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): Álvaro Fernández García
Grupo de prácticas: A2
Fecha de entrega:
Fecha evaluación en clase:

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): *AMD A10-7300 Radeon R6, 10 Compute Cores 4C+6G*

Sistema operativo utilizado: *Elementary OS Loki*

Versión de gcc utilizada: *gcc (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609*

Adjunte el contenido del fichero /proc/cpuinfo de la máquina en la que ha tomado las medidas

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: `pmm-secuencial.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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

		struct timespec cgt1,cgt2;
		double ncgt;

		if(argc < 2){
				printf("Modo de empleo: %s, N\n", argv[0]);
				exit(-1);
		}

		int N = atoi(argv[1]);
```

```
                //Reservar memoria dinámica:

                double** A = (double**) malloc(N*sizeof(double*));
                double** B = (double**) malloc(N*sizeof(double*));
                double** C = (double**) malloc(N*sizeof(double*));
                for(int i = 0; i<N; ++i){
                                A[i] = (double*) malloc(N*sizeof(double));
                                B[i] = (double*) malloc(N*sizeof(double));
                                C[i] = (double*) malloc(N*sizeof(double));
                }

                if(A == NULL || B == NULL || C == NULL){
                                printf("No se ha podido reservar memoria\n");
                                exit(-1);
                }

                //Inicializar elementos:

                for(int i = 0; i<N; ++i)
                                for(int j = 0; j<N; ++j){
                                                A[i][j] = 0.0;
                                                B[i][j] = N*0.1 + i*0.1;
                                                C[i][j] = N*0.1 - i*0.1;
                                }

                //Realizar R = M * V y medir tiempo:

                clock_gettime(CLOCK_REALTIME, &cgt1);

                for(int i = 0; i<N; ++i)
                                for(int j = 0; j<N; ++j)
                                                for(int k = 0; k<N; ++k)
                                                                A[i][j] += B[i][k] * C[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));

                //Imprimir resultados:

                if(N<12){
                                printf("Tiempo transcurrido (seg): %11.9f \n\nMatriz 1
generada:\n", ncgt);
                                for(int i = 0; i<N; ++i){
                                                for(int j = 0; j<N; ++j)
                                                                printf("[%.1f]", B[i][j]);
                                                printf("\n");
                                }
                                printf("\n\nMatriz 2 generada:\n");
                                for(int i = 0; i<N; ++i){
                                                for(int j = 0; j<N; ++j)
                                                                printf("[%.1f]", C[i][j]);
                                                printf("\n");
                                }

                                printf("\n\nResultado:\n");
                                for(int i = 0; i<N; ++i){
                                                for(int j = 0; j<N; ++j)
                                                                printf("[%.2f]", A[i][j]);
                                                printf("\n");
                                }

                                printf("\n");
                }
                else
                                printf("Tiempo(seg.): %11.9f \nTamaño: %u \nB[0][0]*C[0]
[0]=A[0][0] = %.1f \nB[%d][%d]*C[%d][%d]=A[%d][%d] = %.2f \n",ncgt,N, A[0][0], N-1, N-
1,N-1, N-1,N-1, N-1, A[N-1][N-1]);
```

```
                //Liberar memoria:
                for(int i = 0; i<N; ++i){
                                free(A[i]);
                                free(B[i]);
                                free(C[i]);
                }

                free(A);
                free(B);
                free(C);

                exit(0);
}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: En la primera modificación simplemente se ha modificado el almacenamiento de las tres matrices, sustituyéndolas por un solo vector que guarda la matriz por filas, (por tanto también ha sido necesario modificar los accesos). Y también se han alineado los datos en memoria.

Modificación b) –explicación-: En la segunda modificación se ha pretendido aprovechar la localidad espacial modificando el orden en el que se realizan los bucles (antes i → j → k; después i → k → j) lo cual no solo nos ha permitido mejorar el rendimiento si no también eliminar la dependencia RAW existente entre A[i*N+j].

