2º curso / 2º cuatr.

Grado Ing. Inform.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 4. Optimización de código

Estudiante (nombre y apellidos): Noelia Escalera Mejías
Grupo de prácticas y profesor de prácticas: A2 (Christian Morillas)
Fecha de entrega: 29/05/2019
Fecha evaluación en clase: 29/05/2019

Antes de comenzar a realizar el trabajo de este cuaderno consultar el fichero con los normas de prácticas que se encuentra en SWAD

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): *Intel® Core™ i7-4700MQ CPU @ 2.40GHz*

Sistema operativo utilizado: *KDE neon User Edition 5.15 x86_64*

Versión de gcc utilizada: ***7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04)***

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

```
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~] 2019-05-15 miércoles
$lscpu
Arquitectura:                        x86_64
modo(s) de operación de las CPUs:    32-bit, 64-bit
Orden de los bytes:                  Little Endian
CPU(s):                              8
Lista de la(s) CPU(s) en línea:      0-7
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»:              4
«Socket(s)»                          1
Modo(s) NUMA:                        1
ID de fabricante:                    GenuineIntel
Familia de CPU:                      6
Modelo:                              60
Nombre del modelo:                   Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz
Revisión:                            3
CPU MHz:                             2394.742
CPU MHz máx.:                        3400,0000
CPU MHz mín.:                        800,0000
BogoMIPS:                            4788.99
Virtualización:                      VT-x
Caché L1d:                           32K
Caché L1i:                           32K
Caché L2:                            256K
Caché L3:                            6144K
CPU(s) del nodo NUMA 0:              0-7
Indicadores:                         fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
 fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc c
puid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 movbe popcnt tsc
_deadline_timer aes xsave avx f16c rdrand lahf_lm abm cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi fle
xpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat pln pts md_clear flush_l1d
```

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

   1.1 Modifique el código C para reducir el tiempo de ejecución (evalúe el tiempo y modifique sólo el trozo que hace la multiplicación y el trozo que se muestra en la Figura 1). Justifique los tiempos obtenidos a partir de la modificación realizada. Incorpore los códigos modificados en el cuaderno.

   1.2 Genere los códigos en ensamblador 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

Figura 1 . Código C++ que suma dos vectores

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

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

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

A) MULTIPLICACIÓN DE MATRICES:
CAPTURA CÓDIGO FUENTE: `pmm-secuencial.c`

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

void multiplicacion (double **m1,double **m2,double **resultado,int N){
    for (int i=0;i<N;i++){
        for (int j=0;j<N;j++){
            for (int k=0;k<N;k++){
                resultado[i][j]=resultado[i][j]+m1[i][k]*m2[k][j];
            }
        }
    }
}

int main(int argc, char ** argv){
    if (argc != 2){
        printf("Faltan las dimensiones de la matriz y vector\n");
        exit(1);
    }

    int i, j, k, contador, N = atoi(argv[1]);
    double **m1, **m2, **resultado, t_inicio, t_fin;

    m1 = (double **)malloc(N*sizeof(double *));

    for(i=0; i<N; i++){
        m1[i] = (double*)malloc(N*sizeof(double));
    }

    resultado = (double **)malloc(N*sizeof(double));

    for(i=0; i<N; i++){
        resultado[i] = (double*)malloc(N*sizeof(double));
    }

    m2 = (double **)malloc(N*sizeof(double *));

    for(i=0; i<N; i++){
        m2[i] = (double*)malloc(N*sizeof(double));
    }

    contador = 1;

    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            m2[i][j] = contador;
            contador++;
        }
    }

    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            resultado[i][j] = 0;
        }
    }

    t_inicio = omp_get_wtime();

    multiplicacion(m1,m2,resultado,N);

    t_fin = omp_get_wtime();

    double tiempo = t_fin - t_inicio;

    printf("resultado[0][0]=%f",resultado[0][0]);
    printf("\nresultado[N-1][N-1]=%f\n",resultado[N-1][N-1]);

    printf("Tiempo: %f\n", tiempo);

    for(i=0; i<N; i++){
        free(m1[i]);
    }
    free(m1);

    for(i=0; i<N; i++){
        free(m2[i]);
    }
    free(m2);

    for(i=0; i<N; i++){
        free(resultado[i]);
    }

    free(resultado);
    return (0);
}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: Desenrrollado del bucle más interior de la multiplicación para romper secuencias de instrucciones dependientes.

Modificación b) –explicación-: Trasponemos la segunda matriz para mejorar los accesos a memoria

...

