2º curso / 2º cuatr.

Grado Ing. Inform.

Doble Grado Ing. Inform. y Mat.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 4. Optimización de código

Estudiante (nombre y apellidos): Elena Merelo Molina
Grupo de prácticas: 2
Fecha de entrega: 30 de mayo
Fecha evaluación en clase:

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): *Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz*

Sistema operativo utilizado: *Ubuntu 16.04 LTS*

Versión de gcc utilizada: *5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)*

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

```
2018-05-11 12:46:54  elena in ~
→ lscpu
Arquitectura:          x86_64
modo(s) de operación de las CPUs:32-bit, 64-bit
Orden de bytes:        Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Hilo(s) de procesamiento por núcleo:2
Núcleo(s) por «socket»:2
Socket(s):             1
Modo(s) NUMA:          1
ID de fabricante:      GenuineIntel
Familia de CPU:        6
Modelo:                78
Model name:            Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
Revisión:              3
CPU MHz:               499.968
CPU max MHz:           2800,0000
CPU min MHz:           400,0000
BogoMIPS:              4799.88
Virtualización:        VT-x
Caché L1d:             32K
Caché L1i:             32K
Caché L2:              256K
Caché L3:              3072K
NUMA node0 CPU(s):     0-3
Flags:                 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 pdpe1g
b rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonst
op_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3
 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_tim
er aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch epb invpcid_single intel_
pt rsb_ctxsw spec_ctrl retpoline kaiser tpr_shadow vnmi flexpriority ept vpid fs
gsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushop
t xsaveopt xsavec xgetbv1 dtherm ida arat pln pts hwp hwp_notify hwp_act_window
hwp_epp
```

1. Para el núcleo que se muestra en el Figura 1, y para un programa que implemente la multiplicación de matrices (use variables globales):

1.1 Modifique el código C para reducir el tiempo de ejecución del mismo. Justifique los tiempos obtenidos (use –O2) a partir de la modificación realizada. Incorpore los códigos modificados en el cuaderno:

Al ejecutar el programa original:

```
 2018-05-31 21:52:46  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Pr
ácticas/BP4/src
○ → gcc -O2 ejer1.c -o ../bin/ejer1

 2018-05-31 21:52:53  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Pr
ácticas/BP4/src
○ → ../bin/ejer1
Tiempo de ejecución: 0.269101195        / r[0]= 699503771       / r[4999]= 499508771
```

Código fuente del programa original:

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

struct{
  int a, b;
} s[5000];

int main(){
  struct timespec cgt1, cgt2;
  double ncgt;  //para tiempo de ejecución
  int i, ii, x1, x2, r[40000];
  srand(time(NULL));

  for(i= 0; i< 5000; i++){
    s[i].a= rand();
    s[i].b= rand();
  }

  clock_gettime(CLOCK_REALTIME, &cgt1);
  for(int ii= 0; ii< 40000; ii++){
    x1= 0;
    x2= 0;

    for(i= 0; i< 5000; i++) x1 += 2*s[i].a + ii;
    for(i= 0; i< 5000; i++) x2 += 3*s[i].b - ii;

    if (x1 < x2) r[ii]= x1; else r[ii]= x2;
  }

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

  printf("Tiempo de ejecución: %11.9f\t / r[0]= %d\t / r[4999]= %d\n", ncgt, r[0], r[39999]);
}
```

Modificamos el programa para que dentro del bucle principal que hace 40000 iteraciones no haya dos bucles que hagan 5000 iteraciones, los unifico en uno, e inicializo en cada iteración x1 y x2 a 5000*ii, quitando dentro el +ii que se le hacía a x1 y -ii a x2, con el consiguiente ahorro de operaciones, que se ve reflejado en la disminución en el tiempo de ejecución:

