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): Ernesto Serrano Collado
Grupo de prácticas: A2
Fecha de entrega: 02/06/2016
Fecha evaluación en clase: 03/06/2016

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): `AMD Athlon(tm) X2 Dual-Core QL-60`

Sistema operativo utilizado: `Linux ubuntu 3.19.0-59-generic #65~14.04.1-Ubuntu SMP Tue Apr 19 18:57:09 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux`

Versión de gcc utilizada: `gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4`

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)

```c
#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-:

Simplemente he invertido los bucles j y k, al estar así mas proximos los datos en memoria

Modificación b) –explicación-:

He desenrrollado el bucle k en bloques de 8

1.1. CÓDIGOS FUENTE MODIFICACIONES

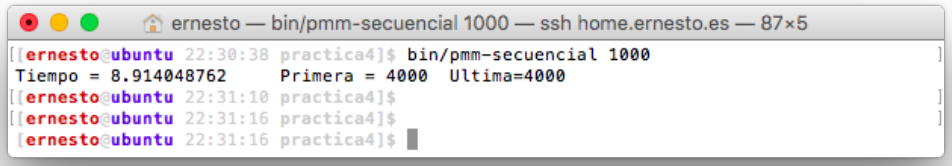
a) `pmm-secuencial-modificado-a.c`

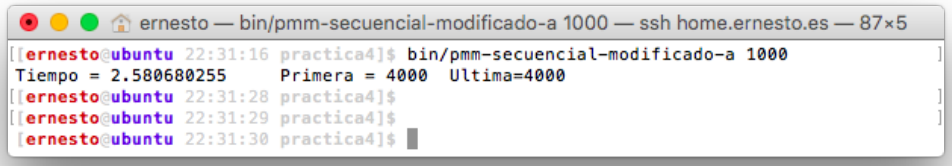
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

```
#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;
}
```

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



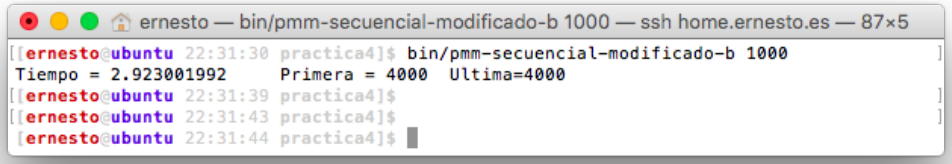


b) `pmm-secuencial-modificado_b.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

#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,s5,s6,s7,s8;
    s1=s2=s3=s4=s5=s6=s7=s8=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/8;

    clock_gettime(CLOCK_REALTIME,&cgt1);

    // Multiplicacion
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
        {
            s1=s2=s3=s4=s5=s6=s7=s8=0;
            for (h=0, k=0;h < iterations; ++h, k+=8)
            {
                    s1 += (b[i][k]  *c[k][j]  );
                    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]);
                    s5 += (b[i][k+4]*c[j][k+4]);
                    s6 += (b[i][k+5]*c[j][k+5]);
                    s7 += (b[i][k+6]*c[j][k+6]);
                    s8 += (b[i][k+7]*c[j][k+7]);
            }

            total = s1 + s2 + s3 + s4 + s5 + s6 + s7 + s8;
            a[i][j]=total;

            for(k=iterations*8; 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	*8.914048762*
Modificación a)	2.580680255
Modificación b)	2.923001992

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

Con una operación tan sencilla como cambiar el orden de los bucles se puede optimizar incluso mas que con un desenrollado de bucle, hay que tener muy en cuenta la arquitectura donde se ejecuta para poder hacer estas optimizaciones, pero se puede llegar a conseguir muy buen rendimiento.

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

```
   leaq  b+4(,%r13,4), %r11
   xorl  %r9d, %r9d
.L14:
   leaq  (%r11,%r9), %rdi
   leaq  b(%r9), %r10
   xorl  %r8d, %r8d
   .p2align 4,,10
   .p2align 3
.L12:
   movl  a(%r9,%r8), %esi
   leaq  c(%r8), %rcx
   movq  %r10, %rax
   .p2align 4,,10
   .p2align 3
.L10:
   movl  (%rax), %edx
   addq  $4, %rax
   addq  $13420, %rcx
   imull -13420(%rcx), %edx
   addl  %edx, %esi
   cmpq  %rdi, %rax
   jne   .L10
   movl  %esi, a(%r9,%r8)
   addq  $4, %r8
   cmpq  %rbp, %r8
```

pmm-secuencial-modificado_b.s

