Grado en INGENIERÍA INFORMÁTICA

ARQUITECTURA DE COMPUTADORES

102775

Departamento de Arquitectura de Computadores y Sistemas Operativos (DACSO)

PRÁCTICA nº 2

Medida y Optimización del Rendimiento de la Jerarquía de Memoria

Enunciado y Documentación

Medir Rendimiento de la Jerarquía de Memoria

ENUNCIADO: La práctica realiza un análisis empírico del efecto que tiene la jerarquía de memoria en el rendimiento del computador. En primer lugar se analizarán programas realistas que se ven afectados por la forma en que se accede a los datos de la memoria. Posteriormente se determinará numéricamente la tasa de fallos de los diferentes niveles de caché, y el efecto de estos fallos en el tiempo de ejecución.

OBJETIVOS:

- Evaluar el efecto del acceso a los datos en memoria en el rendimiento de la ejecución de un programa
- Analizar el código ensamblador para calcular el número total de accesos a memoria realizados por un programa.
- Obtener la traza de referencias a datos (direcciones de memoria) generadas por el programa a partir del análisis del código fuente y del código ensamblador.
- Entender las optimizaciones orientadas al aumento de la localidad y su efecto en el rendimiento del programa.
- Analizar la traza de referencias a datos para estimar la tasa de fallos en los diferentes niveles de caché y de TLB. Corroborar la estimación mediante los resultados obtenidos de los contadores hardware del procesador.
- Cuantificar el efecto que tiene el tamaño del problema en el número total de fallos de caché y en el tiempo de ejecución: generar gráficas del IPC medido empíricamente y utilizarlas para razonar sobre el comportamiento del rendimiento del programa.

MATERIAL y PREPARACIÓN PREVIA

Procesador Intel i7 950 (Laboratorio)
Quad core i7, CPU clock: 3.33 GHz; Arquitectura x86-64; Compiladores: gcc versión 4.4.7, 4.9.0; icc versión 16.0.0
Planificación de la ejecución de instrucciones: Dinámica (ejecución *out-of-order*)
dL1 Cache: 32 KiB, L2 Cache: 256 KiB, L3 Cache: 8 MiB,
Cache Line size is always 64 Bytes, Main Memory: 8 GiB
TLB: 512 entries, with 4KB pages

Antes de asistir a la sesión de laboratorio, se debe leer con detenimiento todo este documento, y se deben contestar las preguntas previas a partir de los datos presentados. La parte de trabajo previo debería ser discutida con el profesor al inicio de la sesión. Hay que llevar preparada una planificación del trabajo a realizar durante la sesión (improvisando sobre la marcha suele faltar tiempo al final). Las dudas sobre lo que hay que hacer hay que resolverlas lo antes posible con el profesor de prácticas.

Durante la sesión es importante asegurar que no se están ejecutando procesos que introduzcan "ruido" en la toma de medidas de tiempo. Se puede utilizar el comando **`top`**, que visualiza el uso de la CPU y los procesos en ejecución.

Las respuestas a la práctica se deben entregar por correo electrónico (o en el Campus Virtual) según os indique vuestro profesor de prácticas hasta 48 horas después de finalizada la segunda sesión correspondiente a esta práctica. Cada día adicional a la fecha límite de entrega penalizará un 10% la nota de la práctica (es decir, que entregar más de 10 días tarde supone automáticamente una evaluación de 0 puntos). El documento electrónico enviado debe indicar el nombre completo de los alumnos, el turno de prácticas al que pertenece el grupo, el nombre del profesor de prácticas, y la fecha de entrega.

CONCEPTOS BÁSICOS

Para configurar el uso de los compiladores se debe utilizar el comando **`module [add|rm]`** [**`gcc/4.4.7| gcc/4.9.0`** | **`intel/16.0.0`**]. Las opciones de compilación importantes son –On (nivel de optimización), -g (compilar con información de "debug" para poder visualizar el código fuente con VTune), -S (para generar un archivo con el código ensamblador), -D (para definir constantes en tiempo de compilación), -v (para verificar la versión del compilador). gcc e icc pueden estar instalados y usarse a la vez.

El comando **`perf`** permite medir rendimiento mediante lecturas de los contadores H/W del procesador. La opción más común es **`stat`**, y se pueden especificar medidas específicas con la opción -e <nombre-contador>. Los comandos **`record`** y **`report`** permiten hacer *profiling* del código y visualizar el código ensamblador más ejecutado.

El uso de VTune requiere instalar el entorno del compilador icc, se ejecuta con el comando "**`amplxe-gui &`**", y es conveniente haber compilado con la opción –g para poder ver el código fuente.

Sesión 1: TRABAJO PREVIO

Ejemplo: Transformación Map y Reduce

El código de la siguiente función realiza dos tipos de transformaciones algorítmicas sobre el contenido de tres vectores de entrada de N y M elementos (X[N], V1[M] y V2[M]) para generar un nuevo vector de salida de M elementos (R[M]).