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) Captura de `pmm-secuencial-modificado_a.c`

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

void multiplicacion (double **m1,double **m2,double **resultado,int N){
    for (int i=0;i<N;i++){
        for (int j=0;j<N;j++){
            double res0=0,res1=0,res2=0,res3=0;
            for (int k=0;k<N;k+=4){
                res0+=resultado[i][j]+m1[i][k]*m2[k][j];
                res1+=resultado[i][j]+m1[i][k+1]*m2[k+1][j];
                res2+=resultado[i][j]+m1[i][k+2]*m2[k+2][j];
                res3+=resultado[i][j]+m1[i][k+3]*m2[k+3][j];
            }
            resultado[i][j]=res0+res1+res2+res3;
        }
    }
}

int main(int argc, char ** argv){
    if (argc != 2){
        printf("Faltan las dimensiones de la matriz y vector\n");
        exit(1);
    }

    int i, j, k, contador, N = atoi(argv[1]);
    double **m1, **m2, **resultado, t_inicio, t_fin;

    m1 = (double **)malloc(N*sizeof(double *));

    for(i=0; i<N; i++){
        m1[i] = (double*)malloc(N*sizeof(double));
    }

    resultado = (double **)malloc(N*sizeof(double));

    for(i=0; i<N; i++){
        resultado[i] = (double*)malloc(N*sizeof(double));
    }

    m2 = (double **)malloc(N*sizeof(double *));
    for(i=0; i<N; i++){
        m2[i] = (double*)malloc(N*sizeof(double));
    }

    contador = 1;


    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            m1[i][j] = contador;
            contador++;
        }
    }

    contador = 1;

    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            m2[i][j] = contador;
            contador++;
        }
    }

    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            resultado[i][j] = 0;
        }
    }

    t_inicio = omp_get_wtime();

    multiplicacion(m1,m2,resultado,N);

    t_fin = omp_get_wtime();

    double tiempo = t_fin - t_inicio;

    printf("resultado[0][0]=%f",resultado[0][0]);
    printf("\nresultado[N-1][N-1]=%f\n",resultado[N-1][N-1]);

    printf("Tiempo: %f\n", tiempo);

    for(i=0; i<N; i++){
        free(m1[i]);
    }

    free(m1);

    for(i=0; i<N; i++){
        free(m2[i]);
    }

    free(m2);

    for(i=0; i<N; i++){
        free(resultado[i]);
    }

    free(resultado);


    return (0);
}
```

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

```
$gcc -fopenmp pmm-secuencial-modificado_a.c -o pmm-secuencial-modificado_a
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$./pmm-secuencial-modificado_a 4
resultado[0][0]=90.000000
resultado[N-1][N-1]=600.000000
Tiempo: 0.000003
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$./pmm-secuencial-modificado_a 20
resultado[0][0]=53410.000000
resultado[N-1][N-1]=1653400.000000
Tiempo: 0.000076
```

b) Captura de `pmm-secuencial-modificado_b.c`

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

void multiplicacion (double **m1,double **m2,double **resultado,int N){
    int slocal;

    for (int i=0;i<N;i++){
        for (int j=0;j<N;j++){
            slocal = 0.0;
            for (int k=0;k<N;k++){
                slocal+=m1[i][k]*m2[j][k];
            }
            resultado[i][j] = slocal;
        }
    }
}

int main(int argc, char ** argv){
    if (argc != 2){
        printf("Faltan las dimensiones de la matriz y vector\n");
        exit(1);
    }

    int i, j, k, fil, col, contador, N = atoi(argv[1]);
    double **m1, **m2, **resultado, t_inicio, t_fin, swap;

    m1 = (double **)malloc(N*sizeof(double *));

    for(i=0; i<N; i++){
        m1[i] = (double*)malloc(N*sizeof(double));
    }

    resultado = (double **)malloc(N*sizeof(double));

    for(i=0; i<N; i++){
        resultado[i] = (double*)malloc(N*sizeof(double));
    }

    m2 = (double **)malloc(N*sizeof(double *));

    for(i=0; i<N; i++){
        m2[i] = (double*)malloc(N*sizeof(double));
    }

    contador = 1;


    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            m1[i][j] = contador;
            contador++;
        }
    }

    contador = 1;

    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            m2[i][j] = contador;
            contador++;
        }
    }

    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            resultado[i][j] = 0;
        }
    }

    for (i=0;i<N;i++){
        for(j=i+1;j<N;j++){
            swap = m2[i][j];
            m2[i][j] = m2[j][i];
            m2[j][i] = swap;
        }
    }

    t_inicio = omp_get_wtime();

    multiplicacion(m1,m2,resultado,N);

    t_fin = omp_get_wtime();

    double tiempo = t_fin - t_inicio;

    printf("resultado[0][0]=%f",resultado[0][0]);
    printf("\nresultado[N-1][N-1]=%f\n",resultado[N-1][N-1]);

    printf("Tiempo: %f\n", tiempo);

    for(i=0; i<N; i++){
        free(m1[i]);
    }

    free(m1);

    for(i=0; i<N; i++){
        free(m2[i]);
    }

    free(m2);

    for(i=0; i<N; i++){
        free(resultado[i]);
    }

    free(resultado);


    return (0);
}
```

