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): Juan Carlos Ruiz García
Grupo de prácticas: C2
Fecha de entrega: 04/05/2017
Fecha evaluación en clase:

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): *AMD A8-5600K APU with Radeon(tm) HD Graphics*

Sistema operativo utilizado: *Linux Mint 18.1 Cinnamon 64-bit*

Versión de gcc utilizada: 5.4.0

Adjunte el contenido del fichero /proc/cpuinfo de la máquina en la que ha tomado las medidas: Adjuntado en cpu_info.txt

1. Para el núcleo que se muestra en la Figura 1 (ver guion de prácticas), 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.

   1.2 Genere los códigos en ensamblador con -O2 para el original y 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.

   1.3 (Ejercicio EXTRA) Intente mejorar los resultados obtenidos transformando el código ensamblador del programa para el que se han conseguido las mejores prestaciones de tiempo

A) MULTIPLICACIÓN DE MATRICES:
CÓDIGO FUENTE: multMatrices_original.c
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#define TAM 1000

int matrizA[TAM][TAM], matrizB[TAM][TAM], resultado[TAM][TAM];

int main(int argc, char const *argv[]) {

  struct timespec cgt1,cgt2;
  double ncgt;
```

```
  // Inicializar matrices
  for (int i = 0; i < TAM; i++) {
    for (int j = 0; j < TAM; j++) {
      matrizA[i][j] = (i+j) * 2;
      matrizB[i][j] = (i+j) * 4;
    }
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);

  // Realizar calculo matrizA x matrizB
  for (int i = 0; i < TAM; i++) {
    for (int j = 0; j < TAM; j++) {
      for (int k = 0; k < TAM; k++) {
        resultado[i][j] += matrizA[i][k] * matrizB[k][j];
      }
    }
  }

  clock_gettime(CLOCK_REALTIME,&cgt2);

  // Mostrar resultado
  printf("Primer elemento resultado: %u\n",resultado[0][0]);
  printf("Ultimo elemento resultado: %u\n",resultado[TAM-1][TAM-1]);

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

  printf("Tiempo:\t%8.6f\n",ncgt);

  return 0;
}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: Lo que haremos sera desenrrollar el tercer bucle anidado. Como este (y todos) bucle va desde 0 a 999 vamos a ir incrementando de 5 en 5 en vez de ir de 1 en 1 de esta forma nos ahorramos comprobaciones e incrementos inecesarios. Cuanto de mas en mas incrementemos más mejorará el tiempo de ejecución, pero cuidado con los incrementos ya que deben ser potencia de TAM.

Modificación b) –explicación-: Lo que haremos será calcular la traspuesta de la segunda matriz para despues a la hora de realizar la multiplicación hacer FILASxFILAS y no FILASxCOLUMNAS. De esta forma se producen menos saltos y menos fallos de caché por lo que el resultado se calculará en menos tiempo y de forma mas eficiente.

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) multMatrices_modificado_a . c

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#define TAM 1000

int matrizA[TAM][TAM], matrizB[TAM][TAM], resultado[TAM][TAM];

int main(int argc, char const *argv[]) {
```

```
  struct timespec cgt1,cgt2;
  double ncgt;

  // Inicializar matrices
  for (int i = 0; i < TAM; i++) {
    for (int j = 0; j < TAM; j++) {
      matrizA[i][j] = (i+j) * 2;
      matrizB[i][j] = (i+j) * 4;
    }
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);

  // Realizar calculo matrizA x matrizB
  for (int i = 0; i < TAM; i++) {
    for (int j = 0; j < TAM; j++) {
      for (int k = 0; k < TAM; k+=5) {
        resultado[i][j] += matrizA[i][k] * matrizB[k][j];
        resultado[i][j] += matrizA[i][k+1] * matrizB[k+1][j];
        resultado[i][j] += matrizA[i][k+2] * matrizB[k+2][j];
        resultado[i][j] += matrizA[i][k+3] * matrizB[k+3][j];
        resultado[i][j] += matrizA[i][k+4] * matrizB[k+4][j];
      }
    }
  }

  clock_gettime(CLOCK_REALTIME,&cgt2);