La transformación de tipo Map genera una matriz de N×M valores a partir de una especie de producto cartesiano del vector X[N] con la combinación de los vectores V1[M] y V2[M]. La función elemental que se aplica a los elementos X_i, $V1_j$ y $V2_j$, para generar cada elemento $t_{i,j}$ es la siguiente:

$$\text{MAP function: } \forall\ i, j \ \{\ t_{i,j} = (X_i - V1_j) * (X_i - V2_j)\ \}$$

La transformación de tipo Reduce genera un vector de M valores a partir de N×M valores usando una función de reducción, que consiste en sumar los elementos. En realidad, podemos pensar que cada elemento del vector resultado R[j] se calcula como la reducción (suma) de todos los valores de la columna j de la matriz tmp[][]. La función elemental que se aplica a los elementos $tmp_{i,j}$ y r_j, para reducir los valores es la siguiente:

$$\text{REDUCE function: } \forall\ i, j \ \{\ r_j = tmp_{i,j} + r_j\ \}$$

MapReduce.c: Fragmento del programa principal que realiza la parte significativa del cómputo (más del 99,999%)

```
float *R, *X, *V1, *V2;

initVec(  X, N, … );  // Fill elements of  X[N] with random values
initVec( V1, M, … );  // Fill elements of V1[M] with random values
initVec( V2, M, … );  // Fill elements of V2[M] with random values

// MAP transformation: Generate 2D matrix from Cartesian product X[] x {V1,V2}[]
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    tmp[i*M+j] = (X[i] - V1[j]) * (X[i] - V2[j]); // 3 FLOATs + 3 LOADs + 1 STORE

// REDUCE transformation: Generate 1D vector from 2D matrix
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    R[j] = R[j] + tmp[i*M+j]; // 1 FLOAT + 2 LOADs + 1 STORE
```

En el código anterior se ejecutan 4×M×N operaciones en punto flotante (2 restas, una multiplicación y una suma), 5×M×N operaciones de lectura a memoria (X[i] en el primer bucle se lee solamente una vez), y 2×M×N operaciones de escritura a memoria. El siguiente código es una optimización del anterior que realiza tres transformaciones: (1) fusionar los bucles Map y Reduce, (2) intercambiar los dos bucles anidados, y (3) re-factorizar el cómputo del bucle interno. En cada iteración del bucle interno se ejecuta una sentencia que corresponde con una operación básica que denominaremos operación MAP-REDUCE.

MapRed2.c: Fragmento del programa principal que se ha optimizado

```
// MAP & REDUCE: fusion + loop interchange + Refactor computation
for (j=0; j<M; j++) {
  float V1j= V1[j],     V2j     = V2[j];
  float Rj = V1j*V2j*N, V1plusV2= V1j + V2j;

  for (i=0; i<N; i++)  // Inner loop: executes a MAP-REDUCE basic operation
    Rj = Rj + X[i]*(X[i] - V1plusV2); // 3 FLOAT + 1 LOAD (N×M times)

  R[j] = Rj;
}
```

Pregunta 1a: Explicar el efecto que tiene por separado cada una de las tres optimizaciones anteriores: (1) fusionar los bucles Map y Reduce; (2) intercambiar el orden de los bucles; y (3) transformar las operaciones de cálculo en otras equivalentes.

El código optimizado se compila con icc –O3 y se ejecuta con diferentes valores de N y de M. La siguiente pantalla generada con el comando **`perf`** y con las opciones **`record`** y **`report`**, para $N=10^5$ y $M=10^6$, nos identifica el bucle interno del programa donde se consume más del 99% del tiempo de la ejecución. El color rojo de las instrucciones indica que son las que

más veces estaban a la salida del procesador al producirse la toma de medidas estadísticas. Este hecho es difícil de interpretar, ya que depende del funcionamiento interno del H/W que toma medidas durante la ejecución. Lo más fiable que podemos decir es que la suma del tiempo asignado a todas las instrucciones dentro del bucle interno representa más del 99% del tiempo, y por tanto es este bucle el que nos interesa comprender y optimizar.

```
660:  movaps  (%r15,%rsi,4),%xmm13
      movaps  %xmm13,%xmm12
      subps   %xmm15,%xmm12
      mulps   %xmm13,%xmm12
      addps   %xmm12,%xmm3
      movaps  %xmm13,%xmm12
      subps   %xmm14,%xmm12
      mulps   %xmm12,%xmm13
      movaps  0x10(%r15,%rsi,4),%xmm12
      add     $0x8,%rsi
      cmp     %rdi,%rsi
      addps   %xmm13,%xmm1
      movaps  %xmm12,%xmm13
      subps   %xmm15,%xmm13
      mulps   %xmm12,%xmm13
      addps   %xmm13,%xmm0
      movaps  %xmm12,%xmm13
      subps   %xmm14,%xmm13
      mulps   %xmm13,%xmm12
      addps   %xmm12,%xmm1
      jb      660
```

El código ensamblador siguiente, obtenido compilando con la opción –S, es el mismo que el anterior. Se ha añadido una descripción usando la notación vectorial de Cilk Plus.

MapRed2.s: bucle interno del programa (icc –O3) y código C equivalente con notación vectorial (Cilk Plus)

```
..B1.63:                                  //   do {
   movaps    (%r15,%rsi,4), %xmm13        //  1.  t13[0:4]  =   X[i:4];        LOAD
   movaps    %xmm13, %xmm12               //  2.  t12[0:4]  = t15[0:4];        FMOV
   subps     %xmm15, %xmm12               //  3.  t12[0:4] -= V1plusV2a[0:4];  FAD
   mulps     %xmm13, %xmm12               //  4.  t12[0:4] *= t12[0:4];        FMUL
   addps     %xmm12, %xmm3                //  5.  Rj3[0:4] += t12[0:4];        FAD
   movaps    %xmm13, %xmm12               //  6.  t12[0:4]  = t13[0:4];        FMOV
   subps     %xmm14, %xmm12               //  7.  t12[0:4] -= V1plusV2b[0:4];  FAD
   mulps     %xmm12, %xmm13               //  8.  t13[0:4] *= t12[0:4];        FMUL
   movaps  16(%r15,%rsi,4), %xmm12        //  9.  t12[0:4]  = X[i+4:4];        LOAD
   addq      $8, %rsi                     // 10.  i         =  i+8;            INT
   cmpq      %rdi, %rsi                   // 11.  c         = i < N;           INT
   addps     %xmm13, %xmm1                // 12.  Rj1[0:4] += t13[0:4];       FAD
   movaps    %xmm12, %xmm13               // 13.  t13[0:4]  = t12[0:4];       FMOV
   subps     %xmm15, %xmm13               // 15.  t13[0:4] -= V1plusV2a[0:4]; FAD
   mulps     %xmm12, %xmm13               // 17.  t13[0:4] *= t12[0:4];       FMUL
   addps     %xmm13, %xmm0                // 19.  Rj0[0:4] += t13[0:4];       FAD
   movaps    %xmm12, %xmm13               // 14.  t13[0:4]  = t12[0:4];       FMOV
   subps     %xmm14, %xmm13               // 16.  t13[0:4] -= V1plusV2b[0:4]; FAD
   mulps     %xmm13, %xmm12               // 18.  t12[0:4] *= t13[0:4];       FMUL
   addps     %xmm12, %xmm1                // 20.  Rj1[0:4] += t12[0:4];       FAD
   jb        ..B1.63                      // 21.} while (c);                  BRN
```

