

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): Marta Gómez Macías

Grupo de prácticas: C1

Fecha de entrega: 02/06/2015 - 23:59

Fecha evaluación en clase: 03/06/2015 - 10:00 (Aula 0.2)

Versión de gcc utilizada: gcc versión 5.1.0

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, y para un programa que implemente la multiplicación de matrices:
 - a. Modifique el código C para reducir el tiempo de ejecución del mismo. Justifique los tiempos obtenidos a partir de la modificación realizada.
 - b. Genere los programas en ensamblador para los programas modificados obtenidos en el punto anterior considerando las distintas opciones de optimización del compilador (-O1, -O2,...) e incorpórelos al cuaderno de prácticas. Compare los tiempos de ejecución de las versiones de código ejecutable obtenidas con las distintas opciones de optimización y explique las diferencias en tiempo a partir de las características de dichos códigos. Destaque las diferencias en el código ensamblador.
 - c. (Ejercicio EXTRA) Intente mejorar los resultados obtenidos transformando el código ensamblador del programa para el que se han conseguido las mejores prestaciones de tiempo

A) MULTIPLICACIÓN DE MATRICES:

CÓDIGO FUENTE: pmm-secuencial-modificado.c

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

```
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];
```

MODIFICACIONES REALIZADAS:

Modificación a) -explicación-: Cambiar los bucles *j* y *k* de sitio para optimizar los accesos a memoria a la hora de multiplicar las matrices.

Modificación b) -explicación-: Transformar la matriz en un vector de n^2 posiciones para hacer alineamiento de memoria.

Modificación	-O0	-O1	-O2	-O3	-Os
Sin modificar	9.097073036	6.821439556	7.766282208	7.215468024	6.797240713
Modificación a)	3.910894016	1.245312651	0.931261759	0.451254464	1.280312545
Modificación b)	4.025883060	0.875032645	0.836272713	0.881186086	0.902114086

COMENTARIOS SOBRE LOS RESULTADOS: En general ambas optimizaciones mejoran bastante los resultados, pero con O3, el alineamiento de datos nos da peores resultados. Tanto en O1 como en O2 como en Os se nota la mejora.

En O0 se trabaja mucho sobre direcciones relativas a la pila, por tanto, debemos estar copiando todo el rato dichas direcciones a los registros de la arquitectura. Además, las etiquetas que corresponderían a cada bucle están desordenadas, por tanto, hay que hacer muchos saltos en cada iteración:

```

        movl $0, -4(%rbp)      # i = 0
        jmp  .L7
.L12:
        movl $0, -12(%rbp)     # k = 0
        jmp  .L8               # goto L8
.L11:
        movl $0, -8(%rbp)      # j = 0
        jmp  .L9               # goto L9
.L10:
        movl -4(%rbp), %eax     # eax = i
        imull -44(%rbp), %eax   # eax = n*i
        movl %eax, %edx        # edx = eax
        movl -8(%rbp), %eax     # eax = j
        addl %edx, %eax        # eax = j + (n*i)
        movl %eax, %eax        # eax = eax
        leaq 0(,%rax,4), %rdx   # rdx = posicion del vector
        movq -24(%rbp), %rax    # rax = a
        addq %rax, %rdx        # rdx = posicion de memoria actual
        movl -4(%rbp), %eax     # eax = i
        imull -44(%rbp), %eax   # eax = i*n
        movl %eax, %ecx        # ecx = eax
        movl -8(%rbp), %eax     # eax = j
        addl %ecx, %eax        # eax = j + i*n
        movl %eax, %eax        # eax = eax
        leaq 0(,%rax,4), %rcx   # rcx = posicion del vector
        movq -24(%rbp), %rax    # rcx = a
        addq %rcx, %rax        # rax = posicion de memoria actual
        movl (%rax), %ecx      # ecx = rax
        movl -4(%rbp), %eax     # eax = i
        imull -44(%rbp), %eax   # eax = i*n
        movl %eax, %esi        # esi = i*n
        movl -12(%rbp), %eax    # eax = k
        addl %esi, %eax        # eax = k + i*n
        movl %eax, %eax        # eax = eax
        leaq 0(,%rax,4), %rsi   # rsi = posicion del vector
        movq -32(%rbp), %rax    # rax = b
        addq %rsi, %rax        # rax = posicion de memoria actual
        movl (%rax), %esi      # esi = rax
        movl -12(%rbp), %eax    # eax = k
        imull -44(%rbp), %eax   # eax = k*n
        movl %eax, %edi        # edi = k*n