  // Mostrar resultado
  printf("Primer elemento resultado: %u\n",resultado[0][0]);
  printf("Ultimo elemento resultado: %u\n",resultado[TAM-1][TAM-1]);

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

  printf("Tiempo:\t%8.6f\n",ncgt);

  return 0;
}
```

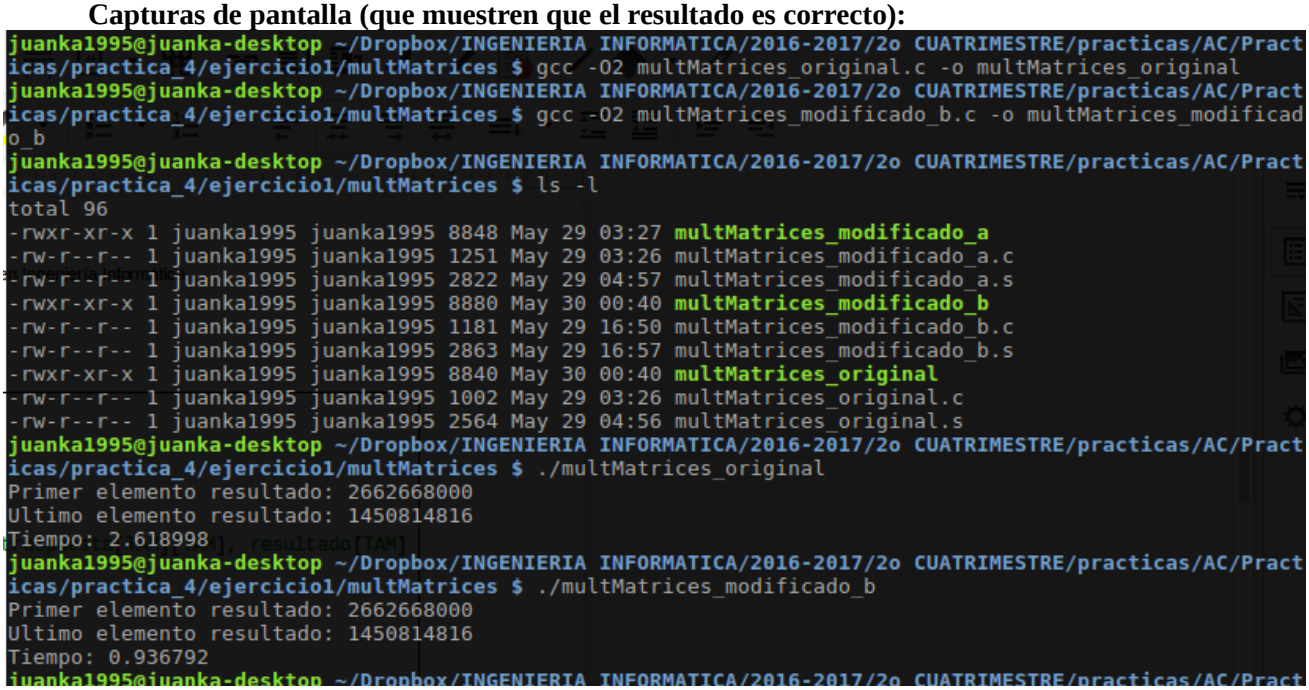
Capturas de pantalla (que muestren que el resultado es correcto):