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) `pmm-secuencial-modificado_a.c`

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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

                struct timespec cgt1,cgt2;
                double ncgt;

                if(argc < 2){
                                printf("Modo de empleo: %s, N\n", argv[0]);
                                exit(-1);
                }

                int N = atoi(argv[1]);

                //Reservar memoria dinámica:
                //Alineación de los datos y modificación de la matriz por un vector:
                double *A, *B, *C;

                double *Atmp = (double *) malloc(N*N*sizeof(double)+63);
                double *Btmp = (double *) malloc(N*N*sizeof(double)+63);
                double *Ctmp = (double *) malloc(N*N*sizeof(double)+63);

                A = (double *) (((long int)Atmp+63)&~(63));
                B = (double *) (((long int)Btmp+63)&~(63));
                C = (double *) (((long int)Ctmp+63)&~(63));

                if(A == NULL || B == NULL || C == NULL){
                                printf("No se ha podido reservar memoria\n");
                                exit(-1);
                }

                //Inicializar elementos:

                for(int i = 0; i<N; ++i)
                                for(int j = 0; j<N; ++j){
                                                A[i*N+j] = 0.0;
```

```
                                        B[i*N+j] = N*0.1 + i*0.1;
                                        C[i*N+j] = N*0.1 - i*0.1;
                                }

                //Realizar R = M * V y medir tiempo:

                clock_gettime(CLOCK_REALTIME, &cgt1);

                for(int i = 0; i<N; ++i)
                                for(int j = 0; j<N; ++j)
                                        for(int k = 0; k<N; ++k)
                                                A[i*N+j] += B[i*N+k] *
C[k*N+j];

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

                //Imprimir resultados:

                if(N<12){
                                printf("Tiempo transcurrido (seg): %11.9f \n\nMatriz 1
generada:\n", ncgt);
                                for(int i = 0; i<N; ++i){
                                        for(int j = 0; j<N; ++j)
                                                printf("[%.1f]", B[i*N+j]);
                                        printf("\n");
                                }
                                printf("\n\nMatriz 2 generada:\n");
                                for(int i = 0; i<N; ++i){
                                        for(int j = 0; j<N; ++j)
                                                printf("[%.1f]", C[i*N+j]);
                                        printf("\n");
                                }

                                printf("\n\nResultado:\n");
                                for(int i = 0; i<N; ++i){
                                        for(int j = 0; j<N; ++j)
                                                printf("[%.2f]", A[i*N+j]);
                                        printf("\n");
                                }

                                printf("\n");
                }
                else
                                printf("Tiempo(seg.): %11.9f \nTamaño: %u \nB[0][0]*C[0]
[0]=A[0][0] = %.1f \nB[%d][%d]*C[%d][%d]=A[%d][%d] = %.2f \n",ncgt,N, A[0], N-1, N-1,N-
1, N-1,N-1, N-1, A[N*N-1]);

                //Liberar memoria:

                free(Atmp);
                free(Btmp);
                free(Ctmp);

                exit(0);
}
```

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

b) `pmm-secuencial-modificado_b.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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

		struct timespec cgt1,cgt2;
		double ncgt;

		if(argc < 2){
				printf("Modo de empleo: %s, N\n", argv[0]);
				exit(-1);
		}

		int N = atoi(argv[1]);

		//Reservar memoria dinámica:
		//Alineación de los datos y modificación de la matriz por un vector:
		double *A, *B, *C;

		double *Atmp = (double *) malloc(N*N*sizeof(double)+63);
		double *Btmp = (double *) malloc(N*N*sizeof(double)+63);
		double *Ctmp = (double *) malloc(N*N*sizeof(double)+63);

		A = (double *) (((long int)Atmp+63)&~(63));
		B = (double *) (((long int)Btmp+63)&~(63));
		C = (double *) (((long int)Ctmp+63)&~(63));

		if(A == NULL || B == NULL || C == NULL){
				printf("No se ha podido reservar memoria\n");
				exit(-1);
		}

		//Inicializar elementos:

		for(int i = 0; i<N; ++i)
				for(int j = 0; j<N; ++j){
						A[i*N+j] = 0.0;
						B[i*N+j] = N*0.1 + i*0.1;
						C[i*N+j] = N*0.1 - i*0.1;
				}
```

```
                //Realizar R = M * V y medir tiempo:

                clock_gettime(CLOCK_REALTIME, &cgt1);

                for(int i = 0; i<N; ++i)
                                for(int k = 0; k<N; ++k)
                                                for(int j = 0; j<N; ++j)
                                                                A[i*N+j] += B[i*N+k] *
C[k*N+j];

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

                //Imprimir resultados:

                if(N<12){
                                printf("Tiempo transcurrido (seg): %11.9f \n\nMatriz 1
generada:\n", ncgt);
                                for(int i = 0; i<N; ++i){
                                                for(int j = 0; j<N; ++j)
                                                                printf("[%.1f]", B[i*N+j]);
                                                printf("\n");
                                }
                                printf("\n\nMatriz 2 generada:\n");
                                for(int i = 0; i<N; ++i){
                                                for(int j = 0; j<N; ++j)
                                                                printf("[%.1f]", C[i*N+j]);
                                                printf("\n");
                                }

                                printf("\n\nResultado:\n");
                                for(int i = 0; i<N; ++i){
                                                for(int j = 0; j<N; ++j)
                                                                printf("[%.2f]", A[i*N+j]);
                                                printf("\n");
                                }

                                printf("\n");
                }
                else
                                printf("Tiempo(seg.): %11.9f \nTamaño: %u \nB[0][0]*C[0]
[0]=A[0][0] = %.1f \nB[%d][%d]*C[%d][%d]=A[%d][%d] = %.2f \n",ncgt,N, A[0], N-1, N-1,N-
1, N-1,N-1, N-1, A[N*N-1]);

                //Liberar memoria:

                free(Atmp);
                free(Btmp);
                free(Ctmp);

                exit(0);
}
```