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

```
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$gcc -fopenmp pmm-secuencial-modificado_b.c -o pmm-secuencial-modificado_b
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$./pmm-secuencial-modificado_b 4
resultado[0][0]=90.000000
resultado[N-1][N-1]=600.000000
Tiempo: 0.000002
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$./pmm-secuencial-modificado_b 20
resultado[0][0]=53410.000000
resultado[N-1][N-1]=1653400.000000
Tiempo: 0.000053
```

1.1. TIEMPOS:

Modificación	**Breve descripción de las modificaciones**	**Tiempo (Tamaño 1000)**
Sin modificar	*Nada modificado*	*13.196566*
Modificación a)	Desenrrollado del bucle más interior de la multiplicación para romper secuencias de instrucciones dependientes	9.267315
Modificación b)	Trasponemos la segunda matriz para mejorar los accesos a memoria	2.863973
...		

1.1. COMENTARIOS SOBRE LOS RESULTADOS Y JUSTIFICACIÓN DE LAS MEJORAS EN TIEMPO:

Como vemos, hemos conseguido mejorar mucho los tiempos, sobre todo se mejora con la segunda modificación. Esto es debido a que tenemos que hacer muchos accesos a memoria y trasponiendo una matriz los hemos facilidtado mucho.

CÓDIGOS ENSAMBLADOR (parte de la multiplicación)

Original:

```
multiplicacion:
.LFB5:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    %rdi, -24(%rbp)
    movq    %rsi, -32(%rbp)
    movq    %rdx, -40(%rbp)
    movl    %ecx, -44(%rbp)
    movl    $0, -12(%rbp)
    jmp .L2
.L7:
    movl    $0, -8(%rbp)
    jmp .L3
.L6:
    movl    $0, -4(%rbp)
    jmp .L4
.L5:
    movl    -12(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -40(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -8(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm1
    movl    -12(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -24(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -4(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm2
    movl    -4(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -32(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -8(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm0
    mulsd   %xmm2, %xmm0
    movl    -12(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -40(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -8(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    addsd   %xmm1, %xmm0
    movsd   %xmm0, (%rax)
    addl    $1, -4(%rbp)
.L4:
    movl    -4(%rbp), %eax
    cmpl    -44(%rbp), %eax
    jl  .L5
    addl    $1, -8(%rbp)
.L3:
    movl    -8(%rbp), %eax
    cmpl    -44(%rbp), %eax
    jl  .L6
    addl    $1, -12(%rbp)
.L2:
    movl    -12(%rbp), %eax
    cmpl    -44(%rbp), %eax
    jl  .L7
    nop
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE5:
    .size   multiplicacion, .-multiplicacion
    .section    .rodata
    .align 8
```

Modificación 1:

```
multiplicacion:
.LFB5:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    %rdi, -56(%rbp)
    movq    %rsi, -64(%rbp)
    movq    %rdx, -72(%rbp)
    movl    %ecx, -76(%rbp)
    movl    $0, -44(%rbp)
    jmp .L2
.L7:
    movl    $0, -40(%rbp)
    jmp .L3
.L6:
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -32(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -24(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -16(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -8(%rbp)
    movl    $0, -36(%rbp)
    jmp .L4
.L5:
    movl    -44(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -72(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -40(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm1
    movl    -44(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -56(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -36(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm2
    movl    -36(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -64(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -40(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm0
    mulsd   %xmm2, %xmm0
    addsd   %xmm1, %xmm0
    movsd   -32(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -32(%rbp)
    movl    -44(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -72(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -40(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm1
    movl    -44(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -56(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -40(%rbp), %edx
    movslq  %edx, %rdx
    addq    $1, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm2
    movl    -36(%rbp), %eax
    cltq
    addq    $1, %rax
    leaq    0(,%rax,8), %rdx
    movq    -64(%rbp), %rax
    addq    %rdx, %rax
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -16(%rbp)
    movl    -44(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -72(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -40(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm1
    movl    -44(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -56(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -36(%rbp), %edx
    movslq  %edx, %rdx
    addq    $3, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm2
    movl    -36(%rbp), %eax
    cltq
    addq    $3, %rax
    leaq    0(,%rax,8), %rdx
    movq    -64(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -40(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    movsd   (%rax), %xmm0
    mulsd   %xmm2, %xmm0
    addsd   %xmm1, %xmm0
    movsd   -8(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -8(%rbp)
    addl    $4, -36(%rbp)
.L4:
    movl    -36(%rbp), %eax
    cmpl    -76(%rbp), %eax
    jl  .L5
```

```
    addsd   -24(%rbp), %xmm0
    addsd   -16(%rbp), %xmm0
    movl    -44(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -72(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -40(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    addsd   -8(%rbp), %xmm0
    movsd   %xmm0, (%rax)
    addl    $1, -40(%rbp)
.L3:
    movl    -40(%rbp), %eax
    cmpl    -76(%rbp), %eax
    jl  .L6
    addl    $1, -44(%rbp)
.L2:
    movl    -44(%rbp), %eax
    cmpl    -76(%rbp), %eax
    jl  .L7
    nop
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE5:
    .size   multiplicacion, .-multiplicacion
    .section    .rodata
    .align 8
```

