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): Miguel Angel Robles Urquiza
Grupo de prácticas: A1
Fecha de entrega: 01/06/2017
Fecha evaluación en clase: 02/06/2017

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): *Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz*

Sistema operativo utilizado: *Ubuntu 17.04*

Versión de gcc utilizada: *gcc (Ubuntu 6.3.0-12ubuntu2) 6.3.0 20170406*

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.

A) MULTIPLICACIÓN DE MATRICES:
CÓDIGO FUENTE: `pmm-secuencial.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#define MAX 3355 //=2^25

int a[MAX][MAX], b[MAX][MAX], c[MAX][MAX];

int main(int argc, char **argv){
    unsigned i, j, k;

    if(argc < 2)    {
        fprintf(stderr, "falta size\n");
        exit(-1);
    }

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

    if (N>MAX) N=MAX;
```

```
    // Inicializamos las matrices
    for (i=0; i<N; i++)    {
        for (j=0; j<N; j++)        {
            a[i][j] = 0;
            b[i][j] = 2;
            c[i][j] = 2;
        }
    }

    struct timespec cgt1,cgt2; double ncgt;

    clock_gettime(CLOCK_REALTIME,&cgt1);

    // Multiplicacion
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            for (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));

    // Pitamos la primera y la ultima linea de la matriz resultante
    printf("Tiempo = %11.9f\t Primera = %d\t Ultima=%d\n",ncgt,a[0][0],a[N-1]
[N-1]);

    return 0;
}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: Con el fin de que los datos almacenados en memoria estén más próximos, he invertido los bucles con índice “k” y “j”, es decir, he puesto que para cada valor de “k”, se recorran todos los de “j” en vez d que para cada valor de “j” se recorran todos los de “k”

Modificación b) –explicación-: He desenrollado el bucle “k” en bloques de 4

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>

#define MAX 3355 //=2^25

int a[MAX][MAX], b[MAX][MAX], c[MAX][MAX];

int main(int argc, char **argv){
    unsigned i, j, k;

    if(argc < 2)    {
        fprintf(stderr, "falta size\n");
        exit(-1);
    }

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

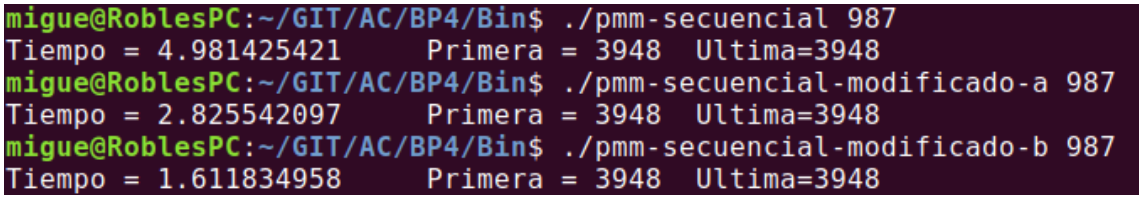
```
    if (N>MAX) N=MAX;

    // Inicializamos las matrices
    for (i=0; i<N; i++)    {
        for (j=0; j<N; j++)        {
            a[i][j] = 0;
            b[i][j] = 2;
            c[i][j] = 2;
        }
    }

    struct timespec cgt1,cgt2; double ncgt;

    clock_gettime(CLOCK_REALTIME,&cgt1);

    // Multiplicacion
    for (i=0; i<N; i++)
        for (k=0; k<N; k++)
            for (j=0; j<N; j++)
                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));

    // Pitamos la primera y la ultima linea de la matriz resultante
    printf("Tiempo = %11.9f\t Primera = %d\t Ultima=%d\n",ncgt,a[0]
[0],a[N-1][N-1]);

    return 0;
}
```