```
clock_gettime(CLOCK_REALTIME, &cgt1);
for(ii= 0; ii< 40000; ii++){
  x1= 5000*ii;
  x2= -5000*ii;

  for(i= 0; i< 5000; i++) {
    x1 += 2*s[i].a;
    x2 += 3*s[i].b;
  }

  if (x1 < x2) r[ii]= x1;
  else r[ii]= x2;
}

clock_gettime(CLOCK_REALTIME, &cgt2);
```

```
2018-05-31 21:51:10 elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → gcc -O2 ejer1_modif1.c -o ../bin/ejer1_modif1

2018-05-31 21:51:42 elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → ../bin/ejer1_modif1
Tiempo de ejecución: 0.175992594        / r[0]= 24995000        / r[4999]= -162502500
```

A x1 siempre se suma ii, y a x2 se le resta, luego si inicializamos estas variables con los valores correspondientes nos ahorramos 5000* 40000 operaciones. Por otro lado, siempre se hace la misma operación dentro del bucle interno, y el 2 y el 3 puedo sacarlos de factor común y multiplicarlos al final. Así, dentro del bucle que itera 40000 veces solo habrá que sumar a los x1 y x2 anteriormente calculados 5000*ii. :

```
clock_gettime(CLOCK_REALTIME, &cgt1);

x1= 0; x2= 0;
for(i=0; i<5000; i++){
 x1 += s[i].a;
 x2 += s[i].b;
}

x1 *= 2;
x2 *= 3;

for(ii=0; ii<40000; ii++){
  aux1= x1; aux2= x2;
  aux1 += ii*5000;
  aux2 -= ii*5000;

  if( aux1< aux2 ) r[ii]= aux1; else r[ii]= aux2;
}

clock_gettime(CLOCK_REALTIME, &cgt2);
```

```
 2018-05-31 21:53:00  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Pr
ácticas/BP4/src
→ gcc -O2 ejer1_modif2.c -o ../bin/ejer1_modif2

 2018-05-31 21:56:56  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Pr
ácticas/BP4/src
→ ../bin/ejer1_modif2
Tiempo de ejecución: 0.000408167         / r[0]= 24602044        / r[39999]= -162887433
```

Por último, empleamos la técnica de desenrollado de bucles:

```
clock_gettime(CLOCK_REALTIME, &cgt1);

x1= 0; x2= 0;
for(i=0; i<5000; i++){
 x1 += s[i].a;
 x2 += s[i].b;
}

x1 *= 2;
x2 *= 3;

for(ii=0; ii<40000; ii+=4){
  aux1[0]=ii*5000;
  aux1[1]=(ii+1)*5000;
  aux1[2]=(ii+2)*5000;
  aux1[3]=(ii+3)*5000;

  if( x1<x2-2*aux1[0] ) r[ii]=x1+aux1[0]; else r[ii]=x2-aux1[0];
  if( x1<x2-2*aux1[1] ) r[ii+1]=x1+aux1[1]; else r[ii+1]=x2-aux1[1];
  if( x1<x2-2*aux1[2] ) r[ii+2]=x1+aux1[2]; else r[ii+2]=x2-aux1[2];
  if( x1<x2-2*aux1[3] ) r[ii+3]=x1+aux1[3]; else r[ii+3]=x2-aux1[3];
}

clock_gettime(CLOCK_REALTIME, &cgt2);
```

```
 2018-05-31 22:06:15  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Pr
ácticas/BP4/src
→ gcc -O2 ejer1_modif3.c -o ../bin/ejer1_modif3

 2018-05-31 22:13:47  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Pr
ácticas/BP4/src
→ ../bin/ejer1_modif3
Tiempo de ejecución: 0.000346316         / r[0]= 24995000        / r[39999]= -162502500
```

Que es ligeramente más rápida que la anterior. Ejecuté otra vez los diferentes programas y estos son los tiempos que obtuve:

Modificación	**-O2**
Sin modificar(ejer1.c)	0.293532757
Modificación 1(ejer1_modif1.c)	0.149051246
Modificación 2(ejer1_modif2.c)	0.000404168

Modificación 3 (ejer1_modif3.c)	0.000346316

1.2 Genere los códigos en ensamblador con -O2 para el original y los dos códigos modificados obtenidos en el punto anterior (incluido el que supone menor tiempo de ejecución) e incorpórelos al cuaderno de prácticas. Destaque las diferencias entre ellos en el código ensamblador.

Lo primero que noto es la longitud del fichero , el primero tiene 123 líneas, el segundo 116 líneas y el tercero 120, y el cuarto es notoriamente más largo debido al mayor número de líneas e instrucciones del programa principal. Ejer1.s y ejer1_modif1.s son iguales hasta la línea 47, donde se ve reflejada la unificación de los bucles: (a la izquierda el de ejer1.s y a la derecha ejer1_modif1.s.