Modificación 2:

```
multiplicacion:
.LFB5:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -24(%rbp)
	movq	%rsi, -32(%rbp)
	movq	%rdx, -40(%rbp)
	movl	%ecx, -44(%rbp)
	movl	$0, -12(%rbp)
	jmp	.L2
.L7:
	movl	$0, -8(%rbp)
	jmp	.L3
.L6:
	movl	$0, -16(%rbp)
	movl	$0, -4(%rbp)
	jmp	.L4
.L5:
	cvtsi2sd	-16(%rbp), %xmm1
	movl	-12(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-24(%rbp), %rax
	addq	%rdx, %rax
	movq	(%rax), %rax
	movl	-4(%rbp), %edx
	movslq	%edx, %rdx
	salq	$3, %rdx
	addq	%rdx, %rax
	movsd	(%rax), %xmm2
	movl	-8(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-32(%rbp), %rax
	addq	%rdx, %rax
	movq	(%rax), %rax
	movl	-4(%rbp), %edx
	movslq	%edx, %rdx
	salq	$3, %rdx
	addq	%rdx, %rax
	movsd	(%rax), %xmm0
	mulsd	%xmm2, %xmm0
	addsd	%xmm1, %xmm0
	cvttsd2si	%xmm0, %eax
	movl	%eax, -16(%rbp)
	addl	$1, -4(%rbp)
.L4:
	movl	-4(%rbp), %eax
	cmpl	-44(%rbp), %eax
	jl	.L5
	movl	-12(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-40(%rbp), %rax
	addq	%rdx, %rax
	movq	(%rax), %rax
	movl	-8(%rbp), %edx
	movslq	%edx, %rdx
	salq	$3, %rdx
	addq	%rdx, %rax
	cvtsi2sd	-16(%rbp), %xmm0
	movsd	%xmm0, (%rax)
	addl	$1, -8(%rbp)
.L3:
	movl	-8(%rbp), %eax
	cmpl	-44(%rbp), %eax
	jl	.L6
	addl	$1, -12(%rbp)
.L2:
	movl	-12(%rbp), %eax
	cmpl	-44(%rbp), %eax
	jl	.L7
	nop
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE5:
	.size	multiplicacion, .-multiplicacion
	.section	.rodata
	.align 8
```

La principal diferencia que vemos es que en la primera modificación tenemos muchas más instrucciones, lo cuál es lógico debido a el desenrollamiento de bucle. Con la segunda modificación la mayor diferencia que apreciamos es el cambio de índices.

B) CÓDIGO FIGURA 1:

CAPTURA CÓDIGO FUENTE: `figura1-original.c`

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

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

void multiplicacion (double *R){
    double X1, X2;

    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;
    }
}

int main()
{
    int N=40000;
    double *R, t_inicio, t_fin;

    R = (double*)malloc(N*sizeof(double));

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

    t_inicio = omp_get_wtime();

    multiplicacion(R);

    t_fin = omp_get_wtime();

    double tiempo = t_fin - t_inicio;

    printf("\nR[N-1]=%f\n",R[N-1]);
    printf("Tiempo: %f\n", tiempo);

    return (0);
}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: Desenrrollamos los dos bucles interiores para romper secuencias de instrucciones dependientes.

Modificación b) –explicación-: Unificamos los bucles internos para reducir el número de iteraciones.

...

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) Captura figura1 **`-modificado_a.c`**

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

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

void multiplicacion (double *R){
    double X1, X2;

    for (int ii=0;ii<40000;ii++) {
        X1=0; X2=0;
        double tmp0=0, tmp1=0, tmp2=0, tmp3=0;
        for(int i=0; i<5000;i+=4){
            tmp0+=2*s[i].a+ii;
            tmp1+=2*s[i+1].a+ii;
            tmp2+=2*s[i+2].a+ii;
            tmp3+=2*s[i+3].a+ii;
        }
        X1 = tmp0+tmp1+tmp2+tmp3;
        tmp0=0, tmp1=0, tmp2=0, tmp3=0;
        for(int i=0; i<5000;i+=4){
            tmp0+=3*s[i].b-ii;
            tmp1+=3*s[i+1].b-ii;
            tmp2+=3*s[i+2].b-ii;
            tmp3+=3*s[i+3].b-ii;
        }
        X2=tmp0+tmp1+tmp2+tmp3;