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

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

#define MAX 3355 //=2^25

int a[MAX][MAX], b[MAX][MAX], c[MAX][MAX];

int main(int argc, char **argv){
    unsigned i, j, k;

    int total = 0;
    int h;
    int s1,s2,s3,s4;
    s1=s2=s3=s4=0;

    if(argc < 2)    {
        fprintf(stderr, "falta size\n");
        exit(-1);
    }
```

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

    if (N>MAX) N=MAX;

    // Inicializamos las matrices
    for (i=0; i<N; i++)    {
        for (j=0; j<N; j++)        {
            a[i][j] = 0;
            b[i][j] = 2;
            c[i][j] = 2;
        }
    }

    struct timespec cgt1,cgt2; double ncgt;

    int iterations = N/4;

    clock_gettime(CLOCK_REALTIME,&cgt1);

    // Multiplicacion
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)        {
            s1=s2=s3=s4=0;
            for (h=0, k=0;h < iterations; ++h, k+=4)
            {
                    s1 += (b[i][k]  *c[j][k]  );
                    s2 += (b[i][k+1]*c[j][k+1]);
                    s3 += (b[i][k+2]*c[j][k+2]);
                    s4 += (b[i][k+3]*c[j][k+3]);

            }

            total = s1 + s2 + s3 + s4;

            for(k=iterations*4; k<N; ++k)
                total += (b[i][k]*c[j][k]);

            a[i][j]=total;
        }

    clock_gettime(CLOCK_REALTIME,&cgt2);

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

    // Pitamos la primera y la ultima linea de la matriz resultante
    printf("Tiempo = %11.9f\t Primera = %d\t Ultima=%d\n",ncgt,a[0]
[0],a[N-1][N-1]);

    return 0;
}
```

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



1.1. TIEMPOS:

Modificación	**-O2**
Sin modificar	*4,981425421*
Modificación a)	2,825542097
Modificación b)	1,611834958

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

Vemos que con la mejora a), accediendo por filas se producen menos fallos en caché y por lo tanto el acceso a memoria es más rápido y mejoramos el rendimiento. En b) desenrollando el bucle en iteraciones de 4 en 4, vemos que el código mejora aún mas el tiempo de ejecución. Estos resultados dependen de de la arquitectura en ala que se ejecuta el código.

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

```
            call
            clock_gettime@PLT
.LVL11:
            movq
            %r15, %rax
            movq
            (%rsp), %r8
            movq
            8(%rsp), %r9
            imulq
            $-13420, %r15,
%r15
            negq
            %rax
            imulq
            $13424, %r14,
%r10
            leaq
            -4(,%rax,4), %r14
            addq
            %r9, %r8
            leaq
            -13420(%r15),
%r11
            addq
            %r13, %r10
.LVL12:
.L9:
            leaq
            (%r14,%rbp), %r9
```

pmm-secuencial-modificado_a.s

```
            call
            clock_gettime@PLT
.LVL11:
            leaq
            c(%rip), %rax
            movq
            (%rsp), %r8
            movq
            8(%rsp), %r9
            addq
            %rax, %r15
.LVL12:
.L9:
            .loc 1 22 0
            movq
            %r9, %rcx
            movq
            %r8, %rdi
            .p2align 4,,10
            .p2align 3
.L8:
.LVL13:
            movl
            (%rdi), %esi
            xorl
            %eax, %eax
.LVL14:
            .p2align 4,,10
            .p2align 3
.L7:
```