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

1.1. TIEMPOS:

Modificación	**-O2**
Sin modificar	8,894814983
Modificación a)	4,535319262
Modificación b)	2,324571332

1.1. COMENTARIOS SOBRE LOS RESULTADOS: El simple hecho de alinear los vectores en caché y realizar un vector unidimensional en lugar de una matriz, hace que los tiempos se reduzcan a la mitad. Nuevamente, al aplicar el cambio en el orden de los bucles para aprovechar la localidad espacial los tiempos vuelven a reducirse a la mitad.

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)

pmm-secuencial.s	pmm-secuencial-modificado_b.s

(RESERVA)
movslq %eax, %rbp
movq %rax, %rbx
movq %rax, 16(%rsp)
salq $3, %rbp
movl %eax, %r13d
movq %rbp, %rdi
call malloc
movq %rbp, %rdi
movq %rax, %r12
call malloc
movq %rbp, %rdi
movq %rax, 8(%rsp)
call malloc
testl %ebx, %ebx
movq %rax, %r15
jle .L3
movl %ebx, %eax
xorl %ebx, %ebx
subl $1, %eax
leaq 8(,%rax,8), %r14
movl %eax, 28(%rsp)
	(RESERVA)
	movl %eax, %r14d
	movq %rax, %r15
	movl %eax, %r13d
	imull %eax, %r14d
	movslq %r14d, %r14
	salq $3, %r14
	leaq 63(%r14), %rbx
	movq %rbx, %rdi
	call malloc
	movq %rbx, %rdi
	movq %rax, %rbp
	movq %rax, 16(%rsp)
	call malloc
	movq %rbx, %rdi
	movq %rax, %r12
	movq %rax, 24(%rsp)
	leaq 63(%rbp), %rbp
	leaq 63(%r12), %r12
	call malloc
	leaq 63(%rax), %rbx
	movq %rax, 32(%rsp)

```
.L5:
movq    %rbp, %rdi
call    malloc
movq    %rbp, %rdi
movq    %rax, (%r12,%rbx)
call    malloc
movq    8(%rsp), %rdi
movq    %rax, (%rdi,%rbx)
movq    %rbp, %rdi
call    malloc
movq    %rax, (%r15,%rbx)
addq    $8, %rbx
cmpq    %rbx, %r14
jne     .L5
pxor    %xmm2, %xmm2
xorl    %edi, %edi
movsd   .LC2(%rip), %xmm3
cvtsi2sd        16(%rsp), %xmm2
mulsd   %xmm3, %xmm2
pxor    %xmm0, %xmm0
movapd  %xmm2, %xmm4
movq    8(%rsp), %rax
movapd  %xmm2, %xmm1
movq    (%r12,%rdi,8), %rsi
movq    (%r15,%rdi,8), %rdx
cvtsi2sd        %edi, %xmm0
movq    (%rax,%rdi,8), %rcx
xorl    %eax, %eax
mulsd   %xmm3, %xmm0
subsd   %xmm0, %xmm4
addsd   %xmm0, %xmm1
movapd  %xmm4, %xmm0
(PRODUCTO)
.L12:
movq    8(%rsp), %rax
movq    (%r12,%r11,8), %rdi
xorl    %ecx, %ecx
movq    (%rax,%r11,8), %rsi
.L10:
movsd   (%rdi,%rcx), %xmm1
xorl    %eax, %eax
.L9:
movq    (%r15,%rax,8), %rdx
movsd   (%rdx,%rcx), %xmm0
mulsd   (%rsi,%rax,8), %xmm0
addq    $1, %rax
cmpl    %eax, %r13d
addsd   %xmm0, %xmm1
jg      .L9
movsd   %xmm1, (%rdi,%rcx)
addq    $8, %rcx
cmpq    %r9, %rcx
jne     .L10
addq    $1, %r11
cmpl    %r11d, %r13d
jg      .L12
```

```
andq    $-64, %rbx
(ALINEAMIENTO)
leaq    63(%rbp), %rbp
leaq    63(%r12), %r12
call    malloc
leaq    63(%rax), %rbx
movq    %rax, 40(%rsp)
andq    $-64, %rbx
(PRODUCTO)
.L12:
leaq    (%r12,%r8), %rdi
movq    %rbx, %rsi
xorl    %edx, %edx
.L10:
xorl    %eax, %eax
.L9:
movsd   (%rdi,%rdx,8), %xmm0
mulsd   (%rsi,%rax,8), %xmm0
addsd   (%r12,%rax,8), %xmm0
movsd   %xmm0, (%r12,%rax,8)
addq    $1, %rax
cmpl    %eax, %r14d
jg      .L9
addq    $1, %rdx
addq    %rcx, %rsi
cmpl    %edx, %r14d
jg      .L10
addl    $1, %r15d
addq    %rcx, %r12
cmpl    %r14d, %r15d
jne     .L12
```