Para entender mejor las optimizaciones que ha realizado el compilador al generar el código ensamblador se va a comparar el número de operaciones que se expresan en el programa original en lenguaje C con el número de instrucciones ejecutadas. Usando el comando `perf`, se han tomado medidas del número total de instrucciones ejecutadas para la ejecución completa del programa con diferentes valores de N y M. Estas medidas se muestran en la Tabla 1.

Problem Size	Instructions (ICount)
$N=10^3$, $M=10^8$	139,2 G
$N=10^4$, $M=10^7$	132,1 G
$N=10^5$, $M=10^6$	131,4 G
$N=10^6$, $M=10^5$	131,3 G
$N=10^7$, $M=10^4$	131,5 G
$N=10^8$, $M=10^3$	133,5 G
$N=10^9$, $M=10^2$	152,9 G

Tabla 1. Total de instrucciones ejecutadas en la ejecución completa del programa MapRed2.c (procesador i7 950)

Pregunta 1b: La complejidad del programa MapRed2.c medida en operaciones básicas MAP-REDUCE es de N×M. Contar el nº de instrucciones que se ejecutan en el bucle interno del programa ensamblador MapRed2.s (ver página anterior) y usar los valores empíricos mostrados en la Tabla 1 para deducir el nº de veces que se ejecuta el bucle interno del programa ensamblador. Usar este último resultado para calcular el número de operaciones básicas MAP-REDUCE que se ejecutan en cada iteración del bucle del código ensamblador. Explicar las pequeñas diferencias en el nº de operaciones básicas MAP-REDUCE por iteración calculado empíricamente para diferentes valores de N y M.
Nota: recordar que las medidas de rendimiento se hacen de la ejecución completa de todo el programa.

Una vez calculado el nº de operaciones MAP-REDUCE que se ejecutan en cada iteración del bucle interno del programa ensamblador, analizar otra vez el código ensamblador y contar el nº de operaciones FLOAT y de operaciones LOAD que se hacen en cada iteración. Comparar este número con el valor que corresponde con las operaciones MAP-REDUCE del programa original.

Pregunta 1c: A partir de los resultados anteriores y del análisis del código ensamblador y del código C equivalente (que sirve de ayuda) explicar las optimizaciones que ha realizado el compilador en el código, de la forma más precisa posible.

Usando de nuevo el comando `perf`, se toman las siguientes medidas de rendimiento para la ejecución completa del programa con diferentes valores de N y M:

Problem Size	Cycles (CCount)
$N=10^3$, $M=10^8$	54,8 G
$N=10^4$, $M=10^7$	50,8 G
$N=10^5$, $M=10^6$	51,9 G
$N=10^6$, $M=10^5$	52,0 G
$N=10^7$, $M=10^4$	72,1 G
$N=10^8$, $M=10^3$	85,9 G
$N=10^9$, $M=10^2$	97,2 G

Tabla 2. Tiempo medido en ciclos de reloj de la ejecución completa del programa MapRed2.c (i7 950)

Pregunta 1d: Analizar ahora el comportamiento del rendimiento (tiempo de ejecución del programa) e identificar posibles anomalías que se producen a medida que se aumenta el valor de N y se reduce el valor de M. No es necesario dar una explicación de porqué ocurren todas las anomalías, pero sí identificar cualquier comportamiento que resulte significativo.
Nota: calcular el valor del IPC proporciona información muy útil para realizar este análisis.

Sesión 1: TRABAJO DURANTE la SESIÓN

Se van a analizar los accesos a memoria que hace el programa y su efecto en el rendimiento (tiempo de ejecución) del programa. En primer lugar se debe calcular el tamaño de los vectores utilizados en el programa. A la cantidad total de memoria que necesita el programa para almacenar sus datos se le denomina *memory footprint*. Calcular en la siguiente tabla el tamaño de los vectores (medido en Bytes) en función de los valores de N y de M que se indican.

Utilizar el comando `perf` para realizar medidas de rendimiento de la caché del procesador. Se debe medir la cantidad total de fallos en la caché (*cache-misses*), que en el caso del procesador del laboratorio corresponde con la denominada caché L3 (o de tercer nivel). Suponiendo que el fichero ejecutable MR2 se ha creado compilando el código MapRed2.c con el compilador icc opción –O3, el comando para realizar la ejecución es:

```
$ perf stat -e instructions,cycles,cache-misses ./MR2 …
```

Pregunta 1e: Completar la tabla siguiente a partir del análisis del código y usando los datos obtenidos de las ejecuciones. Explicar los resultados y correlacionarlos con los resultados de rendimientos proporcionados en las Tablas 1 y 2.