pmm-secuencial-modificado_b.s

```
call clock_gettime@PLT
.LVL11:
     leal
     0(,%r12,4), %eax
     movq
     %rbx, (%rsp)
     movq
     %rbp, 56(%rsp)
     movl
     %eax, %ecx
     movl
     %eax, 24(%rsp)
     cltq
     movq
     %rax, %rdx
     leaq
     0(,%rax,4), %rdi
     negq
     %rdx
     leaq
     0(,%rdx,4), %rbx
     leal
     -1(%r12), %edx
     leaq
     (%r14,%rdi), %r10
     movq
     %rdi, 8(%rsp)
     salq
     $2, %rdx
     movq
```

```
          .loc 1 24 0
          movq
          %r8, %rdi
          .p2align 4,,10
          .p2align 3
.L8:
.LVL13:
          movl
          (%r9), %esi
          leaq
          (%r11,%rdi), %rax
          movq
          %r12, %rcx
.LVL14:
          .p2align 4,,10
          .p2align 3
.L7:
          .loc 1 42 0
discriminator 2
          movl
          (%rcx), %edx
          addq
          $13420, %rax
          addq
          $4, %rcx
          imull
          -13420(%rax),
%edx
          addl
          %edx, %esi
          .loc 1 41 0
discriminator 2
          cmpq
          %rdi, %rax
          jne
          .L7
          movl
          %esi, (%r9)
          addq
          $4, %r9
          leaq
          4(%rax), %rdi
          .loc 1 40 0
discriminator 2
          cmpq
          %rbp, %r9
          jne
          .L8
          addq
          $13420, %rbp
          addq
          $13420, %r12
          .loc 1 39 0
discriminator 2
          cmpq
          %r10, %rbp
          jne
          .L9
.L10:
          .loc 1 44 0
          leaq
          32(%rsp), %rsi
          xorl
          %edi, %edi
          call
          clock_gettime@PLT
          .loc 1 38 0
discriminator 2
          movl
          (%rcx,%rax), %edx
          imull
          %esi, %edx
          addl
          %edx, (%rbx,%rax)
          addq
          $4, %rax
          .loc 1 37 0
discriminator 2
          cmpq
          %rbp, %rax
          jne
          .L7
.LVL15:
          addq
          $13420, %rcx
          addq
          $4, %rdi
          .loc 1 36 0
discriminator 2
          cmpq
          %r15, %rcx
          jne
          .L8
          addq
          $13420, %rbx
          addq
          $13420, %r8
          .loc 1 35 0
discriminator 2
          cmpq
          %r14, %rbx
          jne
          .L9
.L10:
          .loc 1 40 0
          leaq
          32(%rsp), %rsi
          xorl
          %edi, %edi
          call
          clock_gettime@PLT
     %rbx, 40(%rsp)
     subq
     %rax, %rdx
     movq
     %rdx, %rax
     movl
     28(%rsp), %edx
     leaq
     16(,%rax,4), %rax
     subl
     %ecx, %edx
     movq
     %rax, 48(%rsp)
     leaq
     4(,%rdx,4), %r11
.LVL12:
.L12:
     movq
     40(%rsp), %rax
     .loc 1 27 0
     movq
     16(%rsp), %r13
     leaq
     c(%rip), %rbp
     leaq
     (%r10,%rax), %r14
     movq
     48(%rsp), %rax
     leaq
     (%rax,%r10), %rbx
     .p2align 4,,10
     .p2align 3
.L11:
.LVL13:
     .loc 1 45 0
     testl
     %r12d, %r12d
     je
     .L15
     movq
     %rbp, %rdx
     movq
     %r14, %rax
     xorl
     %edi, %edi
     xorl
     %r8d, %r8d
     xorl
     %r9d, %r9d
     xorl
     %ecx, %ecx
.LVL14:
     .p2align 4,,10
     .p2align 3
.L8:
     .loc 1 46 0
discriminator 2
     movl
     (%rax), %esi
     addq
     $16, %rax
     addq
     $16, %rdx
     imull
     -16(%rdx), %esi
     addl
     %esi, %ecx
.LVL15:
     .loc 1 47 0
```

		discriminator 2 **movl** **-12(%rax), %esi** **imull** **-12(%rdx), %esi** **addl** **%esi, %r9d** **.LVL16:** **.loc 1 48 0** **discriminator 2** **movl** **-8(%rax), %esi** **imull** **-8(%rdx), %esi** **addl** **%esi, %r8d** **.LVL17:** **.loc 1 49 0** **discriminator 2** **movl** **-4(%rax), %esi** **imull** **-4(%rdx), %esi** **addl** **%esi, %edi** **.LVL18:** **.loc 1 45 0** **discriminator 2** **cmpq** **%rbx, %rax** **jne** **.L8** addl %r9d, %ecx .LVL19: addl %ecx, %r8d .LVL20: addl %r8d, %edi .LVL21: .L7: .loc 1 55 0 cmpl 24(%rsp), %r15d jbe .L9 movq 8(%rsp), %rax leaq 0(%rbp,%rax), %rcx .loc 1 55 0 is_stmt 0 discriminator 2 xorl %eax, %eax .LVL22: .p2align 4,,10 .p2align 3 .L10: .loc 1 56 0 is_stmt 1 discriminator 2 movl (%r10,%rax), %edx imull (%rcx,%rax), %edx addq $4, %rax addl %edx, %edi