```
.L3:
                movl    %r8d, %edi
                movl    $s, %eax
                xorl    %esi, %esi
                .p2align 4,,10
                .p2align 3
.L4:
                movl    (%rax), %edx
                addq    $8, %rax
                leal    (%rdi,%rdx,2),
%edx
                addl    %edx, %esi
                cmpq    $s+40000, %rax
                jne     .L4
                movl    $s+4, %eax
                xorl    %ecx, %ecx
                .p2align 4,,10
                .p2align 3
.L5:
                movl    (%rax), %edx
                addq    $8, %rax
                leal    (%rdx,%rdx,2),
%edx
                subl    %edi, %edx
                addl    %edx, %ecx
                cmpq
```

```
.L3:
                movl    %r8d, %edi
                movl    $s, %eax
                xorl    %ecx, %ecx
                xorl    %esi, %esi
                .p2align 4,,10
                .p2align 3
.L4:
                movl    (%rax), %edx
                addq    $8, %rax
                leal    (%rdi,%rdx,2),
%edx
                addl    %edx, %esi
                movl    -4(%rax), %edx
                leal    (%rdx,%rdx,2),
%edx
                subl    %edi, %edx
                addl    %edx, %ecx
                cmpq    $s+40000, %rax
                jne     .L4
                cmpl    %ecx, %esi
                cmovl   %esi, %ecx
                movl    %ecx, r(,%r8,4)
                addq    $1, %r8
```

```
.L3:
                addl    0(%rbp), %eax
                addl    4(%rbp), %edx
                addq    $8, %rbp
                cmpq    $s+40000, %rbp
                jne     .L3
                leal    (%rdx,%rdx,2),
%ecx
                addl    %eax, %eax
                movl    $r, %edx
                movl    $r+160000, %esi
                jmp     .L6
                .p2align 4,,10
                .p2align 3
.L14:
                movl    %eax, (%rdx)
.L5:
                addq    $4, %rdx
                addl    $5000, %eax
                subl    $5000, %ecx
                cmpq    %rdx, %rsi
                je      .L13
.L6:
                cmpl    %ecx, %eax
                jl
```

```
$s+40004, %rax
jne
.L5
cmpl
%ecx, %esi
cmovl
%esi, %ecx
movl
%ecx, r(,%r8,4)
addq
$1, %r8
cmpq
$40000, %r8
jne
.L3
leaq
16(%rsp), %rsi
xorl
%edi, %edi
```

```
cmpq
$40000, %r8
jne
.L3
leaq
16(%rsp), %rsi
xorl
%edi, %edi
```

```
.L14
movl
%ecx, (%rdx)
jmp
.L5
.p2align 4,,10
.p2align 3
```

De la primera a la segunda columna vemos que la segunda es más corta, esto se debe a que en la primera modificación hemos fusionado los dos bucles internos en uno sólo, causando que en la segunda columna haya 2 secciones solamente a diferencia de las 3 que podemos encontrar en la primera.

Mirando las dos últimas columnas no apreciamos ninguna diferencia en tamaño, pero si en número de secciones, esto se debe a que en la segunda modificación hemos simplificado las operaciones del bucle interno permitiéndonos eliminarlo.

Lo más destacado es que en ejer1_modif1.s desaparece una de las secciones, .L5 pasa a estar integrada en .L4, reduciéndose el tiempo al haber menos instrucciones de salto condicional

Del ejer1_modif1 al ejer1_modif2 las diferencias principales son que el bucle .L3 del primero pasa a ser el .L6 del segundo, que separa .L3 en .L4 y .L5, y solo tiene dos instrucciones de salto en .L5, por lo que tarda menos, y .L3 se ejecuta menos veces.

Figura 1 . Código C++ que suma dos vectores