Problem Size	Tamaño de V1[], V2[] y R[]	Tamaño de X[]	L3 cache misses
$N=10^3$, $M=10^8$			
$N=10^4$, $M=10^7$			
$N=10^5$, $M=10^6$			
$N=10^6$, $M=10^5$			
$N=10^7$, $M=10^4$			
$N=10^8$, $M=10^3$			
$N=10^9$, $M=10^2$			

Tabla 3. Valores calculados y valores medidos experimentalmente sobre el rendimiento de la memoria

El siguiente código es una modificación del programa que pretende mejorar la forma en que se accede a los datos y así mejorar el tiempo de ejecución del programa. A la estrategia utilizada se le denomina *data blocking* y consiste en dividir una gran estructura de datos en bloques lógicos más pequeños y acceder a los datos bloque a bloque. Analizar el código siguiente para entender la idea de la optimización. En aras a una mayor claridad, se supone que BLK, el tamaño del bloque, es un divisor exacto de N, el tamaño del vector X[]. El valor BLK está definido dentro del programa como una variable que se inicializa en tiempo de ejecución con el valor que el usuario le proporciona al programa en la línea de comandos.

MapRedBLK.c: Fragmento del programa optimizado para mejorar localidad con *data blocking* (BLK divide a N)

```
for (j=0; j<M; j++)
  R[j] = V1[j]*V2[j]*N;  // Hay que inicializar R[] fuera del bucle principal

for (k=0; k<N; k+=BLK) // para cada bloque de BLK elementos de X[]
  for (j=0; j<M; j++) {
    float Rj= R[j], V1plusV2 = V1[j] + V2[j];
    for (i=k; i<k+BLK; i++) // cálculo para el bloque X[k:BLK]
      Rj = Rj + X[i]*(X[i] - V1plusV2);
    R[j] = Rj;
  }
```

A continuación se muestra una porción de la pantalla generada con el comando **`perf`** y las opciones **`record`** y **`report`**, para $N=10^7$, $M=10^4$ y $BLK=10^6$, con el bucle interno del programa donde se consume más del 99% del tiempo de la ejecución. En la página siguiente se muestra también el código ensamblador generado por el compilador icc al utilizar la opción -S.

```
16,92 | 6c0:  movaps (%r9,%r15,4),%xmm6
 0,00 |       movaps 0x10(%r9,%r15,4),%xmm8
 4,90 |       movaps %xmm6,%xmm5
 0,00 |       movaps %xmm8,%xmm7
17,55 |       add    $0x8,%r15
 0,00 |       cmp    %rdi,%r15
      |       subps  %xmm1,%xmm5
 0,00 |       subps  %xmm1,%xmm7
18,05 |       mulps  %xmm5,%xmm6
 0,16 |       mulps  %xmm7,%xmm8
11,72 |       addps  %xmm6,%xmm4
 0,85 |       addps  %xmm8,%xmm3
29,76 |       jb     6c0
```

MapRedBLK.s: bucle interno compilado con icc –O3 y código C equivalente con notación vectorial de Cilk Plus.

```
..B1.87:                                  // do {
  movaps    (%r9,%r15,4), %xmm6           // 1.  t6[0:4]   = X[i:4];          LOAD
  movaps  16(%r9,%r15,4), %xmm8           // 2.  t8[0:4]   = X[i+4:4];        LOAD
  movaps    %xmm6, %xmm5                  // 3.  t5[0:4]   = t6[0:4];         FMOV
  movaps    %xmm8, %xmm7                  // 4.  t7[0:4]   = t8[0:4];         FMOV
  addq      $8, %r15                      // 5.  i         =  i+8;            INT
  cmpq      %rdi, %r15                    // 6.  c         = i < N;           INT
  subps     %xmm1, %xmm5                  // 7.  t5[0:4]  -= V1plusV2[0:4];   FAD
  subps     %xmm1, %xmm7                  // 8.  t7[0:4]  -= V1plusV2[0:4];   FAD
  mulps     %xmm5, %xmm6                  // 9.  t6[0:4]  *= t5[0:4];         FMUL
  mulps     %xmm7, %xmm8                  // 10. t8[0:4]  *= t7[0:4];         FMUL
  addps     %xmm6, %xmm4                  // 11. Rj1[0:4] += t6[0:4];         FAD
  addps     %xmm8, %xmm3                  // 12. Rj2[0:4] += t8[0:4];         FAD
  jb        ..B1.87                       // 13.} while (c);                  BRN
```

Pregunta 1f: Completar la tabla siguiente compilando con icc –O3 y ejecutando el programa anterior para tamaños grandes de N, y con *data blocking* de BLK= 10^4 y 10^6 y usando los datos obtenidos de las ejecuciones. Explicar los resultados y compararlos con los resultados de rendimientos proporcionados en las Tablas 1, 2 y 3.

Problem Size (BLK=10^6)	Elapsed Time	ICount	CCount	IPC	Cache misses
N=10^7, M= 10^4					
N=10^8, M= 10^3					
N=10^9, M= 10^2					
Problem Size (BLK=10^4)	Elapsed Time	ICount	CCount	IPC	Cache misses
N=10^7, M= 10^4					
N=10^8, M= 10^3					
N=10^9, M= 10^2					

Tabla 4. Rendimiento de la ejecución del programa MapRedBLK.c (i7 950)

Pregunta 1g: El reto final de esta sesión es optimizar el código anterior (MapRedBLK.c) que utiliza la estrategia de *data blocking*. Para lograrlo hay que analizar con detenimiento el código ensamblador de las dos versiones anteriores (MapRed2.c y MapRedBLK.c) y entender qué transformación hace el compilador en la primera versión que no hace en la segunda versión.

Ayuda: se puede suponer que M es múltiplo de 2 o de 4 para simplificar la codificación del programa.