```
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/practicas/AC/Pract
icas/practica_4/ejercicio1/multMatrices $ gcc -O2 multMatrices_original.c -o multMatrices_original
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/practicas/AC/Pract
icas/practica_4/ejercicio1/multMatrices $ gcc -O2 multMatrices_modificado_a.c -o multMatrices_modificad
o_a
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/practicas/AC/Pract
icas/practica_4/ejercicio1/multMatrices $ ls -l
total 48
-rwxr-xr-x 1 juanka1995 juanka1995 8848 May 29 03:27 multMatrices_modificado_a
-rw-r--r-- 1 juanka1995 juanka1995 1251 May 29 03:26 multMatrices_modificado_a.c
-rwxr-xr-x 1 juanka1995 juanka1995 8840 May 29 03:27 multMatrices_original
-rw-r--r-- 1 juanka1995 juanka1995 1002 May 29 03:26 multMatrices_original.c
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/practicas/AC/Pract
icas/practica_4/ejercicio1/multMatrices $ ./multMatrices_original
Primer elemento resultado: 2662668000
Ultimo elemento resultado: 1450814816
Tiempo: 2.662540
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/practicas/AC/Pract
icas/practica_4/ejercicio1/multMatrices $ ./multMatrices_modificado_a
Primer elemento resultado: 2662668000
Ultimo elemento resultado: 1450814816
Tiempo: 2.330334
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/practicas/AC/Pract
```

b) multMatrices_modificado_b.c
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#define TAM 1000

int matrizA[TAM][TAM], matrizB[TAM][TAM], traspuesta[TAM][TAM], resultado[TAM]
[TAM];

int main(int argc, char const *argv[]) {

  struct timespec cgt1,cgt2;
  double ncgt;

  // Inicializar matrices
  for (int i = 0; i < TAM; i++) {
    for (int j = 0; j < TAM; j++) {
      matrizA[i][j] = (i+j) * 2;
      matrizB[i][j] = (i+j) * 4;
    }
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);

  // Hacer traspuesta de matrizB
  for (int i = 0; i < TAM; i++) {
    for (int j = 0; j < TAM; j++) {
      traspuesta[i][j] = matrizB[j][i];
    }
  }

  // Realizar calculo matrizA x matrizB
  for (int i = 0; i < TAM; i++) {
    for (int j = 0; j < TAM; j++) {
      for (int k = 0; k < TAM; k++) {
        resultado[i][j] += matrizA[i][k] * traspuesta[j][k];
      }
    }
  }

  clock_gettime(CLOCK_REALTIME,&cgt2);

  // Mostrar resultado
  printf("Primer elemento resultado: %u\n",resultado[0][0]);
  printf("Ultimo elemento resultado: %u\n",resultado[TAM-1][TAM-1]);

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

  printf("Tiempo:\t%8.6f\n",ncgt);

  return 0;
}
```

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



1.1. TIEMPOS:

Modificación	**-O2**
Sin modificar	*2,662540 s*
Modificación a)	2,330334 s
Modificación b)	0,936792 s

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

Como podemos ver en los tiempo obtenidos, la mejor opción es calcular la matriz traspuesta para multiplicar FILASxFILAS y asi producir menos fallos de caché. Esta opción es mucho más rapida que la primera modificación por lo que interesa perder tiempo en calcular la traspuesta de una matriz para despues hacer el cálculo con ella.

1.2. CÓDIGO EN ENSAMBLADOR DEL ORIGINAL Y DE DOS MODIFICACIONES (ADJUNTAR AL .ZIP):

(PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR EVALUADA, USE COLORES PARA DESTACAR LAS DIFERENCIAS)

multMatrices_original.s	multMatrices_modificado_a.s	pmm-secuencial-modificado_b.s