```
   leaq  b+4(,%r13,4), %r10
   xorl  %edx, %edx
.L14:
   leaq  (%r10,%rdx), %r9
   leaq  b(%rdx), %r8
   xorl  %esi, %esi
   .p2align 4,,10
   .p2align 3
.L12:
   movl  (%r8), %edi
   xorl  %eax, %eax
   .p2align 4,,10
   .p2align 3
.L10:
   movl  c(%rsi,%rax), %ecx
   imull %edi, %ecx
   addl  %ecx, a(%rdx,%rax)
   addq  $4, %rax
   cmpq  %rbx, %rax
   jne   .L10
   addq  $4, %r8
   addq  $13420, %rsi
   cmpq  %r9, %r8
   jne   .L12
   addq  $13420, %rdx
```

pmm-secuencial-modificado_c.s

```
   movl  %ebx, %eax
   leal  0(,%rbx,8), %ebx
   movq  $0, 16(%rsp)
   subl  $1, %eax
   movl  $0, 56(%rsp)
   addq  $1, %rax
   movl  %ebx, 36(%rsp)
   imulq $107360, %rax, %rax
   movq  %rax, 40(%rsp)
.L17:
   movl  56(%rsp), %r14d
   movq  16(%rsp), %rax
   xorl  %r13d, %r13d
   movq  $c+4, (%rsp)
   addq  $b, %rax
   imulq $3355, %r14, %r14
   movq  %rax, 48(%rsp)
   .p2align 4,,10
   .p2align 3
.L15:
   movl  32(%rsp), %eax
   movl  %r13d, 8(%rsp)
   testl %eax, %eax
   je .L18
   movl  %r15d, 12(%rsp)
```

```
   jne   .L12
   addq  $13420, %r9
   cmpq  %r12, %r9
   jne   .L14
.L11:
   leaq  16(%rsp), %rsi
   xorl  %edi, %edi
```

```
   cmpq  %r12, %rdx
   jne   .L14
.L11:
   leaq  16(%rsp), %rsi
   xorl  %edi, %edi
```