Mostrar una tabla con los resultados de rendimiento obtenidos para los mismos valores de N, M y BLK que en la pregunta anterior

Sesión 2: TRABAJO PREVIO

Ejemplo nº 2: Multiplicación de Matrices

Analizamos el algoritmo clásico de multiplicación de matrices y algunas de sus variaciones, para estudiar el comportamiento de los accesos a memoria y para aplicar optimizaciones que mejoren la localidad de los accesos, y por tanto el rendimiento del sistema de memoria y del procesador.

En primer lugar se muestra el código del programa principal. Los dos primeros parámetros de entrada al programa que se reciben en tiempo de ejecución son: (1) el tamaño de las matrices, N, y (2) la versión del código a ejecutar (n | s | t | r | b). Observar el uso que se hace de los parámetros `argc` y `argv`, que reciben la información escrita en la línea de comandos por el usuario. Otra cosa a resaltar es el uso de funciones especiales de reserva dinámica de memoria (solamente para el compilador icc), que sirven para obtener direcciones alineadas a bloques de 64 Bytes. Finalmente, se incluye un código de verificación que calcula valores de *checksum* de la matriz resultado y de las dos primeras filas, para poder comprobar que las diferentes versiones del mismo algoritmo son todas funcionalmente idénticas.

```
int main (int argc, char **argv)
{
  int  N, step;    char o='n', c= ' ';     double *A, *B, *C;

  if (argc>1) { N= atoi(argv[1]); }       // convertir string en un nº entero
  if (argc>2) { o= argv[2][0]; }          // 1º letra del segundo parámetro
  if (argc>2) { c= argv[2][1]; }          // 2ª letra del segundo parámetro
  step= N;                                // valor por defecto de step
  if (argc>3) { step= atoi(argv[3]); }    // convertir string en un nº entero
  if (argc<2 || N<2 || N>10000) {
     printf("argumentos: N(2-10000) [opt] [step]\n");
     return 1;
  }

  // dynamically allocate free memory in addresses aligned to 64 Bytes
  double *A=(double*) __mm_malloc ( sizeof(double)*N*N, 64 );
  double *B=(double*) __mm_malloc ( sizeof(double)*N*N, 64 );
  double *C=(double*) __mm_malloc ( sizeof(double)*N*N, 64 );

  init_mat (A, N, 1);
  init_mat (B, N, 2);

  printf("Matrix Multiply with N= %d, step= %d, and version %c\n", N, step, o);

  switch (o) {
    case 'n':  mult_mat       (A, B, C, N); break;
    case 's':  mult_matSIMD   (A, B, C, N); break;
    case 't':  mat_transpose  (B, N);
               mult_mat_transp(A, B, C, N); break;
    case 'r':  mat_transpose  (B, N); // needed for transposed version
               mult_mat_transp_reg_tile(A, B, C, N); break;
    case 'b':  mat_transpose  (B, N); // needed for transposed version
               mult_mat_trnsp_blocked (A, B, C, N, step); break;
  }

  if (c=='1') {
    printf("CheckSum (matrix)= %e\n", mult_checksum(C, N));
    printf("CheckSum (row 0) = %e\n", vect_checksum(C, N));
    printf("CheckSum (row 1) = %e\n", vect_checksum(C+N, N));
  }

  __mm_free(A);  __mm_free (B);  __mm_free (C);

  return 0;
}
```

A continuación se muestran 2 versiones de la multiplicación de matrices: la versión clásica y la misma versión añadiendo una directiva especial de OpenMP para ayudar al compilador a optimizar el código. En ambas se realiza la multiplicación de 2 matrices de elementos de tipo *double*, donde cada matriz tiene tamaño N×N (es decir, que cada matriz contiene N^2

elementos). Tal como se puede ver en el programa principal, N se define al ejecutar el programa, reservando memoria para estas matrices de forma dinámica. Los vectores se definen de una dimensión, pero se usan como si fueran matrices de 2 dimensiones, accediendo a los elementos usando dos índices (i,j) y expresando de forma explícita la conversión de 2 dimensiones a una dimensión (i*N+j).

OpenMP es una extensión de funciones (API) y sentencias especiales integradas en forma de *pragmas*, disponible para algunos lenguajes de programación (C, C++ y Fortran) que permite ayudar al compilador a vectorizar y paralelizar una aplicación secuencial. En este ejemplo se muestra el uso de directivas para ayudar a la vectorización de bucles. OpenMP es un estándar soportado por muchos compiladores, entre ellos los compiladores gcc e icc. Para activar el uso de las directivas o comandos de OpenMP se usa la opción –fopenmp en el compilador gcc, y la opción –openmp en el compilador icc.

Todas las directivas de OpenMP comienzan por "`#pragma omp`". La cláusula `simd` indica que el bucle que viene a continuación es vectorizable. Esto supone varias cosas: (1) que las iteraciones del bucle son todas independientes entre sí, y (2) que no hay problemas de alias en memoria con las áreas apuntadas por los punteros (*memory aliasing*). La cláusula `reduction(+:S)` indica que la operación `+` que se realiza con la variable `S` es una reducción y que las operaciones se pueden reordenar. Se debe recordar que la aritmética en punto flotante no es completamente asociativa, ya que los resultados tienen pequeños errores de cálculo que pueden provocar pequeñas diferencias al cambiar el orden de las operaciones de suma. Finalmente, la directiva `aligned(a,b,c:64)` informa al compilador de que los punteros indicados están alineados a bloques de tamaño de 64 Bytes. Esto le permite al compilador generar un código más simple al vectorizar el programa.