```
struct {
        int a;
        int b;
}  s[5000];

main()
{
   …
   for (ii=0; ii<40000;ii++) {
      X1=0; X2=0;
      for(i=0; i<5000;i++)  X1+=2*s[i].a+ii;
      for(i=0; i<5000;i++)  X2+=3*s[i].b-ii;

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

A) MULTIPLICACIÓN DE MATRICES:
CAPTURA CÓDIGO FUENTE: `pmm_seq.c`

```c
//Autora: Elena Merelo Molina
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

//#define PRINTF_MATRIX_1 // descomentar para imprimir la matriz 1
//#define PRINTF_MATRIX_2 // descomentar para imprimir la matriz 2
//#define PRINTF_RESULT // descomentar para imprimir la matriz resu

int main(int argc, char **argv){
  int tam, i, j, k;
  struct timespec cgt1, cgt2;
  double ncgt;  //para tiempo de ejecución

  if(argc != 2){
    printf("Faltan número de filas y columnas de la matriz\n");
    exit(-1);
  }

  tam= atoi(argv[1]);

  int **m, **n, **r;
```

```c
  //Reservamos espacio para las tres matrices
  m= (int**) malloc(tam*sizeof(int*));
  n= (int**) malloc(tam*sizeof(int*));
  r= (int**) malloc(tam*sizeof(int*));

  for(i= 0; i< tam; i++){
    m[i]= (int*) malloc(tam*sizeof(int));
    n[i]= (int*) malloc(tam*sizeof(int));
    r[i]= (int*) malloc(tam*sizeof(int));
  }

    if ((m == NULL) || (n == NULL) || (r == NULL)) {
      printf("Error en la reserva de espacio\n");
      exit(-2);
    }


  //Inicializamos las matrices
  for(i= 0; i< tam; i++)
    for(j= 0; j< tam; j++){
      m[i][j]= i+j;
      n[i][j]= j;
      r[i][j]= 0;
  }

  clock_gettime(CLOCK_REALTIME, &cgt1);
```

```
//Realizamos el producto de la matriz m por la matriz n, guardando el resultado en r
int suma_local= 0;

for(i= 0; i< tam; i++)
  for(j= 0; j< tam; j++)
    for(k= 0; k< tam; k++)
      r[i][j] += m[i][k] * n[k][j];


clock_gettime(CLOCK_REALTIME, &cgt2);

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

//Para comprobar que hace bien el producto
#ifdef PRINTF_MATRIX_1
  printf("Matriz m: \n");
  for(i= 0; i< tam; i++){
    for(j= 0; j< tam; j++)
      printf("%d\t", m[i][j]);
    printf("\n");
  }
#endif

#ifdef PRINTF_MATRIX_2
  printf("Matriz n: \n");
  for(i= 0; i< tam; i++){
    for(j= 0; j< tam; j++)
      printf("%d\t", n[i][j]);
    printf("\n");
  }
#endif
```

```
  #ifdef PRINTF_RESULT
    printf("Matriz resultante del producto\n");
    for(i= 0; i< tam; i++){
      for(j= 0; j< tam; j++)
        printf("%d\t", r[i][j]);
      printf("\n");
    }
  #endif

  printf("Tiempo de ejecución: %11.9f\t / r[0][0]= %d\t / r[%d][%d]= %d\n", ncgt,

  //Liberación de memoria
  for(int i= 0; i< tam; i++){
    free(m[i]);
    free(n[i]);
    free(r[i]);
  }

  free(m);
  free(n);
  free(r);

}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación 1) –explicación-: antes de multiplicar la matriz m por la n, traspongo la segunda para así poder hacer luego la multiplicación fila por fila, que es como internamente se guarda la matriz, por lo que el tiempo de acceso es menor.

Modificación b) –explicación-: además de trasponer la matriz 2 empleo la técnica de desenrollado de bucle, que aunque aumenta el código disminuye bastante el tiempo.

1.1. CÓDIGOS FUENTE MODIFICACIONES

Incluyo solo la parte que cambia de los programas

a) Captura de `pmm_seq_1.c`