La mejor optimización obtenida ha sido la modificación b, ya que el tiempo pasa de 8 a 2 segundos. Con respecto a las variaciones existentes en el código ensamblador, en la parte referente a la reserva de memoria se puede ver fácilmente que se reduce muchísimo ya que, al tratarse de un vector en lugar de una matriz, desaparecen el bucle (la etiqueta .L5 y todo lo derivante) y las tres llamadas a malloc subyacentes. Como además estamos alineando la matriz podemos observar que en el programa modificado, la llamada a malloc utiliza un leaq con 63 (BOUND-1) cosa que no aparece en el programa original. También se añade la parte de código correspondiente al alineamiento. Con respecto al calculo del producto, realmente solo se ha modificado el orden de los bucles, algo que no es apreciable en el código ensamblador, aunque

si que se observan algunas diferencias en las instrucciones empleadas, por ejemplo en el programa modificado se utilizan leaq y distintos ordenes y números de operaciones movsd.

B) CÓDIGO FIGURA 1:

CÓDIGO FUENTE: `figura1-original.c`

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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


int main(){

		struct timespec cgt1,cgt2;
		double ncgt;
		int* R = (int*) malloc(40000*sizeof(int));
		int X1, X2;

		clock_gettime(CLOCK_REALTIME, &cgt1);

		for(int ii=0; ii<40000;ii++){
				X1=0;
				X2=0;

				for(int i=0; i<5000;i++) X1+=2*s[i].a+ii;

				for(int 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));

		//Imprimir resultados:
		printf("Tiempo transcurrido (seg): %11.9f // R[0] = %d // R[39999] =
%d\n", ncgt, R[0], R[39999]);

		free(R);

}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: En lugar de utilizar dos bucles separados para calcular X1 y X2 se ha utilizado uno solo que realiza ambas operaciones. También se ha sustituido el bloque condicional por el operador ternario.

Modificación b) –explicación-: Se ha procedido a desenrollar el bucle que calcula X1 y X2, realizando las operaciones de 4 en 4.