Mult.c:

```
void mult_mat ( double *a, double *b, double *c, int N ){   // opción 'n'
  int i, j, k;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
    {
      double S=0.0;
      for (k=0; k<N; k++)
        S += a[i*N+k] * b[k*N+j];
      c[i*N+j] = S;
    }
}

void mult_matSIMD ( double *a, double *b, double *c, int N ){     // opción 's'
  int i, j, k;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
    {
      double S=0.0;
#pragma omp simd reduction(+:S) aligned(a,b,c:64)
      for (k=0; k<N; k++)
        S += a[i*N+k] * b[k*N+j];
      c[i*N+j] = S;
    }
}
```

Se utiliza el compilador `icc` y se ejecuta el código para tamaños de problema N=1.000 y 2.000, con las opciones o= 'n' y 's'. Los resultados del rendimiento de la ejecución medidos con la herramienta `perf` se muestran en la Tabla 1. La columna "L3 *Cache Misses*" indica el número absoluto de fallos de la caché de último nivel (*Last Level Cache* o *LLC*), que en el caso del procesador i7 950 del laboratorio es la de tercer nivel (L3). Este valor se obtiene empíricamente y tiene un alto error en la medida, por lo que no puede considerarse como un valor absolutamente preciso (más bien al contrario), sino como un indicador relativo de rendimiento de la memoria caché.

Compilador	Parámetros	Elapsed Time	Instructions	Cycles	L3 Cache Misses
icc –O3 –openmp (v16.0)	1000 n	1,65 s.	5,6 G	5,6 G	0,039 G
	1000 s	1,90 s.	4,2 G	6,4 G	0,039 G
	2000 n	17,6 s.	44,2 G	59,5 G	1,0 G
	2000 s	20,9 s.	33,4 G	70,5 G	1,0 G

Tabla 1. Medidas de Rendimiento (ejecución completa del programa) para dos versiones de multiplicar matrices

A partir de los datos anteriores y para las 4 ejecuciones, calcular el valor de IPC y la eficiencia de codificación del programa, medida como instrucciones ejecutadas por operación. Consideraremos que cada iteración del bucle interno del programa fuente equivale a una operación básica del algoritmo, que consiste en leer dos valores de tipo double, multiplicarlos, y acumular el resultado usando una suma.

Pregunta 2a: Usando las métricas de IPC y eficiencia de codificación del programa, comparar el rendimiento entre las dos versiones del programa y explicar las diferencias sin utilizar ninguna información sobre el código ensamblador generado.

Pregunta 2b: Comparar el rendimiento entre las ejecuciones con diferentes tamaños de problema, N= 1.000 y N= 2.000. Explicar las diferencias solamente con los datos calculados anteriormente.

A continuación se muestra el código ensamblador correspondiente al bucle más interno de la primera versión del código (opción 'n'). Buscar en el sitio web http://x86.renejeschke.de/ la funcionalidad de las instrucciones **mvhpd** y **unpckl** para entender el código y poder contestar a las preguntas que se hacen a continuación (cuidado, porque en esta documentación se usa la nomenclatura de Intel, en la que el primer operando es el destino, mientras que en la captura de pantalla de abajo se usa la nomenclatura de los sistemas Linux, y que venimos usando durante el curso, en la que el último operando es el destino).

Bucle más interno de la función `mult_mat()`, compilado con icc 16.0.0 –O3 –openmp

```
 5,90 |1a63:  movsd   (%r10,%rax,8),%xmm3
 0,05 |       inc     %r15d
      |       movsd   (%rsi,%r14,8),%xmm2
57,94 |       inc     %rax
 4,53 |       movhpd  0x8(%rsi,%r14,8),%xmm2
10,21 |       add     %rbx,%r14
 0,02 |       unpckl  %xmm3,%xmm3
 3,88 |       cmp     %r12d,%r15d
 0,01 |       mulpd   %xmm2,%xmm3
10,73 |       addpd   %xmm3,%xmm1
 6,69 |       jb      1a63
```

Pregunta 2c: A partir del código fuente original y del código ensamblador anterior, calcular la cantidad total de instrucciones LOAD, STORE, FAD y FMUL que se ejecutan en toda la tarea de multiplicar matrices (al ejecutar completamente los tres bucles anidados). Recordad que instrucción no es sinónimo de μoperación.

A continuación se muestra el tamaño en Bytes que ocupan los datos diferentes leídos de a[] y b[] (algo que se define como *memory footprint*) en tres casos: (1) en una ejecución completa del bucle interno del programa (k=0 hasta N) para un cierto valor fijo de **i** y de **j**; (2) en una ejecución completa del bucle intermedio del programa (k=0 hasta N y j=0 hasta N) para un valor fijo de **i**; y (3) en la ejecución completa de los tres bucles. Las escrituras en c[] no las contamos porque son aproximadamente N veces menos frecuentes que las lecturas, y tienen un efecto mínimo en el rendimiento.

mult_mat	Sólo Bucle Interno	Bucle Intermedio	Ejecución Completa
a[]	**8×N Bytes (1 fila)**	**8×N Bytes (1 fila)**	**$8\times N^2$ Bytes (la matriz)**
b[]	**8×N Bytes (1 fila)**	**$8\times N^2$ Bytes (la matriz)**	**$8\times N^2$ Bytes (la matriz)**
Total:	**16N**	**$8N + 8N^2$**	**$16N^2$**

Tabla 2. *Memory footprint* (total Bytes accedidos) en la ejecución de la función mult_mat

El bucle interno lee 16N Bytes diferentes, pero el bucle intermedio, que ejecuta el bucle interno N veces, no lee $16N^2$ Bytes diferentes, sino aproximadamente $8N^2$ datos diferentes. ¿Por qué? Porque en el bucle intermedio se vuelve a leer la misma fila de a[] cada vez (es decir, existe localidad temporal en el acceso a los datos de a[]). Del mismo modo, durante la ejecución completa del programa se leen $16N^2$ datos diferentes para un total de lecturas proporcional a N^3 (revisar la respuesta a la pregunta 2c): porque el bucle externo vuelve a leer la misma matriz b[] cada vez (es decir, existe localidad temporal en el acceso a los datos de b[]).