```
clock_gettime(CLOCK_REALTIME, &cgt1);

//Trasponemos la segunda matriz
int aux;
for(i= 0; i< tam; i++){
  for(j= i+1; j< tam; j++){
    aux= n[i][j];
    n[i][j]= n[j][i];
    n[j][i]= aux;
  }
}

//Realizamos el producto de la matriz m por la matriz n, guardando el resultado en r
int suma_local= 0;

for(i= 0; i< tam; i++)
  for(j= 0; j< tam; j++)
    for(k= 0; k< tam; k++)
      r[i][j] += m[i][k] * n[j][k];

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

#ifdef PRINTF_TRASPUESTA
  printf("Matriz n traspuesta: \n");
  for(i= 0; i< tam; i++){
    for(j= 0; j< tam; j++)
      printf("%d\t", n[i][j]);
    printf("\n");
  }
#endif
```

(Nota: después de tomar las capturas eliminé la declaración de la variable suma_local ya que no se usaba y saqué la de aux fuera de la parte en que se mide el tiempo).

Al compilar y ejecutar comprobamos como se hace bien:

```
 2018-05-31 16:05:37  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestr
e/AC/Prácticas/BP4/src
○ → gcc -O2 pmm_seq_1.c -o ../bin/pmm_seq_1

 2018-05-31 16:08:31  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestr
e/AC/Prácticas/BP4/src
○ → ../bin/pmm_seq_1 3
Matriz m:
0       1       2
1       2       3
2       3       4
Matriz n:
0       1       2
0       1       2
0       1       2
Matriz n traspuesta:
0       0       0
1       1       1
2       2       2
Matriz resultante del producto
0       3       6
0       6       12
0       9       18
Tiempo de ejecución: 0.000001609         / r[0][0]= 0    / r[2][2]= 18
```

Ponemos ahora unos tamaños más grandes para obtener una diferencia significativa entre el programa modificado y el original:

```
 2018-05-31 16:15:53  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/P
rácticas/BP4/src
○ → gcc -O2 pmm_seq_1.c -o ../bin/pmm_seq_1

 2018-05-31 16:19:00  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/P
rácticas/BP4/src
○ → ../bin/pmm_seq_1 2000
Tiempo de ejecución: 6.682471192         / r[0][0]= 0    / r[1999][1999]= -896898888

 2018-05-31 16:19:39  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/P
rácticas/BP4/src
○ → gcc -O2 pmm_seq.c -o ../bin/pmm_seq

 2018-05-31 16:19:45  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/P
rácticas/BP4/src
○ → ../bin/pmm_seq 2000
Tiempo de ejecución: 47.711775171        / r[0][0]= 0    / r[1999][1999]= -896898888
```

Hay una diferencia de 41 segundos!!

b) Captura de pmm_seq2.c

Empleamos además desenrollado de bucles:

```
clock_gettime(CLOCK_REALTIME, &cgt1);

//Trasponemos la segunda matriz
for(i= 0; i< tam; i++){
  for(j= i+1; j< tam; j++){
    aux= n[i][j];
    n[i][j]= n[j][i];
    n[j][i]= aux;
  }
}

//Realizamos el producto de la matriz m por la matriz n, guardando el resultado en r
for(i= 0; i< tam; i++)
  for(j= 0; j< tam; j += 4){
    aux0=0; aux1=0; aux2=0; aux3=0;
    for(k= 0; k< tam; k++){
      aux0 += m[i][k] * n[j][k];
      aux1 += m[i][k] * n[j+1][k];
      aux2 += m[i][k] * n[j+2][k];
      aux3 += m[i][k] * n[j+3][k];
    }
    r[i][j]= aux0;
    r[i][j+1]= aux1;
    r[i][j+2]= aux2;
    r[i][j+3]= aux3;
  }