.L3:
...
call clock_gettime
movl $resultado+4000, %r10d
movl $matrizA, %r9d
movl $matrizA+4000000, %r11d
.L5:
leaq -4000(%r10), %r8
movl $matrizB+4000000, %edi
.p2align 4,,10
.p2align 3
.L9:
movl (%r8), %esi
leaq -4000000(%rdi), %rax
movq %r9, %rcx
	.L3:
	...
	call clock_gettime
	movl $resultado+4000, %r10d
	movl $matrizA, %r9d
	movl $matrizA+4000000, %r11d
	.L5:
	leaq -4000(%r10), %r8
	movl $matrizB+4000000, %edi
	.p2align 4,,10
	.p2align 3
	.L9:
	movl (%r8), %eax
	leaq -4000000(%rdi), %rdx
	movq %r9, %rcx
		.L3:
		...
		call clock_gettime
		movl $matrizB, %edi
		movl $traspuesta, %r8d
		.L5:
		leaq 4000000(%rdi), %rsi
		movq %r8, %rdx
		movq %rdi, %rax
		.p2align 4,,10
		.p2align 3
		.L6:
		movl (%rax), %ecx
		addq $4000, %rax
		addq $4, %rdx

```
.p2align 4,,10
.p2align 3
.L6:
movl (%rcx), %edx
addq $4000, %rax
addq $4, %rcx
imull -4000(%rax), %edx
addl %edx, %esi
cmpq %rdi, %rax
jne .L6
movl %esi, (%r8)
addq $4, %r8
leaq 4(%rax), %rdi
cmpq %r10, %r8
jne .L9
addq $4000, %r9
leaq 4000(%r8), %r10
cmpq %r9, %r11
jne .L5
leaq 16(%rsp), %rsi
xorl %edi, %edi
call clock_gettime
...
```

```
.p2align 4,,10
.p2align 3
.L6:
movl (%rcx), %esi
addq $20000, %rdx
addq $20, %rcx
imull -20000(%rdx), %esi
addl %esi, %eax
movl -16(%rcx), %esi
imull -16000(%rdx), %esi
addl %esi, %eax
movl -12(%rcx), %esi
imull -12000(%rdx), %esi
addl %esi, %eax
movl -8(%rcx), %esi
imull -8000(%rdx), %esi
addl %eax, %esi
movl -4(%rcx), %eax
imull -4000(%rdx), %eax
addl %esi, %eax
cmpq %rdx, %rdi
jne .L6
movl %eax, (%r8)
addq $4, %r8
addq $4, %rdi
cmpq %r10, %r8
jne .L9
addq $4000, %r9
leaq 4000(%r8), %r10
cmpq %r9, %r11
jne .L5
leaq 16(%rsp), %rsi
xorl %edi, %edi
call clock_gettime
...
```

```
movl %ecx, -4(%rdx)
cmpq %rsi, %rax
jne .L6
addq $4, %rdi
addq $4000, %r8
cmpq $matrizB+4000, %rdi
jne .L5
movl $resultado+4000000, %r10d
movl $resultado, %r9d
.L10:
movq %r9, %rdi
movq %r9, %r8
xorl %esi, %esi
subq $resultado, %rdi
.p2align 4,,10
.p2align 3
.L12:
movl (%r8), %ecx
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L8:
movl matrizA(%rdi,%rax), %edx
imull traspuesta(%rsi,%rax),
%edx
addq $4, %rax
addl %edx, %ecx
cmpq $4000, %rax
jne .L8
addq $4000, %rsi
movl %ecx, (%r8)
addq $4, %r8
cmpq $4000000, %rsi
jne .L12
addq $4000, %r9
cmpq %r9, %r10
jne .L10
leaq 16(%rsp), %rsi
xorl %edi, %edi
call clock_gettime
...
```

B) CÓDIGO FIGURA 1:

CÓDIGO FUENTE: `figura1_sin_optimizar.c`

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#define TAM_VECTOR 5000
#define NUM_ITER 40000

struct{
  int a;
  int b;
} s[TAM_VECTOR];

int R[NUM_ITER];

int main(int argc, char const *argv[]) {

  int X1, X2;
  struct timespec cgt1,cgt2;
  double ncgt;
```

```
  for (int i = 0; i < TAM_VECTOR; i++) {
    s[i].a = i*2;
    s[i].b = i*4;
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);

  for (int ii = 0; ii < NUM_ITER; ii++) {
    X1 = 0; X2 = 0;
    for (int i = 0; i < TAM_VECTOR; i++) {
      X1 += 2*s[i].a+ii;
    }
    for (int i = 0; i < TAM_VECTOR; 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("R[0]: %u\nR[39999]: %u\n\nTiempo:\t%8.6f\n",R[0],R[39999],ncgt);

  return 0;
}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: Reduccion de 2 bucles a uno solo. Existen dentro del primer *for* dos mas los cuales van desde 0 hasta TAM_VECTOR-1. La idea es juntar ambos bucles en uno y así reducir el tiempo de computo a la mitad.