		.LVL23: .loc 1 55 0 discriminator 2 cmpq %rax, %r11 jne .L10 .LVL24: .L9: .loc 1 58 0 discriminator 2 movl %edi, 0(%r13) addq $13420, %rbp addq $4, %r13 .loc 1 43 0 discriminator 2 cmpq (%rsp), %rbp jne .L11 addq $13420, 16(%rsp) addq $13420, %r10 movq 16(%rsp), %rax .loc 1 42 0 discriminator 2 cmpq 56(%rsp), %rax jne .L12 jmp .L13 .LVL25: .L3: .loc 1 39 0 leaq 64(%rsp), %rsi xorl %edi, %edi call clock_gettime@PLT

B) CÓDIGO FIGURA 1:

CÓDIGO FUENTE: `figura1-original.c`

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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

int main(int argc, char **argv){
    int ii, i, X1, X2;
    int R[40000];

    for (int i = 0; i < 5000; i++) {
```

```
      s[i].a = rand()%8;
      s[i].b = rand()%8;
    }

    struct timespec cgt1,cgt2; double ncgt;

    clock_gettime(CLOCK_REALTIME,&cgt1);

    for (ii = 1; 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("R[0] = %i, R[39999] = %i\n", R[0], R[39999]);
    printf("\nTiempo (seg.) = %11.9f\n", ncgt);

    return 0;
}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: Puesto que los dos bucles van desde 0 hasta 4999, los he agrupado en uno solo y he puesto las sentencias dentro de este. También he agrupado en una sentencia el if/else, utilizando los operadores "?" Y ":".

Modificación b) –explicación-: Partiendo de la optimización anterior, he desenrollado el bucle en iteraciones de 4 en 4

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) figura1-modificado_a.c

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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

int main(int argc, char **argv){
    int ii, i, X1, X2;
    int R[40000];

    struct timespec cgt1,cgt2; double ncgt;

  for (int i = 0; i < 5000/8; i+=8) {
      s[i].a = rand()%8;
      s[i+2].a = rand()%8;
```

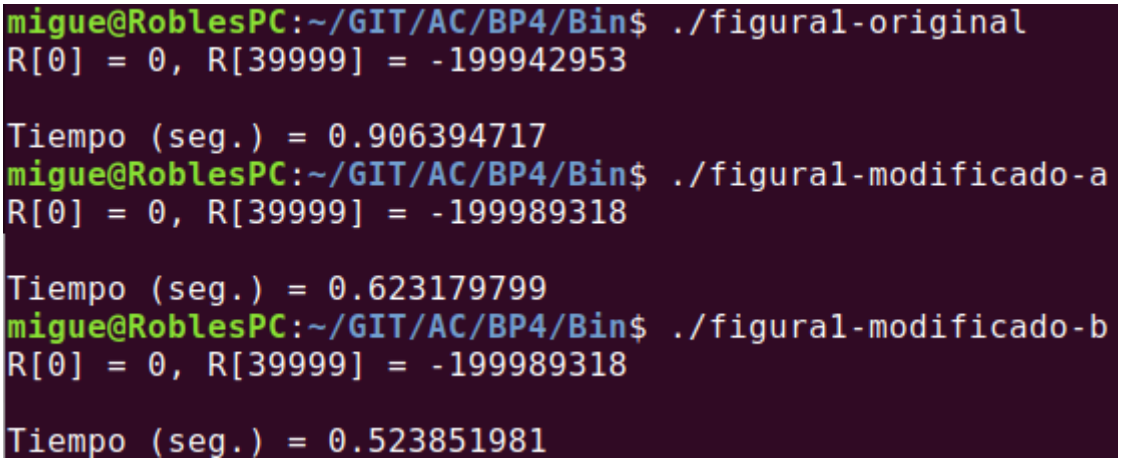
```
      s[i+3].a = rand()%8;
      s[i+4].a = rand()%8;
      s[i+5].a = rand()%8;
      s[i+6].a = rand()%8;
      s[i+7].a = rand()%8;
      s[i+8].a = rand()%8;
      s[i].b = rand()%8;
      s[i+2].b = rand()%8;
      s[i+3].b = rand()%8;
      s[i+4].b = rand()%8;
      s[i+5].b = rand()%8;
      s[i+6].b = rand()%8;
      s[i+7].b = rand()%8;
      s[i+8].b = rand()%8;
    }
    clock_gettime(CLOCK_REALTIME,&cgt1);