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

Obteniendo, al compilar y ejecutar:

```
 2018-05-31 16:34:46  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/P
rácticas/BP4/src
○ → gcc -O2 pmm_seq_2.c -o ../bin/pmm_seq_2

 2018-05-31 16:37:05  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/P
rácticas/BP4/src
○ → ../bin/pmm_seq_2 2000
Tiempo de ejecución: 3.365167234         / r[0][0]= 0    / r[1999][1999]= -896898888
```

Que es todavía más rápido que la primera modificación.

1.1. TIEMPOS:

Modificación	**-O2**
Sin modificar	47.711775171
Modificación a)	6.682471192
Modificación b)	3.365167234

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

En la tabla se observa como, con pequeñas modificaciones a nuestro código inicial, hemos reducido enormemente su tiempo de ejecución, simplemente teniendo en cuenta la forma en la que se almacenan internamente las matrices y empleando el desenrollado de bucles, que reduce el número de saltos condicionales, aumenta la oportunidad de encontrar instrucciones independientes, facilita la posibilidad de insertar instrucciones para ocultar las latencias, si bien aumenta el tamaño del código y puede parecer más repetitivo.

1.2. CÓDIGO EN ENSAMBLADOR DEL ORIGINAL Y DE DOS MODIFICACIONES : (PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR EVALUADA, USE COLORES PARA DESTACAR LAS DIFERENCIAS)