Modificación b) –explicación-: Cambiar el if(…) {} else {} por `R[ii] = (X1 < X2) ? X1 : X2;` Esto ahorra muchos saltos a la hora de la ejecución lo que hace mas efeciente nuestro código.

1.1. CÓDIGOS FUENTE MODIFICACIONES

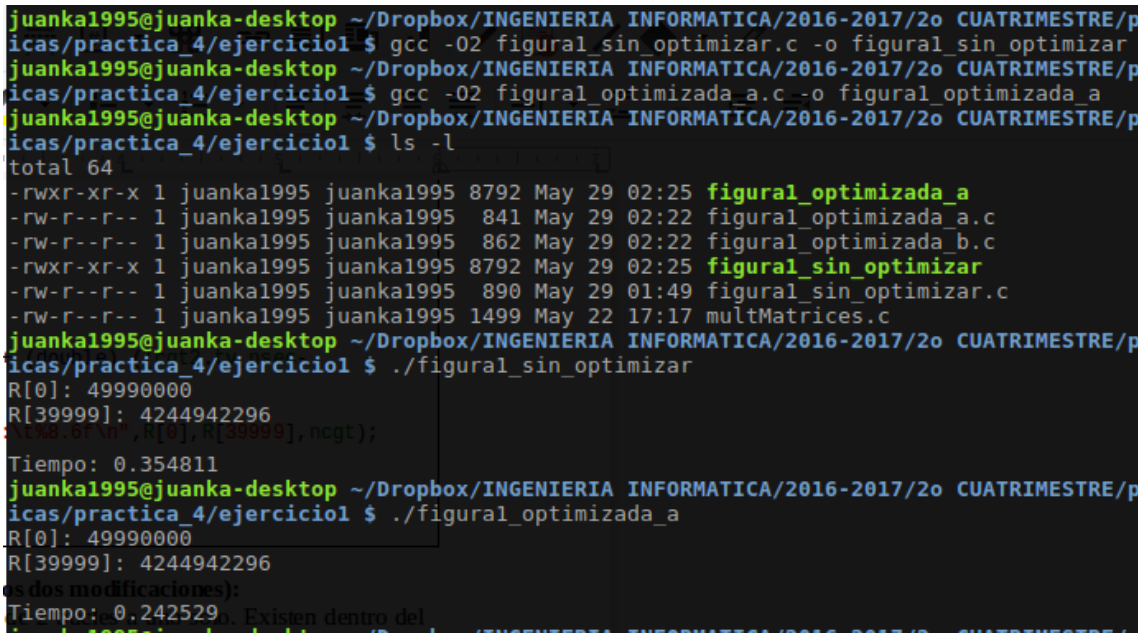
a) figura1_optimizada_a.c

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#define TAM_VECTOR 5000
#define NUM_ITER 40000

struct{
  int a;
  int b;
} s[TAM_VECTOR];

int R[NUM_ITER];

int main(int argc, char const *argv[]) {
```

```
  int X1, X2, ii;
  struct timespec cgt1,cgt2;
  double ncgt;

  for (int i = 0; i < TAM_VECTOR; i++) {
    s[i].a = i*2;
    s[i].b = i*4;
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);