    for (ii = 1; ii < 40000; ii++)    {
        X1 = 0; X2 = 0;

        for (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));

    printf("R[0] = %i, R[39999] = %i\n", R[0], R[39999]);
    printf("\nTiempo (seg.) = %11.9f\n", ncgt);

    return 0;
}
```

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

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

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

int main(int argc, char **argv){
    int ii, i, X1, X2;
    int R[40000];

    struct timespec cgt1,cgt2; double ncgt;

  for (int i = 0; i < 5000/8; i+=8) {
      s[i].a = rand()%8;
      s[i+2].a = rand()%8;
      s[i+3].a = rand()%8;
      s[i+4].a = rand()%8;
      s[i+5].a = rand()%8;
```

```
      s[i+6].a = rand()%8;
      s[i+7].a = rand()%8;
      s[i+8].a = rand()%8;
      s[i].b = rand()%8;
      s[i+2].b = rand()%8;
      s[i+3].b = rand()%8;
      s[i+4].b = rand()%8;
      s[i+5].b = rand()%8;
      s[i+6].b = rand()%8;
      s[i+7].b = rand()%8;
      s[i+8].b = rand()%8;
    }
    clock_gettime(CLOCK_REALTIME,&cgt1);

    for (ii = 1; ii < 40000; ii++)    {
        X1 = 0; X2 = 0;

        for (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));

    printf("R[0] = %i, R[39999] = %i\n", R[0], R[39999]);
    printf("\nTiempo (seg.) = %11.9f\n", ncgt);

    return 0;
}
```

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



1.1. TIEMPOS:

Modificación	**-O2**
Sin modificar	*0,906394717*
Modificación a)	0,623179799
Modificación b)	0,523851981

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

Vemos que con la mejora a) optimizamos el tiempo de ejecución del programa puesto que tenemos que realizar 49999 iteraciones menos. La mejora b) deriva de la a) y al desenrollar el bucle vemos que el tiempo de ejecución también se reduce.

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.c

```
call  clock_gettime@PLT
.LVL4:
      leaq
      40004+s(%rip), %r8
      leaq
      48(%rsp), %r10
      movl
      $1, %r9d
.LVL5:
      .p2align 4,,10
      .p2align 3
.L7:
      movl
      %r9d, %edi
.LVL6:
      movq
      %rbx, %rax
      xorl
      %esi, %esi
.LVL7:
      .p2align 4,,10
      .p2align 3
.L3:
      .loc 1 27 0
discriminator 2
      movl
      (%rax), %edx
      addq
      $8, %rax
      leal
      (%rdi,%rdx,2), %edx
      addl
      %edx, %esi
.LVL8:
      .loc 1 26 0
discriminator 2
      cmpq
      %rbp, %rax
      jne
      .L3
      leaq
      4+s(%rip), %rax
      .loc 1 26 0 is_stmt 0
      xorl
      %ecx, %ecx
      .p2align 4,,10
```

figura1-modificado_a.c

```
      call
      clock_gettime@PLT
.LVL18:
      leaq
      40000+s(%rip), %r8
      leaq
      48(%rsp), %r10
      movl
      $1, %r9d
.LVL19:
      .p2align 4,,10
      .p2align 3
.L4:
      movl
      %r9d, %edi
.LVL20:
      movq
      %rbx, %rax
      xorl
      %esi, %esi
      xorl
      %ecx, %ecx
.LVL21:
      .p2align 4,,10
      .p2align 3
.L3:
      .loc 1 42 0
discriminator 2
      movl
      (%rax), %edx
      addq
      $8, %rax
      leal
      (%rdi,%rdx,2), %edx
      addl
      %edx, %ecx
.LVL22:
      .loc 1 43 0
discriminator 2
      movl
      -4(%rax), %edx
      leal
      (%rdx,%rdx,2), %edx
      subl
      %edi, %edx
      addl
```