Pregunta 2d: Con los datos sobre el procesador del laboratorio que aparecen al principio de este documento indicar si el tamaño de las memorias caché L1, L2 y L3 permiten reutilizar los datos de la fila de a[] que se lee en cada iteración del bucle intermedio, y si permiten reutilizar los datos de la matriz b[] que se lee en cada iteración del bucle externo. Usar estos resultados para explicar con más detalle la respuesta a la pregunta 2b.

Matrix Transposition

Tanto el algoritmo como la forma en que los datos se almacenan en memoria determinan el patrón de acceso a los datos y el rendimiento de la memoria. En ocasiones es conveniente modificar el "*layout*" de los datos, incluso aunque este cambio se haga en tiempo de ejecución. En este ejemplo, transponer los datos de la matriz requiere $\Theta(N^2)$ operaciones, mientras que la operación de multiplicar matrices requiere $\Theta(N^3)$ operaciones; por tanto, para valores de N suficientemente grandes la operación de transponer se puede despreciar frente a la operación de multiplicación de matrices. A continuación se muestra el código de la transposición de una matriz y el código de multiplicar matrices, suponiendo que la matriz b[] está transpuesta. Luego se muestran los resultados de rendimiento de la ejecución completa del programa (incluyendo tanto la inicialización de las matrices como la transposición de la matriz b). También se muestra el código ensamblador correspondiente al bucle más interno de la versión transpuesta del código (opción 't'). Observar que la nueva función **`mult_mat_transp`** se diferencia de la implementación anterior únicamente en el orden en que se accede a los elementos de la matriz b[]. En las pruebas que se han hecho no se encuentra diferencia de rendimiento entre la versión que añade directivas de vectorización de OpenMP y la versión que no incluye ninguna directiva: se ha preferido dejar la directiva, porque en versiones futuras del compilador el resultado puede mejorar.

```
void mat_transpose (double *M, int N ) {
  int j, k;
  for (k=0; k<N; k++)
    for (j=k+1; j<N; j++)  {
      double T = M[k*N+j];
      M[k*N+j]= M[j*N+k];
      M[j*N+k]= T;
}  }

void mult_mat_transp ( double *a, double *b, double *c, int N ){  // opción 't'
  int i, j, k;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
    {
      double S=0.0;
#pragma omp simd reduction(+:S) aligned(a,b,c:64)
      for (k=0; k<N; k++)
        S += a[i*N+k] * b[j*N+k];
      c[i*N+j] = S;
    }
}
```

Compilador	Parámetros	Elapsed Time	Instructions	Cycles	L3 Cache Misses
`icc -O3 -openmp (v16.0)`	**`1000 t`**	**`0,58 s.`**	**`1,95 G`**	**`1,96 G`**	**`0,005 G`**
	`2000 t`	**`5,75 s.`**	**`15,3 G`**	**`19,4 G`**	**`0,122 G`**

Tabla 3. Medidas de Rendimiento (ejecución completa del programa) para mult_mat_transp()

Bucle más interno de la función **`mult_mat_transp()`**, compilado con icc 16.0.0 –O3 –openmp

```
 4,34 15f0:  movaps (%rdx,%r11,8),%xmm5
 0,35        movaps 0x10(%rdx,%r11,8),%xmm6
 0,04        movaps 0x20(%rdx,%r11,8),%xmm7
 0,05        movaps 0x30(%rdx,%r11,8),%xmm8
 4,45        mulpd  (%r10,%r11,8),%xmm5
55,38        mulpd  0x10(%r10,%r11,8),%xmm6
 8,13        mulpd  0x20(%r10,%r11,8),%xmm7
 6,92        mulpd  0x30(%r10,%r11,8),%xmm8
10,11        addpd  %xmm5,%xmm4
 0,42        addpd  %xmm6,%xmm3
 1,02        addpd  %xmm7,%xmm2
 1,53        addpd  %xmm8,%xmm1
 6,70        add    $0x8,%r11
 0,13        cmp    %r13,%r11
 0,02        jb     15f0
```

Pregunta 2e: Hacer los mismos cálculos que en las preguntas 2a-2d (incluyendo los de la Tabla 2), presentarlos, y utilizarlos para explicar la mejora de rendimiento. Analizar también el efecto de cambiar el tamaño del problema de 1.000 a 2.000, y compararlo con el efecto que tenía en las versiones anteriores.

Sesión 2: TRABAJO DURANTE la SESIÓN

Register Tiling

La propuesta de transponer la matriz b[], en el apartado anterior, no solamente ha tenido un efecto importante en el rendimiento de la ejecución, sino que además facilita la realización de nuevas optimizaciones (algo habitual en la práctica). A continuación se presenta una optimización del algoritmo de multiplicación de matrices que se denomina *register tiling*, y que aprovecha la localidad temporal del algoritmo para hacer un mejor uso de los registros y así reducir el nº total de accesos a la memoria de datos (y el nº total de fallos de caché). La idea consiste en usar un cierto número de variables locales para mantener valores que son leídos una única vez de memoria, son mantenidos en un registro del procesador, y son usados (reutilizados) más de una vez (en general, estas estrategias buscan maximizar la reutilización de datos o *data reuse*). La estrategia se ha aplicado con *tiles* (baldosas) de 2×2, pero se podría aplicar a *tiles* de mayor tamaño (teniendo en cuenta que el nº total de registros del procesador determina el límite a partir del cual la optimización deja de ser efectiva). Observar que se trata de desenrollar simultáneamente los dos bucles más externos. Para simplificar la codificación, supondremos que N es siempre múltiplo de 2.

```
void mult_mat_trnsp_reg_tiling (double *a, double *b, double *c, int N ) { // op. 'r'
  int i, j, k;
  for (i=0; i<N; i+=2)
    for (j=0; j<N; j+=2) {
      double c00,c01,c10,c11;
      c00= c01= c10= c11 = 0.0;
#pragma omp simd reduction(+:c00,c01,c10,c11) aligned(a,b,c:64)
      for (k=0; k<N; k++) {
        double a0, a1, b0, b1;
        a0 = a[i*N+k];   a1 = a[(i+1)*N+k];
        b0 = b[j*N+k];   b1 = b[(j+1)*N+k];
        c00 += a0*b0;    c01 += a0*b1;
        c10 += a1*b0;    c11 += a1*b1;
      }
      c[i*N+j]    = c00;  c[i*N+j+1]     = c01;
      c[(i+1)*N+j]= c10;  c[(i+1)*N+j+1] = c11;
}  }
```