```
   movq  (%rsp), %rdx
   xorl  %ecx, %ecx
   movq  48(%rsp), %rax
   movq  40(%rsp), %r15
   xorl  %r12d, %r12d
   xorl  %ebp, %ebp
   xorl  %ebx, %ebx
   xorl  %r11d, %r11d
   xorl  %r10d, %r10d
   xorl  %r9d, %r9d
   xorl  %r8d, %r8d
   xorl  %edi, %edi
   .p2align 4,,10
   .p2align 3
.L13:
   movl  (%rax), %esi
   addq  $32, %rdx
   addq  $32, %rax
   imull c(%rcx,%r13,4),
%esi
   addq  $107360, %rcx
   addl  %esi, %edi
   movl  -28(%rax), %esi
   imull -32(%rdx), %esi
   addl  %esi, %r8d
   movl  -24(%rax), %esi
   imull -28(%rdx), %esi
   addl  %esi, %r9d
   movl  -20(%rax), %esi
   imull -24(%rdx), %esi
   addl  %esi, %r10d
   movl  -16(%rax), %esi
   imull -20(%rdx), %esi
   addl  %esi, %r11d
   movl  -12(%rax), %esi
   imull -16(%rdx), %esi
   addl  %esi, %ebx
   movl  -8(%rax), %esi
   imull -12(%rdx), %esi
   addl  %esi, %ebp
   movl  -4(%rax), %esi
   imull -8(%rdx), %esi
   addl  %esi, %r12d
   cmpq  %r15, %rcx
   jne   .L13
   addl  %r8d, %edi
   movl  12(%rsp), %r15d
   addl  %r9d, %edi
   addl  %r10d, %edi
   addl  %r11d, %edi
   addl  %edi, %ebx
   addl  %ebx, %ebp
   addl  %ebp, %r12d
.L12:
   movl  36(%rsp), %eax
   cmpl  %eax, %r15d
   jbe   .L9
   movl  8(%rsp), %esi
   imulq $3355, %rsi, %rsi
   .p2align 4,,10
   .p2align 3
.L10:
   movl  %eax, %edx
   addl  $1, %eax
   leaq  (%rsi,%rdx), %rcx
   leaq  (%r14,%rdx), %rdi
   movl  b(,%rdi,4), %edx
   imull c(,%rcx,4), %edx
   addl  %edx, %r12d
```

		cmpl %r15d, %eax
		jne .L10
		.L9:
		movq 16(%rsp), %rax
		addq $13420, (%rsp)
		movl %r12d, a(%rax,
		%r13,4)
		addq $1, %r13
		cmpq 24(%rsp), %r13
		jne .L15
		addl $1, 56(%rsp)
		addq $13420, 16(%rsp)
		cmpl %r15d, 56(%rsp)
		jne .L17
		jmp .L14
		.L24:
		leaq 64(%rsp), %rsi
		xorl %edi, %edi

B) CÓDIGO FIGURA 1:

CÓDIGO FUENTE: `figura1-original.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

    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-:

He agrupado los dos for en uno solo, ya que hacen lo mismo y se reduce a la mitad, además he utilizado el operador ternario para asignar los valores en lugar de un if/else

Modificación b) –explicación-:

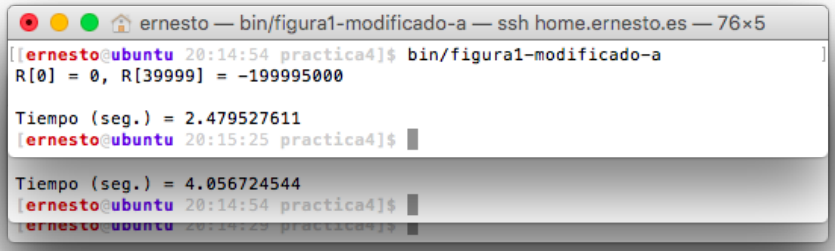
En este caso también he agrupado los dos for en uno solo, y lo he desenrrollado en operaciones de 5, al igual que en la anterior he utilizado el operador ternario

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) figura1**`-modificado-a.c`**
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

    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;
}
```

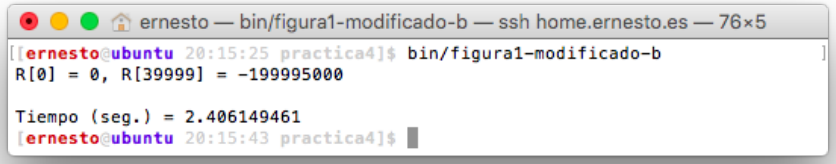
Capturas de pantalla (que muestren que el resultado es correcto):



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

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

    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	*4.056724544*
Modificación a)	2.479527611
Modificación b)	2.406149461

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

El reducir el numero de bucles a la mitad evidentemente reducirá a la mitad la ejecución, en cambio el reducir el numero de iteraciones al desenrollar aunque se gana algo de velocidad no se nota de igual manera

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`

```
   leaq  52(%rsp), %r8
   movl  $1, %ecx
   .p2align 4,,10
   .p2align 3
.L2:
   movl  $s, %eax
   xorl  %esi, %esi
   .p2align 4,,10
   .p2align 3
.L9:
   movl  (%rax), %edx
   addq  $8, %rax
   leal  (%rcx,%rdx,2), %edx
   addl  %edx, %esi
   cmpq  $s+40000, %rax
   jne   .L9
   movl  $s+4, %eax
   xorl  %edi, %edi
   .p2align 4,,10
   .p2align 3
.L5:
   movl  (%rax), %edx
   addq  $8, %rax
   leal  (%rdx,%rdx,2), %edx
   subl  %ecx, %edx
   addl  %edx, %edi
```