figura1-modificado_b.c

```
      call
      clock_gettime@PLT
.LVL18:
      leaq
      40000+s(%rip), %rdi
      leaq
      48(%rsp), %r9
      movl
      $1, %r8d
.LVL19:
      .p2align 4,,10
      .p2align 3
.L4:
      movl
      %r8d, %edx
.LVL20:
      movq
      %rbx, %rax
      xorl
      %ecx, %ecx
      xorl
      %r10d, %r10d
.LVL21:
      .p2align 4,,10
      .p2align 3
.L3:
      .loc 1 42 0
discriminator 2
      movl
      (%rax), %esi
      addq
      $32, %rax
      leal
      (%rdx,%rsi,2), %r11d
      .loc 1 43 0
discriminator 2
      movl
      -28(%rax), %esi
      .loc 1 42 0
discriminator 2
      addl
      %r11d, %r10d
.LVL22:
      .loc 1 43 0
discriminator 2
      leal
```

```
      .p2align 3
.L4:
.LVL9:
      .loc 1 30 0 is_stmt 1
discriminator 2
      movl
      (%rax), %edx
      addq
      $8, %rax
      leal
      (%rdx,%rdx,2), %edx
      subl
      %edi, %edx
      addl
      %edx, %ecx
.LVL10:
      .loc 1 29 0
discriminator 2
      cmpq
      %r8, %rax
      jne
      .L4
      .loc 1 33 0
      cmpl
      %ecx, %esi
      cmovl
      %esi, %ecx
.LVL11:
      movl
      %ecx, (%r10,%r9,4)
.LVL12:
      addq
      $1, %r9
      .loc 1 23 0
      cmpq
      $40000, %r9
      jne
      .L7
      .loc 1 38 0
      leaq
      32(%rsp), %rsi
.LVL13:
      xorl
      %edi, %edi
.LVL14:
      call
      clock_gettime@PLT
```

```
      %edx, %esi
.LVL23:
      .loc 1 41 0
discriminator 2
      cmpq
      %rax, %r8
      jne
      .L3
      .loc 1 46 0
      cmpl
      %esi, %ecx
      cmovg
      %esi, %ecx
.LVL24:
      movl
      %ecx, (%r10,%r9,4)
.LVL25:
      addq
      $1, %r9
      .loc 1 38 0
      cmpq
      $40000, %r9
      jne
      .L4
      .loc 1 50 0
      leaq
      32(%rsp), %rsi
.LVL26:
      xorl
      %edi, %edi
.LVL27:
      call
      clock_gettime@PLT
```

```
      (%rsi,%rsi,2), %esi
      subl
      %edx, %esi
      addl
      %esi, %ecx
.LVL23:
      .loc 1 44 0
discriminator 2
      movl
      -24(%rax), %esi
      leal
      (%rdx,%rsi,2), %r11d
      .loc 1 45 0
discriminator 2
      movl
      -20(%rax), %esi
      .loc 1 44 0
discriminator 2
      addl
      %r10d, %r11d
.LVL24:
      .loc 1 45 0
discriminator 2
      leal
      (%rsi,%rsi,2), %esi
      subl
      %edx, %esi
      addl
      %ecx, %esi
.LVL25:
      .loc 1 46 0
discriminator 2
      movl
      -16(%rax), %ecx
      leal
      (%rdx,%rcx,2), %r10d
      .loc 1 47 0
discriminator 2
      movl
      -12(%rax), %ecx
      .loc 1 46 0
discriminator 2
      addl
      %r10d, %r11d
.LVL26:
      .loc 1 47 0
discriminator 2
      leal
      (%rcx,%rcx,2), %ecx
      subl
      %edx, %ecx
      addl
      %ecx, %esi
.LVL27:
      .loc 1 48 0
discriminator 2
      movl
      -8(%rax), %ecx
      leal
      (%rdx,%rcx,2), %r10d
      .loc 1 49 0
discriminator 2
      movl
      -4(%rax), %ecx
      .loc 1 48 0
discriminator 2
      addl
      %r11d, %r10d
.LVL28:
```

		.loc 1 49 0 **discriminator 2** **leal** **(%rcx,%rcx,2), %ecx** **subl** **%edx, %ecx** **addl** **%esi, %ecx** **.LVL29:** **.loc 1 41 0** **discriminator 2** **cmpq** **%rax, %rdi** **jne** **.L3** **.loc 1 52 0** **cmpl** **%ecx, %r10d** **cmovle** **%r10d, %ecx** **.LVL30:** **movl** **%ecx, (%r9,%r8,4)** **.LVL31:** **addq** **$1, %r8** **.loc 1 38 0** **cmpq** **$40000, %r8** **jne** **.L4** .loc 1 55 0 leaq 32(%rsp), %rsi .LVL32: xorl %edi, %edi call clock_gettime@PLT

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.