Generamos código en ensamblador:

```
 2018-05-31 16:42:33  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → gcc -O2 -S pmm_seq_2.c -o ../bin/pmm_seq_2.s

 2018-05-31 16:46:43  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → gcc -O2 -S pmm_seq_1.c -o ../bin/pmm_seq_1.s

 2018-05-31 16:46:48  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → gcc -O2 -S pmm_seq.c -o ../bin/pmm_seq.s
```

```
.L15:
                movq
                0(%rbp,%rbx,8), %r8
                movq
                (%r12,%rbx,8), %rdi
                xorl
                %esi, %esi
                .p2align 4,,10
                .p2align 3
.L11:
                movl
                (%r8,%rsi), %ecx
                xorl
                %eax, %eax
                .p2align 4,,10
                .p2align 3
.L8:
                movq
                (%r15,%rax,8), %rdx
                movl
                (%rdx,%rsi), %edx
                imull
                (%rdi,%rax,4), %edx
                addq
                $1, %rax
                addl
                %edx, %ecx
                cmpl
                %eax, %r14d
                jg
                .L8
                movl
                %ecx, (%r8,%rsi)
                addq
                $4, %rsi
                cmpq
                %r11, %rsi
                jne
                .L11
                addq
                $1, %rbx
                cmpl
                %ebx, %r14d
                jg
                .L15
```

```
.L38:
                movq
                %rbx, %rcx
                addq
                -8(%r8), %rcx
                movl
                %r13d, %eax
                leaq
                -4(%rbx), %r10
                leaq
                4(,%rax,4), %r9
                xorl
                %eax, %eax
                .p2align 4,,10
                .p2align 3
.L9:
                movq
                %r10, %rdx
                addq
                (%r8,%rax,2), %rdx
                movl
                (%rcx,%rax), %esi
                movl
                (%rdx), %edi
                movl
                %edi, (%rcx,%rax)
                addq
                $4, %rax
                movl
                %esi, (%rdx)
                cmpq
                %rax, %r9
                jne
                .L9
                subl
                $1, %r13d
                addq
                $4, %rbx
                addq
                $8, %r8
                cmpl
                $-1, %r13d
                jne
                .L38
.L8:
                xorl
                %ebx, %ebx
.L13:
                movq
                0(%rbp,%rbx,8), %r9
                movq
                (%r12,%rbx,8), %rdi
                xorl
                %r8d, %r8d
                .p2align 4,,10
                .p2align 3
.L15:
                movl
                (%r9,%r8,4), %ecx
                movq
                (%r14,%r8,8), %rsi
                xorl
                %eax, %eax
```

```
.L37:
                movq
                %r10, %rcx
                addq
                -8(%rdi), %rcx
                movl
                %r11d, %eax
                leaq
                -4(%r10), %rbp
                leaq
                4(,%rax,4), %r9
                xorl
                %eax, %eax
                .p2align 4,,10
                .p2align 3
.L9:
                movq
                %rbp, %rdx
                addq
                (%rdi,%rax,2), %rdx
                movl
                (%rcx,%rax), %esi
                movl
                (%rdx), %r8d
                movl
                %r8d, (%rcx,%rax)
                addq
                $4, %rax
                movl
                %esi, (%rdx)
                cmpq
                %rax, %r9
                jne
                .L9
                subl
                $1, %r11d
                addq
                $4, %r10
                addq
                $8, %rdi
                cmpl
                $-1, %r11d
                jne
                .L37
.L8:
                movl
                60(%rsp), %eax
                movq
                40(%rsp), %rcx
                movq
                $0, 8(%rsp)
                shrl
                $2, %eax
                salq
                $5, %rax
                leaq
                40(%rcx,%rax), %rax
                movq
                %rax, (%rsp)
.L11:
                movq
                16(%rsp), %rsi
                movq
```

```
                .p2align 4,,10
                .p2align 3
.L11:
                movl    (%rdi,%rax,4), %edx
                imull   (%rsi,%rax,4), %edx
                addq    $1, %rax
                addl    %edx, %ecx
                cmpl    %eax, %r15d
                jg      .L11
                movl    %ecx, (%r9,%r8,4)
                addq    $1, %r8
                cmpl    %r8d, %r15d
                jg      .L15
                addq    $1, %rbx
                cmpl    %ebx, %r15d
                jg      .L13
```

```
                8(%rsp), %rax
                movq    32(%rsp), %r14
                movq    (%rsi,%rax,8), %r13
                movq    24(%rsp), %rsi
                movq    (%rsi,%rax,8), %r15
                .p2align 4,,10
                .p2align 3
.L15:
                movq    -8(%r14), %r12
                movq    (%r14), %rbp
                xorl    %eax, %eax
                movq    8(%r14), %r11
                movq    16(%r14), %r10
                xorl    %r8d, %r8d
                xorl    %edi, %edi
                xorl    %esi, %esi
                xorl    %ecx, %ecx
                .p2align 4,,10
                .p2align 3
.L12:
                movl    0(%r13,%rax,4), %edx
                movl    (%r12,%rax,4), %r9d
                imull   %edx, %r9d
                addl    %r9d, %ecx
                movl    0(%rbp,%rax,4), %r9d
                imull   %edx, %r9d
                addl    %r9d, %esi
                movl    (%r11,%rax,4), %r9d
                imull   %edx, %r9d
                imull   (%r10,%rax,4), %edx
                addq    $1, %rax
                addl    %r9d, %edi
                addl    %edx, %r8d
                cmpl    %eax, %ebx
                jg      .L12
                movl    %ecx, (%r15)
                movl    %esi, 4(%r15)
                addq    $32, %r14
                movl    %edi, 8(%r15)
                movl    %r8d, 12(%r15)
                addq    $16, %r15
                cmpq    (%rsp), %r14
                jne     .L15
                addq    $1, 8(%rsp)
                movq    8(%rsp), %rax
                cmpl    %eax, %ebx
                jg      .L11
```

En la primera columna podemos ver claramente los tres secciones (correspondientes a los tres bucles) que realizan el cálculo de la multiplicación. Con respecto a la versión de la primera modificación, vemos que hay 3 secciones más, lo que hace el código un poco más largo. Esto se

debe a que las dos primeras secciones realizan la transposición de la matriz (sobre ella misma, es decir, usando la memoria ya reservada) y la .L8 para ajustar el registro %ebx.
En la tercera columna, aparecen destacadas las 3 secciones que realizan la multiplicación, vemos que son considereblamente más largas que en las otras columnas, esto es debido a que en la segunda modificación hemos usado el desenrollado de bucle. Además también vemos que la sección .L8 es más compleja, supongo que harán falta más ajustes antes de empezar la multiplicación debido al desenrollado.