  for (ii = 0; ii < NUM_ITER; ii++) {
    X1 = 0; X2 = 0;
    for (int i = 0; i < TAM_VECTOR; i++) {
      X1 += 2*s[i].a+ii;
      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("R[0]: %u\nR[39999]: %u\n\nTiempo:\t
%8.6f\n",R[0],R[39999],ncgt);

  return 0;
}
```

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



b) figura1_optimizada_b.c
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#define TAM_VECTOR 5000
#define NUM_ITER 40000

struct{
  int a;
  int b;
} s[TAM_VECTOR];

int R[NUM_ITER];

int main(int argc, char const *argv[]) {

  int X1, X2, ii;
  struct timespec cgt1,cgt2;
  double ncgt;

  for (int i = 0; i < TAM_VECTOR; i++) {
    s[i].a = i*2;
    s[i].b = i*4;
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);

  for (ii = 0; ii < NUM_ITER; ii++) {
    X1 = 0; X2 = 0;
    for (int i = 0; i < TAM_VECTOR; i++) {
```

```
      X1 += 2*s[i].a+ii;
    }
    for (int i = 0; i < TAM_VECTOR; i++) {
      X2 += 3*s[i].b-ii;
    }
    R[ii] = (X1 < X2) ? X1 : 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("R[0]: %u\nR[39999]: %u\n\nTiempo:\t
%8.6f\n",R[0],R[39999],ncgt);

  return 0;
}
```

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

```
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/p
icas/practica_4/ejercicio1 $ gcc -O2 figura1_sin_optimizar.c -o figura1_sin_optimizar
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/p
icas/practica_4/ejercicio1 $ gcc -O2 figura1_optimizada_b.c -o figura1_optimizada_b
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/p
icas/practica_4/ejercicio1 $ ls -l
total 80
-rwxr-xr-x 1 juanka1995 juanka1995 8792 May 29 02:25 figura1_optimizada_a
-rw-r--r-- 1 juanka1995 juanka1995  841 May 29 02:22 figura1_optimizada_a.c
-rwxr-xr-x 1 juanka1995 juanka1995 8792 May 29 02:27 figura1_optimizada_b
-rw-r--r-- 1 juanka1995 juanka1995  862 May 29 02:22 figura1_optimizada_b.c
-rwxr-xr-x 1 juanka1995 juanka1995 8792 May 29 02:27 figura1_sin_optimizar
-rw-r--r-- 1 juanka1995 juanka1995  890 May 29 01:49 figura1_sin_optimizar.c
-rw-r--r-- 1 juanka1995 juanka1995 1499 May 22 17:17 multMatrices.c
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/p
icas/practica_4/ejercicio1 $ ./figura1_sin_optimizar
R[0]: 49990000
R[39999]: 4244942296

Tiempo: 0.353787
juanka1995@juanka-desktop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUATRIMESTRE/p
icas/practica_4/ejercicio1 $ ./figura1_optimizada_b
R[0]: 49990000
R[39999]: 4244942296

Tiempo: 0.346150
```

1.1. TIEMPOS:

Modificación	**-O2**
Sin modificar	*0,353787 s*
Modificación a)	0,242529 s
Modificación b)	0,346450 s

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

Como podemos observar se optiene un mayor beneficio cuando en vez de utilizar 2 for usamos 1 solo. Esto tiene sentido debido a que supongamos un bucle de 60 iteraciones a 1s por iteración, si usamos 2 bucles como estos, se tardaría un total de 2 minutos en ejecutarse mientras que si aprovechamos y en 1 solo hacemos las mismas operaciones el tiempo se reduciría a la mitad (1 minuto).