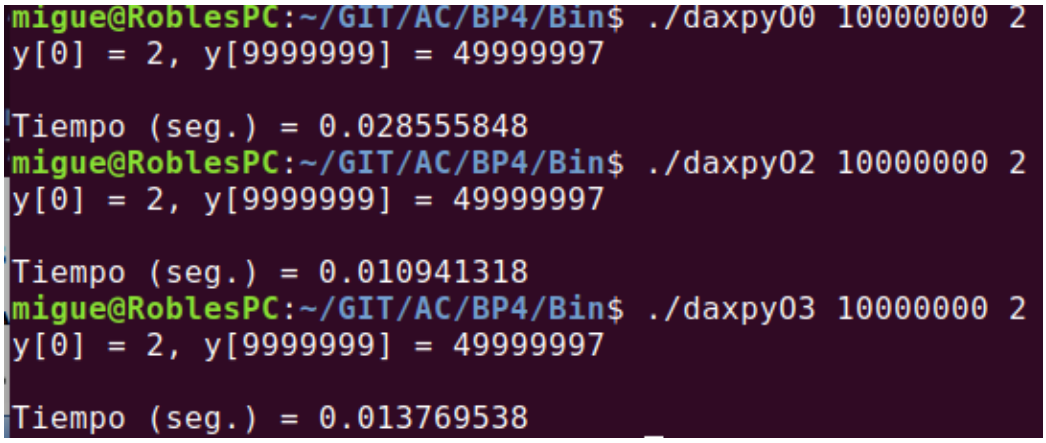
CÓDIGO FUENTE: `daxpy.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

```
#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)    {
        fprintf(stderr, "ERROR: Falta tamanio del vector y
constante\n");
        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("y[0] = %i, y[%i] = %i\n", y[0], n-1, y[n-1]);
    printf("\nTiempo (seg.) = %11.9f\n", ncgt);

    free(y);
    free(x);

    return 0;
}
```

Tiempos ejec.	**-O0**	**-O2**	**-O3**
	0,028555848	0,010941318	0,013769538

CAPTURAS DE PANTALLA:



COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:
Usamos 3 optimizaciones diferentes para generar este código ensamblador.

-O0 utilizamos direcciones relativas a pila
-O2 utilizamos registros de la arquitectura
-O3 desenrollamos del bucle

Con O2 se ahorran instrucciones y por eso el código es más corto que con O0
Con O3, puesto que el código de los bucles está desarrollado, el código es mucho mas largo

La tabla de tiempos de ejecución nos dice que es más rápida la complicación O2 que las otras dos