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

int main()
{
    int N=40000;
    double *R, t_inicio, t_fin;

    R = (double*)malloc(N*sizeof(double));

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

    t_inicio = omp_get_wtime();

    multiplicacion(R);

    t_fin = omp_get_wtime();

    double tiempo = t_fin - t_inicio;

    printf("R[0]=%f",R[0]);
    printf("\nR[N-1]=%f\n",R[N-1]);

    printf("Tiempo: %f\n", tiempo);

    return (0);
}
```

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

```
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$gcc -fopenmp figura1-modificado_a.c -o figura1-modificado_a
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$./figura1-modificado_a
R[0]=24995000.000000
R[N-1]=-162502500.000000
Tiempo: 0.756480
```

b) Captura figura1**-`modificado_b.c`**

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

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

void multiplicacion (double *R){
    double X1, X2;

    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;
        }

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

int main()
{
    int N=40000;
    double *R, t_inicio, t_fin;

    R = (double*)malloc(N*sizeof(double));

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

    t_inicio = omp_get_wtime();

    multiplicacion(R);

    t_fin = omp_get_wtime();
    for(i=0; i<N; i++){
        m2[i] = (double*)malloc(N*sizeof(double));
    }

    contador = 1;


    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            m1[i][j] = contador;
            contador++;
        }
    }

    contador = 1;

    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            m2[i][j] = contador;
            contador++;
        }
    }

    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            resultado[i][j] = 0;
        }
    }

    for (i=0;i<N;i++){
        for(j=i+1;j<N;j++){
            swap = m2[i][j];
            m2[i][j] = m2[j][i];
            m2[j][i] = swap;
        }
    }

    t_inicio = omp_get_wtime();

    multiplicacion(m1,m2,resultado,N);

    t_fin = omp_get_wtime();
```

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

```
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$gcc -fopenmp figura1-modificado_b.c -o figura1-modificado_b
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer1] 2019-05-28 martes
$./figura1-modificado_b
R[0]=24995000.000000
R[N-1]=-162502500.000000
Tiempo: 0.754052
```

1.1. TIEMPOS:

Modificación	**Breve descripción de las modificaciones**	**-O2**
Sin modificar	*Sin modificar*	*1.104818*
Modificación a)	Desenrrollado de los dos bucles interiores para romper secuencias de instrucciones dependientes.	0.746580
Modificación b)	Unificar los bucles internos para reducir iteraciones	0.754052
...		

1.1. COMENTARIOS SOBRE LOS RESULTADOS Y JUSTIFICACIÓN DE LAS MEJORAS EN TIEMPO:

Como vemos aquí, la diferencia de tiempos no es muy grande y las dos modificaciones restan más o menos lo mismo.

CÓDIGO ENSAMBLADOR (sólo multiplicación)

Código original:

```
multiplicacion:
.LFB5:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -40(%rbp)
	movl	$0, -28(%rbp)
	jmp	.L2
.L10:
	pxor	%xmm0, %xmm0
	movsd	%xmm0, -16(%rbp)
	pxor	%xmm0, %xmm0
	movsd	%xmm0, -8(%rbp)
	movl	$0, -24(%rbp)
	jmp	.L3
.L4:
	movl	-24(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	leaq	s(%rip), %rax
	movl	(%rdx,%rax), %eax
	leal	(%rax,%rax), %edx
	movl	-28(%rbp), %eax
	addl	%edx, %eax
	cvtsi2sd	%eax, %xmm0
	movsd	-16(%rbp), %xmm1
	addsd	%xmm1, %xmm0
	movsd	%xmm0, -16(%rbp)
	addl	$1, -24(%rbp)
.L3:
	cmpl	$4999, -24(%rbp)
	jle	.L4
	movl	$0, -20(%rbp)
	jmp	.L5
.L6:
	movl	-20(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	leaq	4+s(%rip), %rax
	movl	(%rdx,%rax), %edx
	movl	%edx, %eax
	addl	%eax, %eax
	addl	%edx, %eax
	subl	-28(%rbp), %eax
	cvtsi2sd	%eax, %xmm0
	movsd	-8(%rbp), %xmm1
	addsd	%xmm1, %xmm0
	movsd	%xmm0, -8(%rbp)
	addl	$1, -20(%rbp)
.L5:
	cmpl	$4999, -20(%rbp)
	jle	.L6
	movsd	-8(%rbp), %xmm0
	ucomisd	-16(%rbp), %xmm0
	jbe	.L12
	movl	-28(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-40(%rbp), %rax
	addq	%rdx, %rax
	movsd	-16(%rbp), %xmm0
	movsd	%xmm0, (%rax)
	jmp	.L9
.L12:
	movl	-28(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-40(%rbp), %rax
	addq	%rdx, %rax
	movsd	-8(%rbp), %xmm0
	movsd	%xmm0, (%rax)
.L9:
	addl	$1, -28(%rbp)
.L2:
	cmpl	$39999, -28(%rbp)
	jle	.L10
	nop
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE5:
	.size	multiplicacion, .-multiplicacion
	.section	.rodata
```