1.1. CÓDIGOS FUENTE MODIFICACIONES

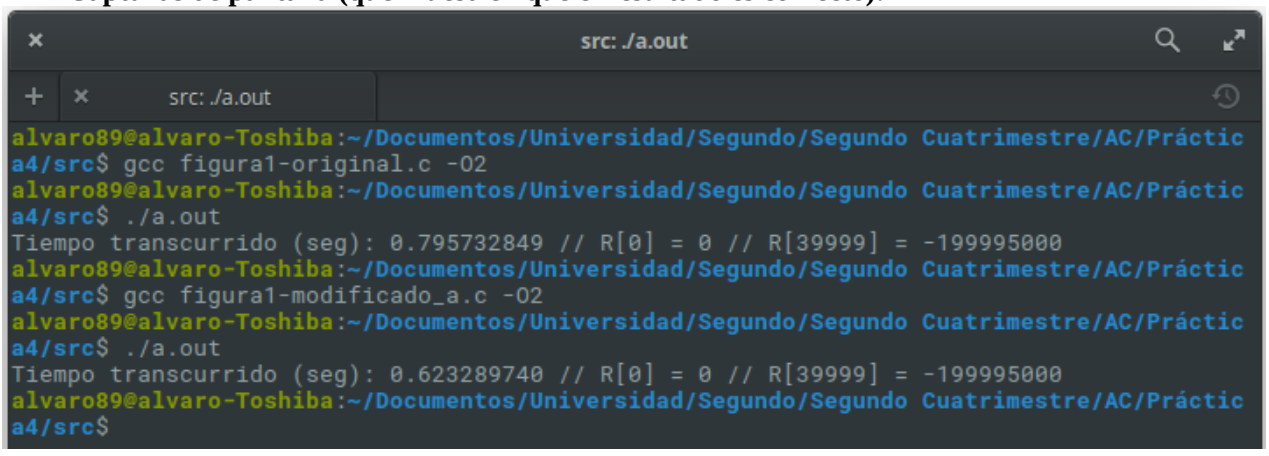
a) `figura1-modificado_a.c`

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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


int main(){

		struct timespec cgt1,cgt2;
		double ncgt;
		int* R = (int*) malloc(40000*sizeof(int));
		int X1, X2;

		clock_gettime(CLOCK_REALTIME, &cgt1);

		for(int ii=0; ii<40000;ii++){
				X1=0;
				X2=0;

				for(int i=0; i<5000;i++){
						X1+=2*s[i].a+ii;
						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));

		//Imprimir resultados:
		printf("Tiempo transcurrido (seg): %11.9f // R[0] = %d // R[39999] =
%d\n", ncgt, R[0], R[39999]);

		free(R);

}
```

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

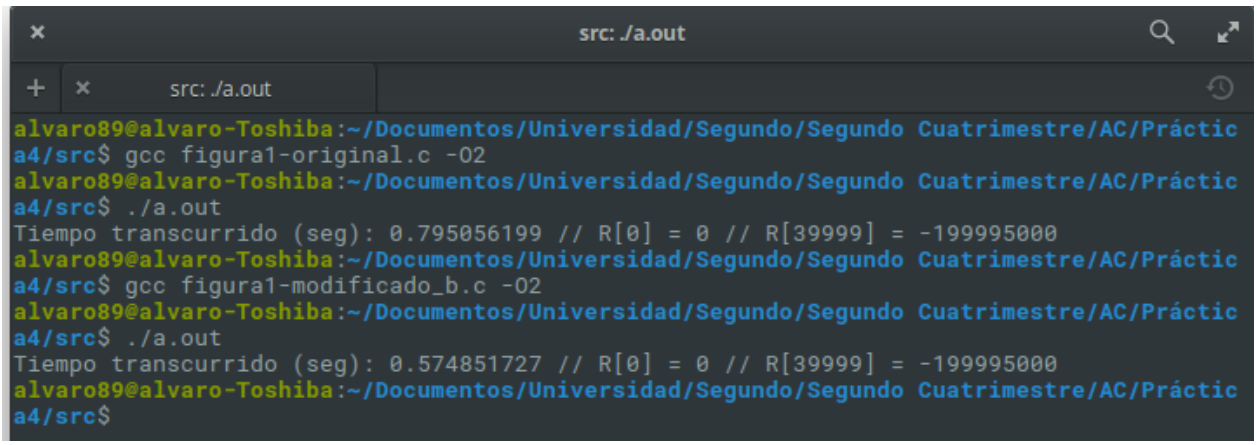


b) figura1-modificado_b.c
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

struct{
		int a;
		int b;
```

```
} s[5000];


int main(){

                struct timespec cgt1,cgt2;
                double ncgt;
                int* R = (int*) malloc(40000*sizeof(int));
                int X1, X2;

                clock_gettime(CLOCK_REALTIME, &cgt1);

                for(int ii=0; ii<40000;ii++){
                                X1=0;
                                X2=0;

                                for(int i = 0; i < 5000; i+=4){
            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;
            X1 += 2*s[i+2].a+ii;
            X2 += 3*s[i+2].b-ii;
            X1 += 2*s[i+3].a+ii;
            X2 += 3*s[i+3].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));

                //Imprimir resultados:
                printf("Tiempo transcurrido (seg): %11.9f // R[0] = %d // R[39999] =
%d\n", ncgt, R[0], R[39999]);

                free(R);

}
```