`figura1-modificado-a.s`

```
   leaq  52(%rsp), %r8
   movl  $1, %ecx
   .p2align 4,,10
   .p2align 3
.L2:
   movl  $s, %eax
   xorl  %edi, %edi
   xorl  %esi, %esi
   .p2align 4,,10
   .p2align 3
.L5:
   movl  (%rax), %edx
   addq  $8, %rax
   leal  (%rcx,%rdx,2),
%edx
   addl  %edx, %esi
   movl  -4(%rax), %edx
   leal  (%rdx,%rdx,2),
%edx
   subl  %ecx, %edx
   addl  %edx, %edi
   cmpq  $s+40000, %rax
   jne   .L5
   cmpl  %edi, %esi
   cmovg %edi, %esi
```

`figura1-modificado-b.s`

```
   leaq  52(%rsp), %r11
   movl  $1, %edx
   .p2align 4,,10
   .p2align 3
.L2:
   movl  $s, %eax
   xorl  %ecx, %ecx
   xorl  %r10d, %r10d
   .p2align 4,,10
   .p2align 3
.L5:
   movl  4(%rax), %r9d
   movl  (%rax), %esi
   addq  $32, %rax
   leal  (%r9,%r9,2), %r8d
   movl  -20(%rax), %r9d
   leal  (%rdx,%rsi,2), %esi
   subl  %edx, %r8d
   addl  %esi, %r10d
   movl  -24(%rax), %esi
   leal  (%r9,%r9,2), %edi
   addl  %r8d, %ecx
   movl  -12(%rax), %r9d
   subl  %edx, %edi
   leal  (%rdx,%rsi,2), %esi
```

```
   cmpq  $s+40004, %rax
   jne   .L5
   cmpl  %esi, %edi
   jle   .L6
   movl  %esi, (%r8)
.L7:
   addl  $1, %ecx
   addq  $4, %r8
   cmpl  $40001, %ecx
   jne   .L2
   leaq  32(%rsp), %rsi
   xorl  %edi, %edi
```

```
   addl  $1, %ecx
   addq  $4, %r8
   movl  %esi, -4(%r8)
   cmpl  $40001, %ecx
   jne   .L2
   leaq  32(%rsp), %rsi
   xorl  %edi, %edi
```

```
   addl  %ecx, %edi
   movl  -16(%rax), %ecx
   addl  %esi, %r10d
   leal  (%r9,%r9,2), %esi
   movl  -4(%rax), %r9d
   leal  (%rdx,%rcx,2), %ecx
   subl  %edx, %esi
   addl  %edi, %esi
   addl  %ecx, %r10d
   movl  -8(%rax), %ecx
   leal  (%rdx,%rcx,2), %ecx
   addl  %ecx, %r10d
   leal  (%r9,%r9,2), %ecx
   subl  %edx, %ecx
   addl  %esi, %ecx
   cmpq  $s+40000, %rax
   jne   .L5
   cmpl  %ecx, %r10d
   cmovle      %r10d, %ecx
   addl  $1, %edx
   addq  $4, %r11
   movl  %ecx, -4(%r11)
   cmpl  $40001, %edx
   jne   .L2
   leaq  32(%rsp), %rsi
   xorl  %edi, %edi
```

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.

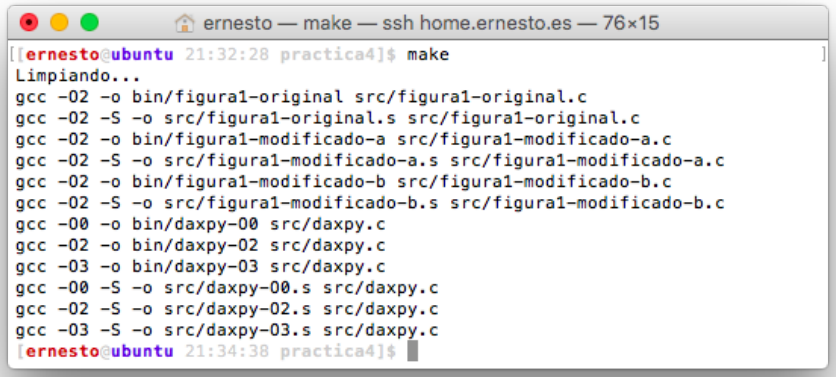
CÓDIGO FUENTE: `daxpy.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