Modificación 1:

```
multiplicacion:
.LFB5:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    %rdi, -72(%rbp)
    movl    $0, -60(%rbp)
    jmp .L2
.L10:
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -16(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -8(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -48(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -40(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -32(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -24(%rbp)
    movl    $0, -56(%rbp)
    jmp .L3
.L4:
    movl    -56(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    leaq    s(%rip), %rax
    movl    (%rdx,%rax), %ea
    leal    (%rax,%rax), %ed
    movl    -60(%rbp), %eax
    addl    %edx, %eax
    cvtsi2sd    %eax, %xmm0
    movsd   -48(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -48(%rbp)
    movl    -56(%rbp), %eax
    addl    $1, %eax
    cltq
    leaq    0(,%rax,8), %rdx
    leaq    s(%rip), %rax
    movl    (%rdx,%rax), %ea
    leal    (%rax,%rax), %ed
    movl    -60(%rbp), %eax
    addl    %edx, %eax
    cvtsi2sd    %eax, %xmm0
    movsd   -40(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -40(%rbp)
    movl    -56(%rbp), %eax
    addl    $2, %eax
    cltq
    leaq    0(,%rax,8), %rdx
    leaq    s(%rip), %rax
    movl    (%rdx,%rax), %eax
    leal    (%rax,%rax), %edx
    movl    -60(%rbp), %eax
    addl    %edx, %eax
    cvtsi2sd    %eax, %xmm0
    movsd   -32(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -32(%rbp)
    movl    -56(%rbp), %eax
    addl    $3, %eax
    cltq
    leaq    0(,%rax,8), %rdx
    leaq    s(%rip), %rax
    movl    (%rdx,%rax), %eax
    leal    (%rax,%rax), %edx
    movl    -60(%rbp), %eax
    addl    %edx, %eax
    cvtsi2sd    %eax, %xmm0
    movsd   -24(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -24(%rbp)
    addl    $4, -56(%rbp)
.L3:
    cmpl    $4999, -56(%rbp)
    jle .L4
    movsd   -48(%rbp), %xmm0
    addsd   -40(%rbp), %xmm0
    addsd   -32(%rbp), %xmm0
    movsd   -24(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -16(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -48(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -40(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -32(%rbp)
    pxor    %xmm0, %xmm0
    movsd   %xmm0, -24(%rbp)
    movl    $0, -52(%rbp)
    jmp .L5
.L6:
    movl    -52(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    leaq    4+s(%rip), %rax
    movl    (%rdx,%rax), %edx
    movl    %edx, %eax
    addl    %eax, %eax
    addl    %edx, %eax
    subl    -60(%rbp), %eax
    cvtsi2sd    %eax, %xmm0
    movsd   -48(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -48(%rbp)
    movl    -52(%rbp), %eax
    addl    $1, %eax
    cltq
    leaq    0(,%rax,8), %rdx
    leaq    4+s(%rip), %rax
    movl    (%rdx,%rax), %edx
    movl    %edx, %eax
    addl    %eax, %eax
    addl    %edx, %eax
    subl    -60(%rbp), %eax
    cvtsi2sd    %eax, %xmm0
    movsd   -40(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -40(%rbp)
    movl    -52(%rbp), %eax
    addl    $2, %eax
    cltq
    leaq    0(,%rax,8), %rdx
    leaq    4+s(%rip), %rax
    movl    (%rdx,%rax), %edx
    movl    %edx, %eax
    addl    %eax, %eax
    addl    %edx, %eax
    subl    -60(%rbp), %eax
    cvtsi2sd    %eax, %xmm0
    movsd   -32(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -32(%rbp)
    movl    -52(%rbp), %eax
    addl    $3, %eax
    cltq
    leaq    0(,%rax,8), %rdx
```

```
    addsd   -24(%rbp), %xmm0
    addsd   -16(%rbp), %xmm0
    movl    -44(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -72(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    movl    -40(%rbp), %edx
    movslq  %edx, %rdx
    salq    $3, %rdx
    addq    %rdx, %rax
    addsd   -8(%rbp), %xmm0
    movsd   %xmm0, (%rax)
    addl    $1, -40(%rbp)
.L3:
    movl    -40(%rbp), %eax
    cmpl    -76(%rbp), %eax
    jl  .L6
    addl    $1, -44(%rbp)
.L2:
    movl    -44(%rbp), %eax
    cmpl    -76(%rbp), %eax
    jl  .L7
    nop
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE5:
    .size   multiplicacion, .-multiplicacion
    .section    .rodata
    .align 8
```

```
    .cfi_endproc
.LFE5:
    .size   multiplicacion, .-multiplicacion
    .section    .rodata
```