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



1.1. TIEMPOS:

Modificación	-O2
Sin modificar	0,795056199
Modificación a)	0,623289740
Modificación b)	0,574851727

1.1. COMENTARIOS SOBRE LOS RESULTADOS: Como podemos ver en este caso las mejoras no son tan palpables como en el ejemplo anterior, el hecho de utilizar un único bucle en lugar de dos y la utilización del operador ternario ha hecho que los tiempos se reduzcan en apenas 0,1 segundos. Si además desenrollamos el bucle, el tiempo se reduce otros 0,1 segundos.

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-original.s

```
.L2:
 movl %r10d, %edi
 movl $s, %eax
 xorl %esi, %esi
.L3:
 movl (%rax), %edx
 addq $8, %rax
 leal (%rdi,%rdx,2), %edx
 addl %edx, %esi
 cmpq %rax, %r9
 jne .L3
 movl $s+4, %eax
 xorl %ecx, %ecx
.L4:
 movl (%rax), %edx
 addq $8, %rax
 leal (%rdx,%rdx,2), %edx
 subl %edi, %edx
 addl %edx, %ecx
 cmpq %rax, %r8
 jne .L4
 cmpl %ecx, %esi
 cmovl %esi, %ecx
 movl %ecx, (%rbx,%r10,4)
 addq $1, %r10
 cmpq $40000, %r10
 jne .L2
```

pmm-secuencial-modificado_b.s

```
.L2:
 movl %r10d, %r8d
 movl $s, %eax
 xorl %ecx, %ecx
 xorl %edx, %edx
.L3:
 movl (%rax), %esi
 addq $32, %rax
 leal (%r8,%rsi,2), %esi
 addl %esi, %edx
 movl -28(%rax), %esi
 leal (%rsi,%rsi,2), %edi
 movl -24(%rax), %esi
 subl %r8d, %edi
 leal (%r8,%rsi,2), %esi
 addl %edi, %ecx
 addl %esi, %edx
 movl -20(%rax), %esi
 leal (%rsi,%rsi,2), %esi
 subl %r8d, %esi
 leal (%rsi,%rcx), %edi
 movl -16(%rax), %ecx
 leal (%r8,%rcx,2), %ecx
 addl %ecx, %edx
 movl -12(%rax), %ecx
 leal (%rcx,%rcx,2), %ecx
 subl %r8d, %ecx
 leal (%rcx,%rdi), %esi
 movl -8(%rax), %ecx
 leal (%r8,%rcx,2), %ecx
 addl %ecx, %edx
 movl -4(%rax), %ecx
 leal (%rcx,%rcx,2), %ecx
 subl %r8d, %ecx
 addl %esi, %ecx
 cmpq %rax, %r9
 jne .L3
 cmpl %ecx, %edx
 cmovg %ecx, %edx
 movl %edx, (%rbx,%r10,4)
 addq $1, %r10
 cmpq $40000, %r10
 jne .L2
```

He decidido comparar el programa original con la segunda modificación realizada. Entre las principales diferencias de uno y otro, podemos observar la eliminación de un bucle (En el original tenemos .L2 que representa el bucle externo con 40000 iteraciones y a .L3 y .L4 que representan los bucles internos; por el contrario en la versión modificada solo se presenta .L2 que representa el bucle externo y .L3 que representa al interno). Además el cuerpo del bucle de la versión modificada es mucho más extenso, ya que contiene las 4 operaciones realizadas frente a la única que se realizaba en la primera versión. Por otra parte, con respecto a la comprobación de la condición podemos ver que el compilador había utilizado la operación cmov también en la versión original, así que no hay modificación con respecto a la modificada.

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.