2. 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 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=1;i<=N,i++) y[i]= a*x[i] + y[i];
```

2.1. Genere 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. Incorpore los códigos al cuaderno de prácticas y destaque las diferencias entre ellos.

```
2018-05-31 19:04:49  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
→ gcc -O0 -S daxpy.c -o ../bin/daxpy0.s

2018-05-31 20:56:36  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
→ gcc -Os -S daxpy.c -o ../bin/daxpys.s

2018-05-31 20:56:47  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
→ gcc -O2 -S daxpy.c -o ../bin/daxpy2.s

2018-05-31 20:56:57  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
→ gcc -O3 -S daxpy.c -o ../bin/daxpy3.s
```

CAPTURA CÓDIGO FUENTE: `daxpy.c`

	-O0	**-Os**	**-O2**	**-O3**
Tiempos ejec.	*De 0,25 a 10 seg. aquí*			

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

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char **argv) {
    if( argc != 2 ) {
        printf("Número de argumentos inválido, introduzca tamaño.\n");
        exit(-1);
    }

    struct timespec cgt1, cgt2;
    double ncgt;
    int n = atoi(argv[1]);
    float *x= (float*) malloc(n*sizeof(float)), *y= (float*) malloc(n*sizeof(float)) ;

    for(int i= 0; i< n ; i++) x[i]= i + 0.1;

    clock_gettime(CLOCK_REALTIME, &cgt1);

    for(int i=0; i< n; i++) y[i] = 3.14 * 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));

    printf("Tiempo de ejecución: %11.9f \t / y[0]=%5.3f / y[%d]=%5.3f\n", ncgt, y[0], n, y[n-1]);

    free(x);
    free(y);
}
```

Diferencias en los tiempos de ejecución:

daxpyOs	daxpyO0	daxpyO1	daxpyO2	daxpyO3
0.137382094	0.216392357	0.131759005	0.156127484	0.082005205

```
 2018-05-31 22:18:50  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → ../bin/daxpys 20000000
Tiempo de ejecución: 0.137382094         / y[0]=0.314 / y[20000000]=62800000.000

 2018-05-31 22:32:38  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → ../bin/daxpy0 20000000
Tiempo de ejecución: 0.216392357         / y[0]=0.314 / y[20000000]=62800000.000

 2018-05-31 22:32:43  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → ../bin/daxpy1 20000000
Tiempo de ejecución: 0.131759005         / y[0]=0.314 / y[20000000]=62800000.000

 2018-05-31 22:32:47  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → ../bin/daxpy2 20000000
Tiempo de ejecución: 0.156127484         / y[0]=0.314 / y[20000000]=62800000.000

 2018-05-31 22:32:50  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Prácticas/BP4/src
○ → ../bin/daxpy3 20000000
Tiempo de ejecución: 0.082005205         / y[0]=0.314 / y[20000000]=62800000.000
```

Generamos el código en ensamblador del peor y mejor:

```
 2018-05-31 22:32:54  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Pr
ácticas/BP4/src
○ → gcc -O0 -S daxpy.c -o ../bin/daxpy0.s

 2018-05-31 22:37:05  elena in ~/Escritorio/University stuff/2º/2º Cuatrimestre/AC/Pr
ácticas/BP4/src
○ → gcc -O3 -S daxpy.c -o ../bin/daxpy3.s
```

El primero tiene 179 líneas, en comparación con las 445 del segundo. No los adjunto precisamente por eso. Se aprecia como -O0 no optimiza, hay una clara relación entre cada línea del código y su versión en ensamblador. En la opción -O3 se desenrollan los bucles, de ahí el considerable aumento en líneas de código, también aplica “function inlining” .