Modificación 2:

```
multiplicacion:
.LFB5:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -40(%rbp)
	movl	$0, -24(%rbp)
	jmp	.L2
.L8:
	pxor	%xmm0, %xmm0
	movsd	%xmm0, -16(%rbp)
	pxor	%xmm0, %xmm0
	movsd	%xmm0, -8(%rbp)
	movl	$0, -20(%rbp)
	jmp	.L3
.L4:
	movl	-20(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	leaq	s(%rip), %rax
	movl	(%rdx,%rax), %eax
	leal	(%rax,%rax), %edx
	movl	-24(%rbp), %eax
	addl	%edx, %eax
	cvtsi2sd	%eax, %xmm0
	movsd	-16(%rbp), %xmm1
	addsd	%xmm1, %xmm0
	movsd	%xmm0, -16(%rbp)
	movl	-20(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	leaq	4+s(%rip), %rax
	movl	(%rdx,%rax), %edx
	movl	%edx, %eax
	addl	%eax, %eax
	addl	%edx, %eax
	subl	-24(%rbp), %eax
	cvtsi2sd	%eax, %xmm0
	movsd	-8(%rbp), %xmm1
	addsd	%xmm1, %xmm0
	movsd	%xmm0, -8(%rbp)
	addl	$1, -20(%rbp)
.L3:
	cmpl	$4999, -20(%rbp)
	jle	.L4
	movsd	(%rax), %xmm0
	mulsd	%xmm2, %xmm0
	addsd	%xmm1, %xmm0
	cvttsd2si	%xmm0, %eax
	movl	%eax, -16(%rbp)
	addl	$1, -4(%rbp)
.L4:
	movl	-4(%rbp), %eax
	cmpl	-44(%rbp), %eax
	jl	.L5
	movl	-12(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-40(%rbp), %rax
	addq	%rdx, %rax
	movq	(%rax), %rax
	movl	-8(%rbp), %edx
	movslq	%edx, %rdx
	salq	$3, %rdx
	addq	%rdx, %rax
	cvtsi2sd	-16(%rbp), %xmm0
	movsd	%xmm0, (%rax)
	addl	$1, -8(%rbp)
.L3:
	movl	-8(%rbp), %eax
	cmpl	-44(%rbp), %eax
	jl	.L6
	addl	$1, -12(%rbp)
.L2:
	movl	-12(%rbp), %eax
	cmpl	-44(%rbp), %eax
	jl	.L7
	nop
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE5:
	.size	multiplicacion, .-multiplicacion
	.section	.rodata
	.align 8
```

Las principales diferencias que encontramos son que en la primera modificación volvemos a tener más instrucciones y en la segunda tenemos menos saltos.

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 que opera con flotantes de doble precisión denominada DAXPY (***D**ouble precision- real **A**lpha **X** **P**lus **Y***) que multiplica un vector por una constante y los suma a otro vector (Lección 3/Tema 1):

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

2.1. Genere los programas en ensamblador para cada una de las siguientes opciones de optimización del compilador: -O0, -Os, -O2, -O3. Explique las diferencias que se observan en el código justificando al mismo tiempo las mejoras en velocidad que acarrean. Incorpore los códigos al cuaderno de prácticas y destaque las diferencias entre ellos. Sólo se debe evaluar el tiempo del núcleo DAXPY

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 y compárela con el valor obtenido para Rmax. -Consulte la Lección 3 del Tema 1.

CAPTURA CÓDIGO FUENTE: daxpy.c

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

void daxpy(double a, double *y, double *x){
    for (int i=1;i<=100000000;i++) y[i]= a*x[i] + y[i];
}

int main(){
    double a = 100, *y, *x, t_inicio, t_fin;

    y = (double*)malloc(100000000*sizeof(double));
    x = (double*)malloc(100000000*sizeof(double));

    for (int i=0; i<100000000; i++){
        y[i] = i;
        x[i] = i;
    }

    t_inicio = omp_get_wtime();

    daxpy(a,y,x);

    t_fin = omp_get_wtime();

    double tiempo = t_fin - t_inicio;

    printf("y[0]=%f",y[0]);
    printf("\ny[N-1]=%f\n",y[100000000-1]);

    printf("Tiempo: %f\n", tiempo);