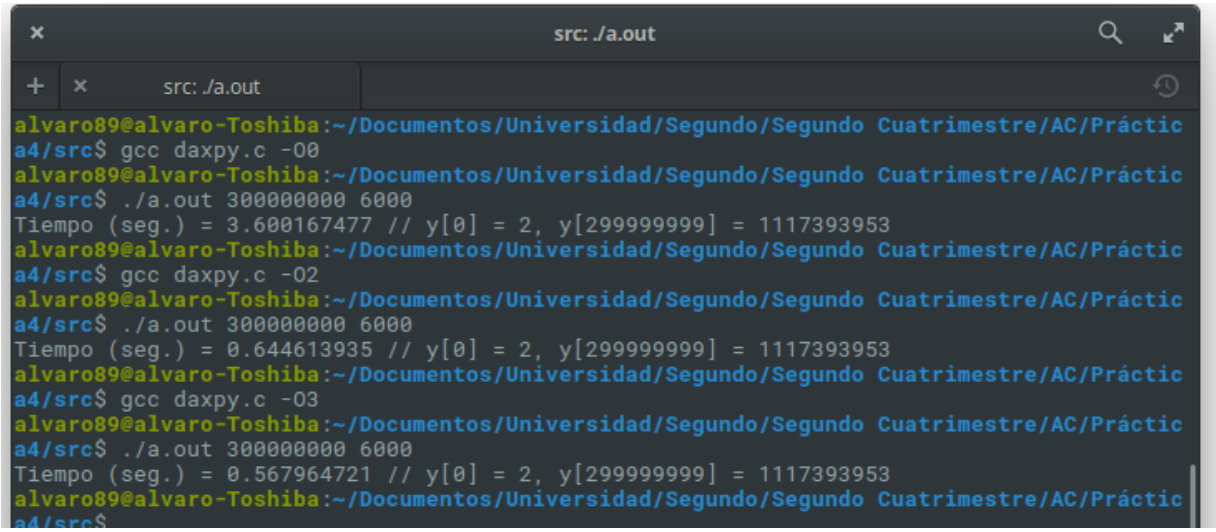
CÓDIGO FUENTE: `daxpy.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

void daxpy(int *y, int *x, int a, unsigned n, struct timespec *cgt1, struct timespec
*cgt2){
    clock_gettime(CLOCK_REALTIME,cgt1);
    unsigned i;
    for (i=0; i<n; i++)
        y[i] += a*x[i];
    clock_gettime(CLOCK_REALTIME,cgt2);
}

int main(int argc, char *argv[]){
    if (argc < 3){
        printf("Modo de empleo %s <tamaño> <a>\n", argv[0]);
        exit(1);
    }

    unsigned n = strtol(argv[1], NULL, 10);
    int a = strtol(argv[2], NULL, 10);
    int *y, *x;
    y = (int*) malloc(n*sizeof(int));
    x = (int*) malloc(n*sizeof(int));
```

```
    unsigned i;
    for (i=0; i<n; i++){
        y[i] = i+2;
        x[i] = i*2;
    }

    struct timespec cgt1,cgt2; double ncgt;

    daxpy(y, x, a, n, &cgt1, &cgt2);

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

    printf("Tiempo (seg.) = %11.9f // y[0] = %i, y[%i] = %i\n", ncgt, y[0], n-1, y[n-
1]);

    free(y);
    free(x);

    return 0;
}
```

Tiempos ejec.	-O0	-O2	-O3
	3,600167477	0,644613935	0,567964721

CAPTURAS DE PANTALLA:



COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:

Una de las primeras diferencias que podemos observar respecto de -O0 con respecto a las demás es que utiliza direcciones de memoria relativas a la pila, mientras que por le contrario, las dos opciones de compilación restantes utilizan directamente registros de la maquina. En el caso de -O2, esto implica obtener un código más reducido ya que nos estamos ahorrando una gran cantidad de operaciones mov, (podemos comprobar que los tiempos se reducen bastante. Por último en la optimización -O3, el compilador ha desenrollado el bucle, obteniendo un código mucho más extenso que las demás versiones. Aunque la verdad las prestaciones no mejoran mucho con respecto a la versión -O2.

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)

daxpyO0.s

```
 movl $0, -4(%rbp)
 jmp .L2
.L3:
 movl -4(%rbp), %eax
 leaq 0(,%rax,4), %rdx
 movq -24(%rbp), %rax
 addq %rax, %rdx
 movl -4(%rbp), %eax
 leaq 0(,%rax,4), %rcx
 movq -24(%rbp), %rax
 addq %rcx, %rax
 movl (%rax), %ecx
 movl -4(%rbp), %eax
 leaq 0(,%rax,4), %rsi
 movq -32(%rbp), %rax
 addq %rsi, %rax
 movl (%rax), %eax
 imull -36(%rbp), %eax
 addl %ecx, %eax
 movl %eax, (%rdx)
 addl $1, -4(%rbp)
.L2:
 movl -4(%rbp), %eax
 cmpl -40(%rbp), %eax
 jb  .L3
 movq -56(%rbp), %rax
 movq %rax, %rsi
```

daxpyO2.s

