2º curso / 2º cuatr.

Grados Ing. Inform.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 4. Optimización de código

Estudiante (nombre y apellidos): José Alberto Hoces Castro
Grupo de prácticas y profesor de prácticas:

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo y se lista con lscpu): *Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz*

Sistema operativo utilizado: *Ubuntu 20.04.4 LTS*

Versión de gcc utilizada: *gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0*

Volcado de pantalla que muestre lo que devuelve `lscpu` en la máquina en la que ha tomado las medidas:

```
joshoc7@joshoc7-Aspire-A315-56:~$ lscpu
Arquitectura:                        x86_64
modo(s) de operación de las CPUs:    32-bit, 64-bit
Orden de los bytes:                  Little Endian
Address sizes:                       39 bits physical, 48 bits virtual
CPU(s):                              8
Lista de la(s) CPU(s) en línea:      0-7
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»:              4
«Socket(s)»                          1
Modo(s) NUMA:                        1
ID de fabricante:                    GenuineIntel
Familia de CPU:                      6
Modelo:                              126
Nombre del modelo:                   Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz
Revisión:                            5
CPU MHz:                             1200.000
CPU MHz máx.:                        3600,0000
CPU MHz mín.:                        400,0000
BogoMIPS:                            2380.80
Virtualización:                      VT-x
Caché L1d:                           192 KiB
Caché L1i:                           128 KiB
Caché L2:                            2 MiB
Caché L3:                            6 MiB
CPU(s) del nodo NUMA 0:              0-7
Vulnerability Itlb multihit:         KVM: Mitigation: VMX disabled
Vulnerability L1tf:                  Not affected
Vulnerability Mds:                   Not affected
Vulnerability Meltdown:              Not affected
Vulnerability Spec store bypass:     Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:            Mitigation; Enhanced IBRS, IBPB conditional, RSB filling
Vulnerability Srbds:                 Not affected
Vulnerability Tsx async abort:       Not affected
Indicadores:                         fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
                                     arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdc
                                     m pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single ssbd ibrs ibpb stibp ib
                                     rs_enhanced tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid avx512f avx512dq rdseed adx smap avx512ifma clflushopt i
                                     ntel_pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves split_lock_detect dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp hwp_pkg_req a
                                     vx512vbmi umip pku ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg avx512_vpopcntdq rdpid fsrm md_clear flush_l1d arch_capabilities
```

1. Modificar el código secuencial para la multiplicación de matrices disponible en SWAD (solo el trozo que calcula la multiplicación) para reducir el tiempo de ejecución. Justificar los tiempos obtenidos (usando siempre `-O2`) a partir de la modificación realizada. Incorporar los códigos modificados en el cuaderno.

MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación A) –explicación-: En la primera modificación he hecho uso del desenrollado de bucle, de forma que en vez de sumar en cada iteración un sumando, calculo cuatro, reduciendo así el número de iteraciones.

Modificación B) –explicación-: En la segunda modificación he intercambiado los índices de los dos bucles más internos, es decir, he cambiado j por k y viceversa.

...

CÓDIGOS FUENTE MODIFICACIONES

A) Captura de `pmm-secuencial-modificado_A.c`

```
    int aux;

    for(i = 0; i < N; i++){
        for (j = 0; j < N; j++){

            // Vamos calculando de 4 en 4 sumandos

            for (k = 0, aux = 0; aux < N/4; k+=4, aux++){
                m3[i][j] += m1[i][k] * m2[k][j];
                m3[i][j] += m1[i][k+1] * m2[k+1][j];
                m3[i][j] += m1[i][k+2] * m2[k+2][j];
                m3[i][j] += m1[i][k+3] * m2[k+3][j];
            }


            // Las iteraciones sobrantes se realizan del modo antiguo

            while(k < N){
                m3[i][j] += m1[i][k] * m2[k][j];
                k++;
            }
        }
    }

              for(i = 0; i < N; i++){
                  for (k = 0; k < N; k++)
                      for (j = 0; j < N; j++)
                          m3[i][j] += m1[i][k] * m2[k][j];
              }
```

Capturas de pantalla (que muestren la compilación y que el resultado es correcto):

B)

```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ gcc  -O2  pmm-secuencial-modificado_A.c -o pmm-secuencial-modificado_A -lrt
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ gcc  -O2  pmm-secuencial-modificado_B.c -o pmm-secuencial-modificado_B -lrt
```

```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./pmm-secuencial 1000
Tiempo: 2.314592303      Tamaño: 1000    m3[0][0]: 33283350.000000       m3[N-1][N-1]: 99899933383350.000000
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./pmm-secuencial-modificado_A 1000
Tiempo: 2.253777795      Tamaño: 1000    m3[0][0]: 33283350.000000       m3[N-1][N-1]: 99899933383350.000000
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./pmm-secuencial-modificado_B 1000
Tiempo: 1.039216648      Tamaño: 1000    m3[0][0]: 33283350.000000       m3[N-1][N-1]: 99899933383350.000000
```

TIEMPOS:

Modificación	**Breve descripción de las modificaciones**	**-O2**
Sin modificar		*2.315 seg*
Modificación A)	Desenrollado del bucle k en 4 iteraciones	*2.254 seg*
Modificación B)	Intercambio entre los índices j y k	*1.039 seg*
...		