    return (0);
}
```

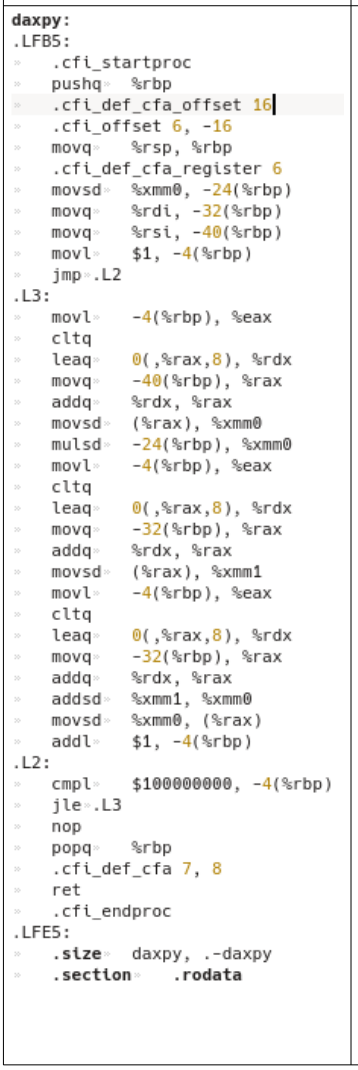
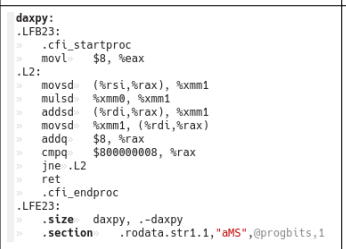
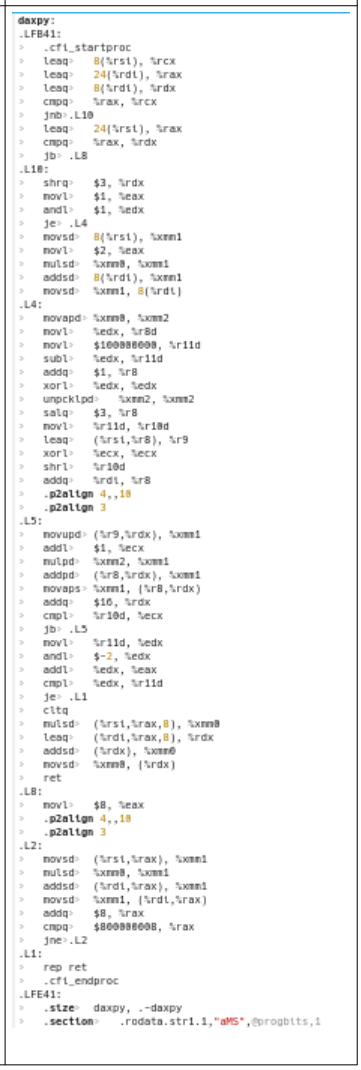
Tiempos ejec.	-O0	-Os	-O2	-O3
	0.351345	0.232318	0.241435	0.234977

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

```
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer2] 2019-05-28 martes
$gcc -O0 -fopenmp daxpy.c -o daxpy
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer2] 2019-05-28 martes
$./daxpy
y[0]=0.000000
y[N-1]=10099999899.000000
Tiempo: 0.351345
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer2] 2019-05-28 martes
$gcc -Os -fopenmp daxpy.c -o daxpy
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer2] 2019-05-28 martes
$./daxpy
y[0]=0.000000
y[N-1]=10099999899.000000
Tiempo: 0.232318
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer2] 2019-05-28 martes
$gcc -O2 -fopenmp daxpy.c -o daxpy
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer2] 2019-05-28 martes
$./daxpy
y[0]=0.000000
y[N-1]=10099999899.000000
Tiempo: 0.241435
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer2] 2019-05-28 martes
$gcc -O3 -fopenmp daxpy.c -o daxpy
[NoeliaEscaleraMejias noelia@noelia-HP-ENVY-17-Notebook-PC:~/Escritorio/Universidad/Segundo/AC/Prácticas/bp4/ejer2] 2019-05-28 martes
$./daxpy
y[0]=0.000000
y[N-1]=10099999899.000000
Tiempo: 0.234977
```

COMENTARIOS QUE EXPLIQUEN LAS DIFERENCIAS EN ENSAMBLADOR:

CÓDIGO EN ENSAMBLADOR (no es necesario introducir aquí el código como captura de pantalla, ajustar el tamaño de la letra para que una instrucción no ocupe más de un renglón)**:**
(PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR DONDE ESTÁ EL CÓDIGO EVALUADO, USE COLORES PARA DESTACAR LAS DIFERENCIAS)

daxpyO0.s	daxpyOs.s	daxpyO2.s	daxpyO3.s
			

Las diferencias entre Os y O2 son mínimas, se cambia básicamente el nombre de la última etiqueta y O2 usa dos instrucciones p2align. De O0 a Os hay muchísimos cambios, se eliminan muchos desplazamientos e incluso una etiqueta, queda un código mucho más compacto. En O3 podemos apreciar que se aplica desenrollar el bucle.