```
 je .L4
 .p2align 4,,10
 .p2align 3
.L5:
 movl 0(%r13,%rax,4),
%esi
 imull %r12d, %esi
 addl %esi, (%rbx,%rax,4)
 addq $1, %rax
 cmpl %eax, %ebp
 ja .L5
.L4:
 popq %rbx
 .cfi_def_cfa_offset 40
 movq %r14, %rsi
 xorl %edi, %edi
 popq %rbp
 .cfi_def_cfa_offset 32
 popq %r12
 .cfi_def_cfa_offset 24
 popq %r13
 popq %r14
```

daxpyO3.s

```
 testl %r14d, %r14d
 je .L10
 leaq 16(%r12), %rax
 cmpq %rax, %rbx
 leaq 16(%rbx), %rax
 setnb %dl
 cmpq %rax, %r12
 setnb %al
 orb %al, %dl
 je .L3
 cmpl $6, %r14d
 jbe .L3
 movq %rbx, %rax
 andl $15, %eax
 shrq $2, %rax
 negq %rax
 andl $3, %eax
 cmpl %r14d, %eax
 cmova %r14, %rax
 xorl %edx, %edx
 testl %eax, %eax
 je .L4
 movl (%r12), %edx
 imull %r13d, %edx
 addl %edx, (%rbx)
 cmpl $1, %eax
 movl $1, %edx
 je .L4
 movl 4(%r12), %edx
 imull %r13d, %edx
 addl %edx, 4(%rbx)
 cmpl $3, %eax
 movl $2, %edx
 jne .L4
 movl 8(%r12), %edx
 imull %r13d, %edx
 addl %edx, 8(%rbx)
 movl $3, %edx
.L4:
 movl %r14d, %edi
 movl %r13d, 12(%rsp)
 xorl %ecx, %ecx
 subl %eax, %edi
 movd 12(%rsp), %xmm4
 salq $2, %rax
 leal -4(%rdi), %esi
 leaq (%rbx,%rax), %r10
 xorl %r9d, %r9d
 pshufd $0, %xmm4, %xmm2
 addq %r12, %rax
 shrl $2, %esi
 addl $1, %esi
 movdqa %xmm2, %xmm3
 leal 0(,%rsi,4), %r8d
 psrlq 32, %xmm3
.L6:
 movdqu (%rax,%rcx), %xmm0
 addl $1, %r9d
 movdqa %xmm0, %xmm1
 psrlq $32, %xmm0
 pmuludq %xmm3, %xmm0
 pshufd $8, %xmm0, %xmm0
 pmuludq %xmm2, %xmm1
 pshufd $8, %xmm1, %xmm1
 punpckldq %xmm0, %xmm1
 movdqa (%r10,%rcx), %xmm0
 paddd %xmm1, %xmm0
 movaps %xmm0, (%r10,%rcx)
```

		addq $16, %rcx
		cmpl %esi, %r9d
		jb .L6
		addl %r8d, %edx
		cmpl %r8d, %edi
		je .L10
		movl %edx, %eax
		movl (%r12,%rax,4), %ecx
		imull %r13d, %ecx
		addl %ecx, (%rbx,%rax,4)
		leal 1(%rdx), %eax
		cmpl %eax, %r14d
		jbe .L10
		movl (%r12,%rax,4), %ecx
		addl $2, %edx
		imull %r13d, %ecx
		addl %ecx, (%rbx,%rax,4)
		cmpl %edx, %r14d
		jbe .L10
		movl %edx, %eax
		imull (%r12,%rax,4), %r13d
		addl %r13d, (%rbx,%rax,4)
		.L10:
		addq $16, %rsp
		.cfi_remember_state
		.cfi_def_cfa_offset 48
		movq %rbp, %rsi
		xorl %edi, %edi
		popq %rbx
		.cfi_def_cfa_offset 40
		popq %rbp
		.cfi_def_cfa_offset 32
		popq %r12
		.cfi_def_cfa_offset 24
		popq %r13
		.cfi_def_cfa_offset 16
		popq %r14
		.cfi_def_cfa_offset 8