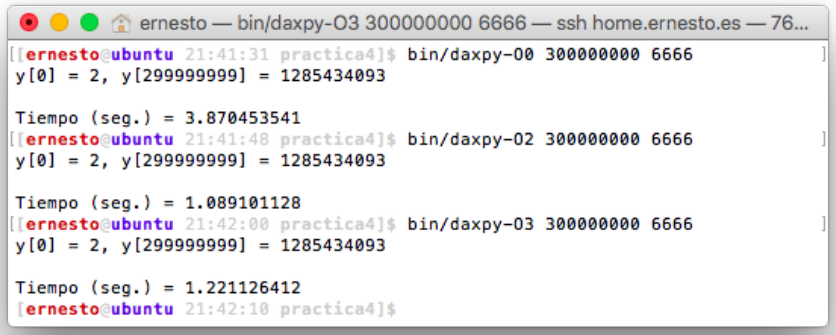
```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

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 tam 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
	3.870453541	1.089101128	1.221126412

CAPTURAS DE PANTALLA:





COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:

Con O0 usamos direcciones relativas a la pila y con O2, registros de la arquitectura. Así ahorramos muchas operaciones move innecesarias y como se puede ver abajo obtenemos un código mucho más reducido para O2 que para O0, donde estamos moviendo a registros de la arquitectura direcciones relativas a la pila y operando con esas direcciones.

Por último, en O3, el compilador ha hecho un desenrollado del bucle, dándonos un código mas largo, y en este caso un poco mas lento que 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

```
   xorl  %eax, %eax
   testl %ebp, %ebp
   je .L3
   .p2align 4,,10
   .p2align 3
.L5:
   movl  (%r12,%rax,4), %edi
   imull %r13d, %edi
   addl  %edi, (%rbx,%rax,4)
   addq  $1, %rax
   cmpl  %eax, %ebp
   ja .L5
.L3:
   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
   movq  %r14, %rsi
   xorl  %edi, %edi
   popq  %r14
   .cfi_def_cfa_offset 8
```

daxpyO3.s

```
   testl %r12d, %r12d
   je .L8
   leaq  16(%rbx), %rax
   cmpq  %rax, %rbp
   leaq  16(%rbp), %rax
   setnb %dl
   cmpq  %rax, %rbx
   setnb %al
   orb   %al, %dl
   je .L3
   cmpl  $4, %r12d
   jbe   .L3
   movl  %r13d, 12(%rsp)
   movl  %r12d, %esi
   xorl  %eax, %eax
   movd  12(%rsp), %xmm6
   shrl  $2, %esi
   xorl  %edx, %edx
   leal  0(,%rsi,4), %ecx
   pshufd   $0, %xmm6, %xmm2
   movdqa   %xmm2, %xmm4
   psrlq $32, %xmm4
.L9:
   movdqu   0(%rbp,%rax), %xmm1
   addl  $1, %edx
   movdqa   %xmm1, %xmm5
   psrlq $32, %xmm1
```

`movl $0, %edi`		`pmuludq %xmm4, %xmm1`
		`pshufd $8, %xmm1, %xmm1`
		`movdqu (%rbx,%rax), %xmm3`
		`pmuludq %xmm2, %xmm5`
		`pshufd $8, %xmm5, %xmm0`
		`punpckldq %xmm1, %xmm0`
		`paddd %xmm3, %xmm0`
		`movdqu %xmm0, (%rbx,%rax)`
		`addq $16, %rax`
		`cmpl %esi, %edx`
		`jb .L9`
		`cmpl %ecx, %r12d`
		`je .L8`
		`movl %ecx, %eax`
		`movl 0(%rbp,%rax,4), %edx`
		`imull %r13d, %edx`
		`addl %edx, (%rbx,%rax,4)`
		`leal 1(%rcx), %eax`
		`cmpl %eax, %r12d`
		`jbe .L8`
		`movl 0(%rbp,%rax,4), %edx`
		`addl $2, %ecx`
		`imull %r13d, %edx`
		`addl %edx, (%rbx,%rax,4)`
		`cmpl %ecx, %r12d`
		`jbe .L8`
		`imull 0(%rbp,%rcx,4), %r13d`
		`addl %r13d, (%rbx,%rcx,4)`
		`.L8:`
		`addq $16, %rsp`
		`.cfi_remember_state`
		`.cfi_def_cfa_offset 48`
		`movq %r14, %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`