuestro caso caben 8 (64 / 8) por lo que accedemos a M[0][0], M[0][8]... Aquí, el valor que toma **F** se obtiene como resultado de dividir **L** entre el número de accesos a diferentes líneas caché que se pueden hacer en una fila. Para **C** igual a 4 esto no se cumple por lo que debemos de doblar el número de filas, siendo **F** el doble que **L**.

En la gráfica 1 se puede ver como en el tramo en el cual los datos pasan de L1 a L2 [S1 - S2] hay un ligero aumento de los ciclos de reloj debido a los fallos de L2 (inicialmente vacía). La bajada que tiene lugar cuando la L2 se llena [S2 - S3] solamente se puede explicar a través de optimizaciones hardware de caché que tiene el procesador usado para las mediciones, ya que

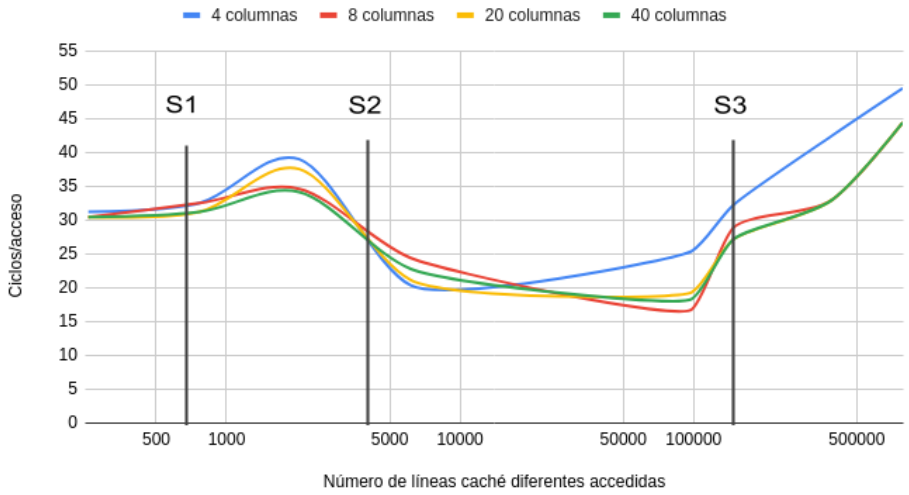

Figura 1: *Estudio 1*

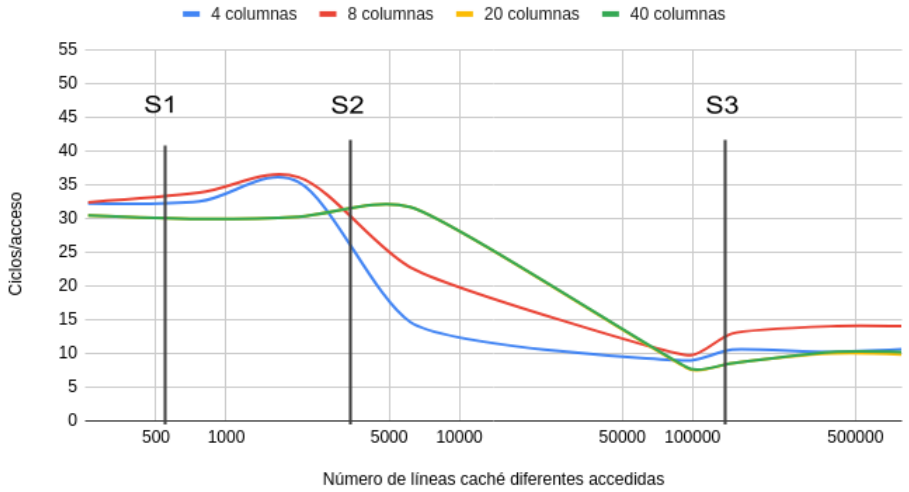

Figura 2: *Estudio 2*

también debería haber un incremento de ciclos debido a los fallos. Por último, al superar el tamaño de L3 [S3 - inf], los accesos se realizarán a memoria principal, que conlleva una mayor lentitud, por eso aumentan considerablemente los ciclos por acceso.

En cuanto a la gráfica 2 podemos observar que, al igual que en la gráfica 1, se produce un leve aumento al llenar la L1 y pasar a la L2, sin embargo, este aumento no se produce en las matrices con 20 y 40 columnas por una posible optimización hardware. La disminución de ciclos en el intervalo [S2 - S3], al igual que en la gráfica anterior, se explica a través de optimizaciones hardware de caché que tiene el procesador usado para las mediciones. Finalmente, en el último tramo los ciclos se mantienen bajos debido al principio de localidad, ya que se está accediendo a datos próximos en la caché debido a que se accede a la matriz con los índices ordenados y no aleatoriamente.

IV. Conclusiones

El ámbito de las memorias es muy investigado debido a que son el factor limitante en la actualidad en cuanto a rendimiento de un computador y las memorias caché por tanto también lo son. Este estudio se ha centrado en como afecta el modo de acceso a los datos en el rendimiento de la ejecución de un programa. Para ello se preparó un código centrado en la medición de los ciclos necesarios para la ejecución del mismo.

Los resultados obtenidos se muestran en 1 y 2 y de ellos podemos extraer una serie de conclusiones:

- El acceso aleatorio en lugar del secuencial influye en gran medida en la latencia del programa ya que, mientras que la gráfica 1 tiene una tendencia ascendente, la 2 es descendente. Esto se debe al principio de locaclidad por el cual si un programa accede a unos datos determinados, es muy probable que acceda más tarde a los adyacentes.