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	daxpyO2.s	daxpyO3.s
call clock_gettime@PLT .loc 1 8 0 movl $0, -4(%rbp) **.L3:** **.loc 1 8 0 is_stmt 0** **discriminator 3** **movl** **-4(%rbp), %eax** **cmpl** **-40(%rbp), %eax** **jnb** **.L2** **.loc 1 9 0 is_stmt 1** **discriminator 2** **movl** **-4(%rbp), %eax** **leaq** **0(,%rax,4), %rdx** **movq** **-24(%rbp), %rax** **addq**	call clock_gettime@PLT .LVL3: xorl %eax, %eax .loc 1 8 0 testl %ebp, %ebp je .L3 .LVL4: .p2align 4,,10 .p2align 3 **.L5:** **.loc 1 9 0 discriminator** **2** **movl** **0(%r13,%rax,4), %esi** **imull** **%r12d, %esi** **addl** **%esi, (%rbx,%rax,4)** **.LVL5:**	callclock_gettime@PLT .LVL3: .loc 1 8 0 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

```
     %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)
     .loc 1 8 0
discriminator 2
     addl
     $1, -4(%rbp)
     jmp
     .L3
.L2:
     .loc 1 10 0
     movq
     -56(%rbp), %rax
     movq
     %rax, %rsi
     movl
     $0, %edi
     call
     clock_gettime@PLT
```

```
   addq
   $1, %rax
.LVL6:
   .loc 1 8 0 discriminator
2
   cmpl
   %eax, %ebp
   ja
   .L5
.L3:
   .loc 1 11 0
   popq
   %rbx
   .cfi_def_cfa_offset 40
.LVL7:
   .loc 1 10 0
   movq
   %r14, %rsi
   xorl
   %edi, %edi
   .loc 1 11 0
   popq
   %rbp
   .cfi_def_cfa_offset 32
.LVL8:
   popq
   %r12
   .cfi_def_cfa_offset 24
.LVL9:
   popq
   %r13
   .cfi_def_cfa_offset 16
.LVL10:
   popq
   %r14
   .cfi_def_cfa_offset 8
.LVL11:
   .loc 1 10 0
   jmp
   clock_gettime@PLT
```

```
    cmpl
    $6, %r14d
    jbe
    .L3
    movq
    %rbx, %rax
    xorl
    %edx, %edx
    shrq
    $2, %rax
    negq
    %rax
    andl
    $3, %eax
    je
    .L4
    .loc 1 9 0
    movl
    (%r12), %edx
    imull
    %r13d, %edx
    addl
    %edx, (%rbx)
.LVL4:
    cmpl
    $1, %eax
    .loc 1 8 0
    movl
    $1, %edx
    je
    .L4
    .loc 1 9 0
    movl
    4(%r12), %edx
    imull
    %r13d, %edx
    addl
    %edx, 4(%rbx)
.LVL5:
    cmpl
    $2, %eax
    .loc 1 8 0
    movl
    $2, %edx
    je
    .L4
    .loc 1 9 0
    movl
    8(%r12), %edx
    imull
    %r13d, %edx
    addl
    %edx, 8(%rbx)
.LVL6:
    .loc 1 8 0
    movl
    $3, %edx
.LVL7:
.L4:
    movl
    %r14d, %edi
    movl
    %r13d, 12(%rsp)
    xorl
    %ecx, %ecx
    subl
    %eax, %edi
    movd
    12(%rsp), %xmm4
```

		movl %eax, %eax leal -4(%rdi), %esi salq $2, %rax xorl %r9d, %r9d pshufd $0, %xmm4, %xmm2 leaq (%rbx,%rax), %r10 shrl $2, %esi addq %r12, %rax addl $1, %esi movdqa %xmm2, %xmm3 leal 0(,%rsi,4), %r8d psrlq $32, %xmm3 .LVL8: **.L7:** **.loc 1 9 0 discriminator 2** **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** **%r9d, %esi** **ja** **.L7** addl %r8d, %edx cmpl %r8d, %edi je .L10 .LVL9: .loc 1 9 0 is_stmt 0 movl %edx, %eax movl

		(%r12,%rax,4), %ecx imull %r13d, %ecx addl %ecx, (%rbx,%rax,4) .loc 1 8 0 is_stmt 1 leal 1(%rdx), %eax .LVL10: cmpl %eax, %r14d jbe .L10 .loc 1 9 0 movl (%r12,%rax,4), %ecx .loc 1 8 0 addl $2, %edx .loc 1 9 0 imull %r13d, %ecx addl %ecx, (%rbx,%rax,4) .LVL11: .loc 1 8 0 cmpl %edx, %r14d jbe .L10 .loc 1 9 0 movl %edx, %eax imull (%r12,%rax,4), %r13d addl %r13d, (%rbx,%rax,4) .LVL12: .L10: .loc 1 11 0 addq $16, %rsp .cfi_remember_state .cfi_def_cfa_offset 48 .loc 1 10 0 movq %rbp, %rsi xorl %edi, %edi .loc 1 11 0 popq %rbx .cfi_def_cfa_offset 40 .LVL13: popq %rbp .cfi_def_cfa_offset 32 .LVL14: popq %r12 .cfi_def_cfa_offset 24 .LVL15: popq %r13 .cfi_def_cfa_offset 16 popq %r14 .cfi_def_cfa_offset 8 .LVL16:

		.loc 1 10 0 jmp clock_gettime@PLT