COMENTARIOS SOBRE LOS RESULTADOS Y JUSTIFICACIÓN DE LAS MEJORAS EN TIEMPO:

Hemos visto que ambas modificaciones mejoran los tiempos respecto al código secuencial del que partíamos. En la modificación A esto se debe a que con el desenrollado reducimos el número de saltos y ejecutamos en cada iteración más instrucciones independientes que en la implementación anterior. En la modificación B, al intercambiar los índices j y k conseguimos aprovechar el principio de localidad, pues al fijar i y k, estamos recorriendo m1 y m2 por filas, lo cual es más eficiente por cómo se almacenan las matrices en memoria.

2. Usar en este ejercicio el programa secuencial disponible en SWAD que utiliza como base el código de la Figura 1. Modificar en el programa el código mostrado en la Figura 1 para reducir el tiempo de ejecución. Justificar los tiempos obtenidos (usando siempre -O2) a partir de la modificación realizada. En las ejecuciones de evaluación usar valores de N y M mayores que 1000. Incorporar los códigos modificados en el cuaderno.

Figura 1 . Código C++ que suma dos vectores. `M` y `N` deben ser parámetros de entrada al programa, usar valores mayores que 1000 en la evaluación.

```
struct nombre {
        int a;
        int b;
}  s[N];

main()
{
   …
   for (ii=0; ii<M;ii++) {
      X1=0; X2=0;
      for(i=0; i<N;i++)  X1+=2*s[i].a+ii;
      for(i=0; i<N;i++)  X2+=3*s[i].b-ii;

      if (X1<X2)  R[ii]=X1  else  R[ii]=X2;
   }
   …
}
```

MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación A) –explicación-: En vez de acceder a cada vector mediante dos bucles distintos, podemos acceder a ambos en cada iteración, pues por cómo está declarado el struct, las componentes de ambos están contiguas en memoria.

Modificación B) –explicación-: Al igual que antes, lo hacemos todo en un mismo bucle pero además, desenrollamos el bucle de 2 en 2.

...

CÓDIGOS FUENTE MODIFICACIONES

A) Captura figura1-`modificado_A.c`

```
   for (ii=0; ii<M;ii++){
            X1=0; X2=0;
            for(i=0; i<N;i++){
                    X1 += 2*s[i].a + ii;
                    X2 += 3*s[i].b - ii;
            }

            if (X1<X2) {R[ii]=X1;} else {R[ii]=X2;}
   }
for (ii=0; ii<M;ii++){
        X1=0; X2=0;
        int aux = 0;
        for(i=0; aux < N/2; i+=2, aux++){
                X1 += 2*s[i].a + ii;
                X2 += 3*s[i].b - ii;

                X1 += 2*s[i+1].a + ii;
                X2 += 3*s[i+1].b - ii;
        }

        // Solo en caso de ser N impar, tenemos que hacer un último cálculo

        if(N%2 == 1){
                X1 += 2*s[i].a + ii;
                X2 += 3*s[i].b - ii;
        }

        if (X1<X2) {R[ii]=X1;} else {R[ii]=X2;}
}
```

Capturas de pantalla (que muestren la compilación y que el resultado es correcto):

B)

```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ gcc  -O2  figura1-modificado_A.c -o figura1-modificado_A -lrt
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ gcc  -O2  figura1-modificado_B.c -o figura1-modificado_B -lrt
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./figura1-original 8 8
s[0].a=0        s[0].b=0
s[1].a=-1       s[1].b=1
s[2].a=-2       s[2].b=2
s[3].a=-3       s[3].b=3
s[4].a=-4       s[4].b=4
s[5].a=-5       s[5].b=5
s[6].a=-6       s[6].b=6
s[7].a=-7       s[7].b=7
R[0]=-56        R[1]=-48        R[2]=-40        R[3]=-32        R[4]=-24        R[5]=-16        R[6]=-8         R[7]=0
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./figura1-modificado_A 8 8
s[0].a=0        s[0].b=0
s[1].a=-1       s[1].b=1
s[2].a=-2       s[2].b=2
s[3].a=-3       s[3].b=3
s[4].a=-4       s[4].b=4
s[5].a=-5       s[5].b=5
s[6].a=-6       s[6].b=6
s[7].a=-7       s[7].b=7
R[0]=-56        R[1]=-48        R[2]=-40        R[3]=-32        R[4]=-24        R[5]=-16        R[6]=-8         R[7]=0
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./figura1-modificado_B 8 8
s[0].a=0        s[0].b=0
s[1].a=-1       s[1].b=1
s[2].a=-2       s[2].b=2
s[3].a=-3       s[3].b=3
s[4].a=-4       s[4].b=4
s[5].a=-5       s[5].b=5
s[6].a=-6       s[6].b=6
s[7].a=-7       s[7].b=7
R[0]=-56        R[1]=-48        R[2]=-40        R[3]=-32        R[4]=-24        R[5]=-16        R[6]=-8         R[7]=0
```