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

figura1_sin_optimizar.s	figura1_optimizada_a.s	figura1_optimizada_b.s

.L2:
...
call clock_gettime
xorl %r9d, %r9d
movl $s+40000, %r8d
.p2align 4,,10
.p2align 3
.L3:
movl %r9d, %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 %rax, %r8
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 $s+40004, %rax
jne .L5
cmpl %ecx, %esi
cmovl %esi, %ecx
movl %ecx, R(,%r9,4)
addq $1, %r9
cmpq $40000, %r9
jne .L3
leaq 16(%rsp), %rsi
xorl %edi, %edi
call clock_gettime
...
	.L2:
	...
	call clock_gettime
	xorl %r9d, %r9d
	movl $s+40000, %r8d
	.p2align 4,,10
	.p2align 3
	.L3:
	movl %r9d, %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 %rax, %r8
	jne .L4
	cmpl %ecx, %esi
	cmovl %esi, %ecx
	movl %ecx, R(,%r9,4)
	addq $1, %r9
	cmpq $40000, %r9
	jne .L3
	leaq 16(%rsp), %rsi
	xorl %edi, %edi
	call clock_gettime
	...
		.L2:
		...
		call clock_gettime
		xorl %r10d, %r10d
		movl $s+40000, %r9d
		movl $s+40004, %r8d
		.p2align 4,,10
		.p2align 3
		.L3:
		movl %r10d, %edi
		movl $s, %eax
		xorl %ecx, %ecx
		.p2align 4,,10
		.p2align 3
		.L4:
		movl (%rax), %edx
		addq $8, %rax
		leal (%rdi,%rdx,2), %edx
		addl %edx, %ecx
		cmpq %rax, %r9
		jne .L4
		movl $s+4, %eax
		xorl %esi, %esi
		.p2align 4,,10
		.p2align 3
		.L5:
		movl (%rax), %edx
		addq $8, %rax
		leal (%rdx,%rdx,2), %edx
		subl %edi, %edx
		addl %edx, %esi
		cmpq %rax, %r8
		jne .L5
		cmpl %esi, %ecx
		cmovg %esi, %ecx
		movl %ecx, R(,%r10,4)
		addq $1, %r10
		cmpq $40000, %r10
		jne .L3
		leaq 16(%rsp), %rsi
		xorl %edi, %edi
		call clock_gettime
		...

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 opciones de optimización del compilador (-O0, -O2, -O3) y explique las diferencias que se observan en el código justificando las mejoras en velocidad que acarrean. Incorpore los códigos al cuaderno de prácticas y destaque las diferencias entre ellos.

2.2. (Ejercicio EXTRA) Para la mejor de las opciones, obtenga los tiempos de ejecución con distintos valores de N y determine para su sistema los valores de Rmax (valor máximo del número de operaciones en coma flotante por unidad de tiempo), Nmax (valor de N para el que se consigue Rmax), y N1/2 (valor de N para el que se obtiene Rmax/2). Estime el valor de la velocidad pico (Rpico) del procesador (consulte en [4] el número de ciclos por instrucción punto flotante para la familia y modelo de procesador que está utilizando) y compárela con el valor obtenido para Rmax. -Consulte la Lección 3 del Tema 1.

CÓDIGO FUENTE: `daxpy.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#define TAM 100000000

int vectorA[TAM], vectorB[TAM];

int main(int argc, char const *argv[]) {

  int a = 5;
  struct timespec cgt1,cgt2;
  double ncgt;

  // Inicializar vectores
  for (int i = 0; i < TAM; i++) {
    vectorA[i] = (i+1) * 2;
    vectorB[i] = (i+2) * 4;
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);

  // Benchmark
  for (int i = 0; i < TAM; i++) {
    vectorA[i] = vectorB[i] * a + vectorA[i];
  }

  clock_gettime(CLOCK_REALTIME,&cgt2);

  // Mostrar resultado
  printf("Primer elemento resultado: %u\n",vectorA[0]);
  printf("Ultimo elemento resultado: %u\n",vectorA[TAM-1]);

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

  printf("Tiempo:\t%8.6f\n",ncgt);

  return 0;
```

```
}
```

Tiempos ejec.	-O0	-O2	-O3
	0,306442 s	0,110154 s	0,107751 s

CAPTURAS DE PANTALLA:

```
juanka1995@juanka-laptop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUA
/practicas/AC/Practicas/practica_4/ejercicio2 $ gcc -O0 daxpy.c -o daxpy0
juanka1995@juanka-laptop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUA
/practicas/AC/Practicas/practica_4/ejercicio2 $ gcc -O2 daxpy.c -o daxpy2
juanka1995@juanka-laptop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUA
/practicas/AC/Practicas/practica_4/ejercicio2 $ gcc -O3 daxpy.c -o daxpy3
juanka1995@juanka-laptop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUA
/practicas/AC/Practicas/practica_4/ejercicio2 $ ls -l
total 56
-rwxr-xr-x 1 juanka1995 juanka1995 8784 May 29 18:51 daxpy0
-rwxr-xr-x 1 juanka1995 juanka1995 8792 May 29 18:51 daxpy2
-rwxr-xr-x 1 juanka1995 juanka1995 8792 May 29 18:51 daxpy3
-rw-r--r-- 1 juanka1995 juanka1995  798 May 29 18:49 daxpy.c
juanka1995@juanka-laptop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUA
/practicas/AC/Practicas/practica_4/ejercicio2 $ ./daxpy0
Primer elemento resultado: 42
Ultimo elemento resultado: 2200000020
Tiempo: 0.306442
juanka1995@juanka-laptop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUA
/practicas/AC/Practicas/practica_4/ejercicio2 $ ./daxpy2
Primer elemento resultado: 42
Ultimo elemento resultado: 2200000020
Tiempo: 0.110154
juanka1995@juanka-laptop ~/Dropbox/INGENIERIA INFORMATICA/2016-2017/2o CUA
/practicas/AC/Practicas/practica_4/ejercicio2 $ ./daxpy3
Primer elemento resultado: 42
Ultimo elemento resultado: 2200000020
Tiempo: 0.107751
```

COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:

Como podemos observar el -O2 es el que menos número de instrucciones utiliza para llevar acabo la misma tarea. Realmente no sabría decirle muchas diferencias debido a que el nivel -O3 y el -O2 empieza a ser bastante complicado de comprender…

CÓDIGO EN ENSAMBLADOR (ADJUNTAR AL .ZIP):
(PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR DONDE ESTÁ EL CÓDIGO EVALUADO, USE COLORES PARA DESTACAR LAS DIFERENCIAS)

daxpy0.s

```
.L2:
...
call clock_gettime
movl $0, -64(%rbp)
jmp .L4
.L5:
movl -64(%rbp), %eax
cltq
movl vectorB(,%rax,4), %eax
imull -60(%rbp), %eax
movl %eax, %edx
movl -64(%rbp), %eax
cltq
movl vectorA(,%rax,4), %eax
```

daxpy2.s

```
L2:
...
call clock_gettime
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L3:
movl vectorB(%rax), %edx
leal (%rdx,%rdx,4), %edx
addl %edx, vectorA(%rax)
addq $4, %rax
cmpq $400000000, %rax
jne .L3
leaq 16(%rsp), %rsi
```

daxpy3.s

```
.L2:
...
call clock_gettime
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L3:
movdqa vectorB(%rax), %xmm0
addq $16, %rax
movdqa %xmm0, %xmm1
pslld $2, %xmm1
paddd %xmm1, %xmm0
paddd vectorA-16(%rax), %xmm0
movaps %xmm0, vectorA-16(%rax)
```

```
addl %eax, %edx
movl -64(%rbp), %eax
cltq
movl %edx, vectorA(,%rax,4)
addl $1, -64(%rbp)
.L4:
cmpl $99999999, -64(%rbp)
jle .L5
leaq -32(%rbp), %rax
movq %rax, %rsi
movl $0, %edi
call clock_gettime
...
```

```
xorl %edi, %edi
call clock_gettime
...
```

```
cmpq $400000000, %rax
jne .L3
leaq 16(%rsp), %rsi
xorl %edi, %edi
call clock_gettime
...
```