- Los ciclos por acceso van aumentando a medida que se accede a las cachés L2, L3 y a memoria principal o RAM, debido a que la velocidad de acceso a los datos va disminuyendo a medida que nos acercamos a la memoria RAM. Esto se podría ver claramente si no hubiera optimizaciones hardware, pero aún así se puede apreciar en el estudio 1, cuando la L3 se llena y los accesos son a memoria principal, más lenta, incrementando así los ciclos/acceso. En el estudio 2 no se aprecia debido a que la localidad de los datos juega a nuestro favor, disminuyendo los ciclos/acceso.

Por último mencionar que de este estudio puede surgir la idea de realizar este mismo experimento con diferentes tipos de caché, totalmente asociativa, de asignación directa o parcialmente asociativa con diferente número de vías, así como con diferentes tamaños para los distintos niveles.

V. Anexo I: código más relevante

Para la reserva de la memoria de la matriz se usa la función `_mm_malloc()` de la biblioteca `pmmintrin.h` de la seguiente manera:

```
   double **M = _mm_malloc(F * sizeof(double *), LINESIZE);
for (i = 0; i <F; i++)
{ M[i] = malloc(C * sizeof(double)); }
```

siendo `F` y `C` las filas y las columnas de la matriz, respectivamente.

Este trozo de código lo que hace es alinear el inicio de la matriz a una línea caché. La función `_mm_malloc()` tiene dos parámetros: el tamaño de memoria que se quiere reservar (igual que `malloc()`) y el tamaño de la línea caché del procesador que se va usar en la ejecución del código.

Para cada medición que se realiza se accede a la matriz de las dos siguientes formas:

- **Para el estudio 1:**
  ```
  for (j = 0; j <F; j++)
  { sum += M[ind[j]][0]; }
  ```
  siendo `ind[j]` un array de índices de 0 a F-1 barajados anteriormente para dar aleatoriedad a los accesos.

- **Para el estudio 2:**
  ```
  for (i = 0; i <F; i++)
  {
  for (j = 0; j <C; j += LINESIZE / sizeof(double))
        sum += M[i][j];
  }
  ```

Para más información acerca del código usado, revisar el anexo II.

VI. Anexo II: código completo

```
int main() {
    int S1 = ceil( x: (double) L1dSIZE / LINESIZE);
    int S2 = ceil( x: (double) L2SIZE / LINESIZE);
    int S3 = ceil( x: (double) L3SIZE / LINESIZE);

    // Se eligen las columnas de la matriz dependiendo del estudio a realizar
    int C = 8;

    int L = ceil( x: 4 * S3);

    // Se eligen las filas de la matriz dependiendo del estudio a realizar
    int F = ceil( x: (LINESIZE * L) / (C * sizeof(double)));
    //int F = L;
    //int F = ceil((double) L / 3);
    //int F = ceil((double) L / 5);

    double ck, sum;
    int ind[F];
    double red[N], clocks[N];
    int i, j, k;

    srand(getpid());

    // Reserva dinámica de memoria de la matriz
    double **M = _mm_malloc( size: F * sizeof(double *), LINESIZE);
    for (i = 0; i < F; i++) {
        M[i] = malloc( size: C * sizeof(double));
    }

    // Incialización y barajada de los índices
    for (i = 0; i < F; i++)
        ind[i] = i;
    barajar(ind, F);

    // Generación aleatoria de los datos de la matriz
    for (i = 0; i < F; i++)
        for (j = 0; j < C; j++)
            M[i][j] = (double) (rand() % 50) / 53;


    printf( format: "\n");
```

```
// Bucle para realizar las N mediciones
for (k = 0; k < N + 1; k++) {
    sum = 0;
    /*
     * Si k == 0, se realiza la precarga de datos. Este dato será atípico.
     * Por esta razón no se mide la primera iteración.
     */
    if (k != 0)
        start_counter();


    // ESTUDIO 1
    /*for (j = 0; j < F; j++) {
        sum += M[ind[j]][0];
    }*/


    // ESTUDIO 2
    for (i = 0; i < F; i++) {
        for (j = 0; j < C; j += LINESIZE / sizeof(double))
            sum += M[i][j];
    }


    if (k != 0) {
        ck = get_counter();
        red[k - 1] = sum;
        printf( format: "[Rep. %d] Ciclos: %1.10lf\n", k - 1, ck);
        clocks[k - 1] = ck;
    }
}
```

```
// Impresión de los resultados para evitar optimizaciones del compilador
printf( format: "\n");
for (i = 0; i < N; i++)
    printf( format: "red[%d] = %lf\n", i, red[i]);
printf( format: "\n");

/* Esta rutina imprime a frecuencia de reloxo estimada coas rutinas start_counter/get_counter */
mhz( verbose: 1,  sleeptime: 1);

// Cálculo de la mediana
quicksort(clocks,  primerInd: 0,  ultimoInd: N - 1);
printf( format: "\nMediana de ciclos: %lf\n\n", (clocks[4] + clocks[5]) / 2);

/*
// Cálculo de la media
sum = 0;
for (i = 0; i < N; i++) {
    sum+=clocks[i];
}
printf("\nMedia de ciclos: %lf\n\n", sum/N);
*/

// Liberación de la memoria de la matriz
for (i = 0; i < F; i++) {
    free(M[i]);
}
_mm_free(M);

exit(EXIT_SUCCESS);
```