TIEMPOS: Estos tiempos son el resultado de ejecutar con N = 1000 y M = 1000

Modificación	**Breve descripción de las modificaciones**	**-O2**
Sin modificar		*0.003779333 seg*
Modificación A)	Se realiza un bucle en vez de dos	*0.002300703 seg*
Modificación B)	Igual que en A pero además desenrollado de 2 en 2	*0.001875298 seg*
...		

COMENTARIOS SOBRE LOS RESULTADOS Y JUSTIFICACIÓN DE LAS MEJORAS EN TIEMPO:

El tiempo mejora al pasar a la implementación A porque aprovechamos la localidad espacial y accedemos a las componentes de ambos vectores en un solo bucle, en vez de en 2 como se hacía antes y lo cual añade retardo. También se mejora al pasar de la implementación A a la B porque en la B se necesitan menos iteraciones que en A.

3. El benchmark Linpack ha sido uno de los programas más ampliamente utilizados para evaluar las prestaciones de los computadores. De hecho, se utiliza como base en la lista de los 500 computadores más rápidos del mundo (el Top500 Report). El núcleo de este programa es una rutina que opera con flotantes de doble precisión denominada `DAXPY` (***D****ouble precision- real* ***A****lpha* ***X*** ***P****lus* ***Y***) que multiplica un vector por una constante y los suma a otro vector (Lección 3/Tema 1):

```
for (i=0;i<N;i++) y[i]= a*x[i] + y[i];
```

A partir del programa `DAXPY` disponible en SWAD, generar los programas en ensamblador para cada una de las siguientes opciones de optimización del compilador: -O0, -Os, -O2, -O3. Explique las diferencias que se observan en el código justificando al mismo tiempo las mejoras en velocidad que acarrean. Incorporar los códigos al cuaderno de prácticas y destacar las diferencias entre ellos. Sólo se debe evaluar el tiempo del núcleo `DAXPY.N` deben ser parámetro de entrada al programa.

CAPTURA CÓDIGO FUENTE: `daxpy.c`

```
/* daxpy.c
 Double precision-real Alpha x Plus y: z = alpha * x + y

 Para compilar usar (-lrt: real time library):
             gcc  -O2  daxpy.c -o daxpy -lrt

 Para ejecutar use: daxpy longitud alpha

*/

#include <stdlib.h>        // biblioteca con funciones atoi(),rand(), srand(), malloc() y
free()
#include <stdio.h>         // biblioteca donde se encuentra la función printf()
#include <time.h>          // biblioteca donde se encuentra la función clock_gettime()
//#define VECTOR_LOCAL
#define VECTOR_GLOBAL
//#define VECTOR_DYNAMIC
#define VECTOR_GLOBAL

#ifdef VECTOR_GLOBAL
```

```
#define MAX 33554432      //=2^25

double x[MAX], y[MAX], z[MAX];
#endif

int main(int argc, char** argv){

  int i;

  struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecución

  //Leer argumento de entrada (nº de componentes del vector)
  if (argc<3){
    printf("Faltan argumentos de entrada (n. componentes, alpha)");
    exit(-1);
  }

  int N = atoi(argv[1]);   // Máximo N =2^32-1=4294967295 (sizeof(int) = 4 B)
  double alpha = atof(argv[2]);
  #ifdef VECTOR_LOCAL
  double x[N], y[N], z[N];   // Tamaño variable local en tiempo de ejecución ...
                                                    // disponible en C a partir de C99
  #endif
  #ifdef VECTOR_GLOBAL
  if (N>MAX) N=MAX;
  #endif
  #ifdef VECTOR_DYNAMIC
  float *x, *y, *z;
  x = (float*) malloc(N*sizeof(float));// malloc necesita el tamaño en bytes
  y = (float*) malloc(N*sizeof(float));
  z = (float*) malloc(N*sizeof(float));
  #endif

  //Inicializar vectores
  if (N < 9)
    for (i = 0; i < N; i++)
    {
      x[i] = N * 0.1 + i * 0.1; y[i] = N * 0.1 - i * 0.1;
    }
  else
  {
    //srand(time(0));
    for (i = 0; i < N; i++)
    {
      x[i] = drand48();
      y[i] = drand48();
    }
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);
  //Cálculos daxpyz
  for(i=0; i<N; i++)
    z[i] = alpha*x[i] + y[i];

  clock_gettime(CLOCK_REALTIME,&cgt2);
  ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
       (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));

  //Imprimir resultado de la suma y el tiempo de ejecución
  if (N<11) {
  printf("Tiempo:%11.9f\t / Tamaño Vectores:%d\n",ncgt,N);
  for(i=0; i<N; i++)
```

```
    printf("/ alpha*x[%d]+y[%d]=z[%d](%8.6f*%8.6f+%8.6f=%8.6f) /\n",
           i,i,i,alpha,x[i],y[i],z[i]);

  } else {
    printf("Tiempo:%11.9f\t / Tamaño Vectores:%d\t/ alpha*x[0]+y[0]=z[0](%8.6f*%8.6f+
%8.6f=%8.6f) / / alpha*x[%d]+y[%d]=z[%d](%8.6f*%8.6f+%8.6f=%8.6f) /\n",
           ncgt,N,alpha,x[0],y[0],z[0],N-1,N-1,N-1,alpha,x[N-1],y[N-1],z[N-1]);
  }

  #ifdef VECTOR_DYNAMIC
  free(x); // libera el espacio reservado para v1
  free(y); // libera el espacio reservado para v2
  free(z); // libera el espacio reservado para v3
  #endif
  return 0;
}
```