Parámetros	Elapsed Time	Instructions	Cycles	L3 Cache Misses
1000 r				
2000 r				

Tabla 4. Medidas de Rendimiento (ejecución completa del programa) para versión con *register tiling*

Pregunta 2f: Evaluar y explicar la mejora de la optimización "*register tiling*". Para ello hay que compilar y ejecutar el programa para rellenar la Tabla 4, calcular el IPC y la eficiencia de codificación del programa, calcular el nuevo *memory footprint* del programa y la reutilización de datos que se hace en caché, y también la cantidad total de instrucciones LOAD y STORE que se ejecutan en el programa (usando el código ensamblador que se muestra a continuación).

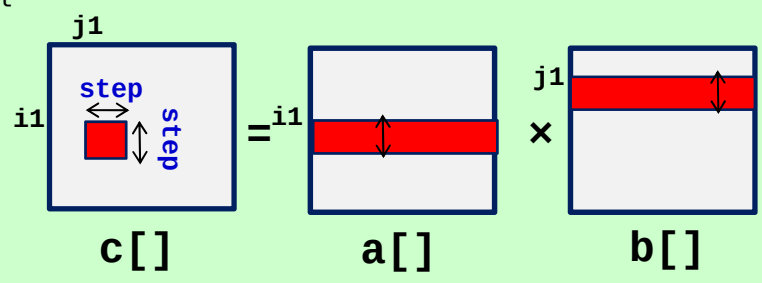
```
 7,98 | e90:  movaps (%rdi,%r15,8),%xmm6
 0,80 |       movaps (%rcx,%r15,8),%xmm8
 0,69 |       movaps %xmm6,%xmm5
 0,15 |       movaps (%r14,%r15,8),%xmm7
30,30 |       movaps (%rsi,%r15,8),%xmm9
31,71 |       add    $0x2,%r15
 0,16 |       mulpd  %xmm7,%xmm5
 2,74 |       mulpd  %xmm9,%xmm6
10,33 |       mulpd  %xmm8,%xmm7
 0,86 |       mulpd  %xmm8,%xmm9
 0,95 |       addpd  %xmm5,%xmm0
 2,02 |       addpd  %xmm6,%xmm4
 8,01 |       addpd  %xmm7,%xmm3
 1,00 |       addpd  %xmm9,%xmm2
 1,78 |       cmp    %r12,%r15
 0,02 |       jb     e90
```

Data Blocking

Finalmente, se presenta una última optimización denominada *data blocking*, que de nuevo aprovecha la localidad temporal del algoritmo para reducir el nº total de fallos en caché. La idea es similar a la estrategia de *register tiling*, pero usando bloques de tamaño mucho más grande, y sin usar variables locales (es decir, registros) para reutilizar datos, sino confiando en la jerarquía de memorias caché para que la mayor parte de los accesos se puedan proporcionar desde una memoria caché "cercana" al núcleo de cómputo y se filtren (eviten) accesos a memorias caché más lentas, o a memoria principal.

A continuación se muestra el esquema del programa que usa como parámetro la variable **`step`**, que determina el tamaño del bloque. En esta implementación simple, el valor de **`step`** debe ser un divisor exacto de `N` y un valor múltiplo de 2.

```
void mult_mat_trnsp_blocked ( ... ) {
  int i, j, k, i1, j1;
  for (i1=0; i1<N; i1+=step)
   for (j1=0; j1< N; j1+=step)
    for (i=i1; i<i1+step; i+=2)
     for (j=j1; j<j1+step; j+=2)
#pragma omp simd reduction(+:) ...
       for (k=0; k<N; k++)
        ... // register tiling
}}}}
```



Pregunta 2g: Ejecutar el programa anterior para N=2000, utilizando la opción 'b' y pasando un tercer parámetro, step, en tiempo de ejecución, con los valores 200, 250, 400, 500 y 1000. Encontrar el valor de step que proporciona mejor rendimiento. La diferencia puede ser pequeña. ¿Qué ventajas y desventajas tiene un valor step más grande?

Pregunta 2h: Evaluar y explicar la mejora de la optimización "*data blocking*". Para ello hay que calcular el IPC y la eficiencia de codificación del programa con el mejor valor de step, calcular el nuevo *memory footprint* del programa y la reutilización de datos que se hace en caché, y también la cantidad total de instrucciones LOAD y STORE que se ejecutan en el programa (usando el código ensamblador que se debe visualizar con alguno de los métodos que hemos enseñado).

Para todos aquellos con espíritu de ingeniero riguroso y metódico (incluso con alma de científico), os proponemos que ejecutéis el programa con las opciones n1, s1, t1, r1 y b1, para verificar que los resultados de todas las implementaciones son idénticos. Es un paso muy importante que todo ingeniero de rendimiento bien formado debe realizar cada vez que comprueba si una modificación del código ha resultado en una mejora del rendimiento (algo que todos los alumnos que se plantean dedicar tiempo al "*challenge*" deberían saber).