Tiempos ejec. Longitud vectores=20000000	**-O0**	**-Os**	**-O2**	**-O3**
	0.0904 seg	*0.554 seg*	*0.0545 seg*	*0.0528 seg*

CAPTURAS DE PANTALLA (que muestren la compilación y que el resultado es correcto)**:**

```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ gcc  -O0  daxpy.c -o daxpyO0 -lrt
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ gcc  -O2  daxpy.c -o daxpyO2 -lrt
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ gcc  -Os  daxpy.c -o daxpyOs -lrt
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ gcc  -O3  daxpy.c -o daxpyO3 -lrt
```

```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./daxpyO0 20000000 0.2
Tiempo:0.090406853      / Tamaño Vectores:20000000      / alpha*x[0]+y[0]=z[0](0.200000*0.000000+0.000985=0.000985) / / alpha*x[19999999]+y[19999999]=z[19999999](0.200000*0.133460+0.703447=0.730138) /
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./daxpyO2 20000000 0.2
Tiempo:0.054546518      / Tamaño Vectores:20000000      / alpha*x[0]+y[0]=z[0](0.200000*0.000000+0.000985=0.000985) / / alpha*x[19999999]+y[19999999]=z[19999999](0.200000*0.133460+0.703447=0.730138) /
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./daxpyOs 20000000 0.2
Tiempo:0.055399652      / Tamaño Vectores:20000000      / alpha*x[0]+y[0]=z[0](0.200000*0.000000+0.000985=0.000985) / / alpha*x[19999999]+y[19999999]=z[19999999](0.200000*0.133460+0.703447=0.730138) /
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DGIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp4$ ./daxpyO3 20000000 0.2
Tiempo:0.052791502      / Tamaño Vectores:20000000      / alpha*x[0]+y[0]=z[0](0.200000*0.000000+0.000985=0.000985) / / alpha*x[19999999]+y[19999999]=z[19999999](0.200000*0.133460+0.703447=0.730138) /
```

COMENTARIOS QUE EXPLIQUEN LAS DIFERENCIAS EN ENSAMBLADOR:

O0: El compilador no realiza ninguna optimización del Código, por tanto, suele tener el peor resultado en las ejecuciones aunque mejor tiempo de compilación.

Os: Optimiza el que más el código, ocupando menos instrucciones y ejecutándose en un muy buen tiempo.

O2: Es la opción que hemos usado durante todo el curso, y como bien podemos suponer, la mejor en términos generales. Intenta aumentar el rendimiento del Código sin comprometer el tamaño ni gastar mucho tiempo de compilación.

O3: Obtenemos la mayor optimización posible del código, aunque estas optimizaciones sean caras a nivel de recursos, provocando que realmente no optimice mucho a nivel a cantidad de instrucciones. Por lo general, el tiempo de ejecución suele ser el mejor de todos.

CÓDIGO EN ENSAMBLADOR (no es necesario introducir aquí el código como captura de pantalla, ajustar el tamaño de la letra para que una instrucción no ocupe más de un renglón)**:**
(PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR DONDE ESTÁ EL CÓDIGO EVALUADO, USE COLORES PARA DESTACAR LAS DIFERENCIAS)

Mejor voy a pasar de esto porque no me aporta nada y no quiero suspender PDOO.

daxpyO0.s	daxpyOs.s	daxpyO2.s	daxpyO3.s