```

```

        movl    -8(%rbp), %eax    # eax = j
        addl    %edi, %eax        # eax = k*n + j
        movl    %eax, %eax        # eax = eax
        leaq    0(,%rax,4), %rdi   # rdi = posicion del vector
        movq    -40(%rbp), %rax    # rax = c
        addq    %rdi, %rax        # rax = posicion de memoria actual
        movl    (%rax), %eax       # eax = rax
        imull    %esi, %eax        # eax = b * c
        addl    %ecx, %eax        # eax = b * c + a
        movl    %eax, (%rdx)      # rdx = eax
        addl    $1, -8(%rbp)      # j++

.L9:
        movl    -8(%rbp), %eax    # eax = j
        cmpl    -44(%rbp), %eax   # compare j:n
        jb      .L10              # if j<n goto L10
        addl    $1, -12(%rbp)

.L8:
        movl    -12(%rbp), %eax    # eax = k
        cmpl    -44(%rbp), %eax   # compare k:n
        jb      .L11              # if k<n goto L11
        addl    $1, -4(%rbp)

.L7:
        movl    -4(%rbp), %eax     # eax = i
        cmpl    -44(%rbp), %eax   # compare i:n
        jb      .L12              # if i<n goto L2

```

Mientras que en O1, tenemos los bucles ordenados y así, nos ahorramos muchos saltos. El bucle más interno sería L12.

```

        testl    %r12d, %r12d      # check n>0
        jne      .L17              # if n!=0 goto L17
        jmp      .L11              # if n==0 goto L11

.L12:
        movl    %eax, %ecx         # ecx = eax
        leal    (%rsi,%rax), %edx   # edx = eax + rsi
        movl    0(%rbp,%rdx,4), %edx # edx = 4*rdx + rbp
        imull    (%r8), %edx        # edx *= r8
        addl    %edx, (%rbx,%rcx,4) # 4*rcx+rbx = edx
        addl    $1, %eax           # j++
        cmpl    %eax, %edi         # compare j:n
        jne      .L12              # if j!=n goto L12
        addl    $1, %r9d           # k++
        addl    %r12d, %esi        # esi += n
        cmpl    %edi, %r9d        # compare k:n
        je       .L13              # if k==n goto L13

.L14:
        movl    %r9d, %eax         # eax = j
        leaq    0(%r13,%rax,4), %r8 # r8 = 4*j + r13
        movl    %r10d, %eax        # eax = rd10
        jmp      .L12

.L13:
        addl    $1, %r11d          # i++
        addl    %r12d, %edi        # edi += n
        subl    %r12d, %r14d       # 
        cmpl    %r11d, %r12d       # compare i:n
        je      .L11              # if i==n goto L11

.L17:
        movl    %edi, %r10d        # i = 0
        subl    %r12d, %r10d       # rd10 = 0
        movl    %r14d, %esi        # esi = 0
        movl    %r10d, %r9d       # j = 0

```

```

        jmp     .L14                # goto L14

```

Esto también se hace en O2, con la diferencia de que los bucles están ordenados en orden ascendente en vez de descendente, es decir, el bucle más interno es L13, el último mientras que en O1, era L2, es decir, el primero.:

```

.L18:
    movl    %r10d, %r13d
    movl    %r15d, %r8d
    subl    %ecx, %r13d
    movl    %r13d, %r12d
    .p2align 4,,10
    .p2align 3

.L15:
    movl    %r12d, %eax
    leaq    0(%rbp,%rax,4), %r11
    movl    %r13d, %eax
    .p2align 4,,10
    .p2align 3

.L13:
    leal    (%r8,%rax), %esi
    movl    %eax, %edi
    addl    $1, %eax
    movl    (%rdx,%rsi,4), %esi
    imull    (%r11), %esi
    addl    %esi, (%rbx,%rdi,4)
    cmpl    %eax, %r10d
    jne     .L13
    addl    $1, %r12d
    addl    %ecx, %r8d
    cmpl    %r10d, %r12d
    jne     .L15
    addl    $1, %r14d
    addl    %ecx, %r10d
    subl    %ecx, %r15d
    cmpl    %r14d, %ecx
    jne     .L18

```

En O3, obtenemos prácticamente el mismo código que con O2.

Por último, en O3, las diferencias básicas se ven en el uso de instrucciones como `incl` para incrementar el valor de una variable. Pero también hay diferencias estructurales: en este caso, el bucle más interno (j) es L13, el incremento de las variables iteradores las hace en etiquetas diferentes (L18, L12) y la inicialización de las variables iteradoras, antes de L13. Así, conseguimos una estructura intermedia entre la que teníamos en O1 y la que teníamos tanto en O2 como en O3.

```

.L10:
    cmpl    %ebx, %edi
    je      .L17
    xorl    %edx, %edx
    xorl    %esi, %esi

.L14:
    cmpl    %ebx, %esi
    je      .L12
    leal    (%rsi,%rcx), %eax
    leaq    (%r12,%rax,4), %r11
    xorl    %eax, %eax

.L13:
    cmpl    %ebx, %eax
    je      .L18

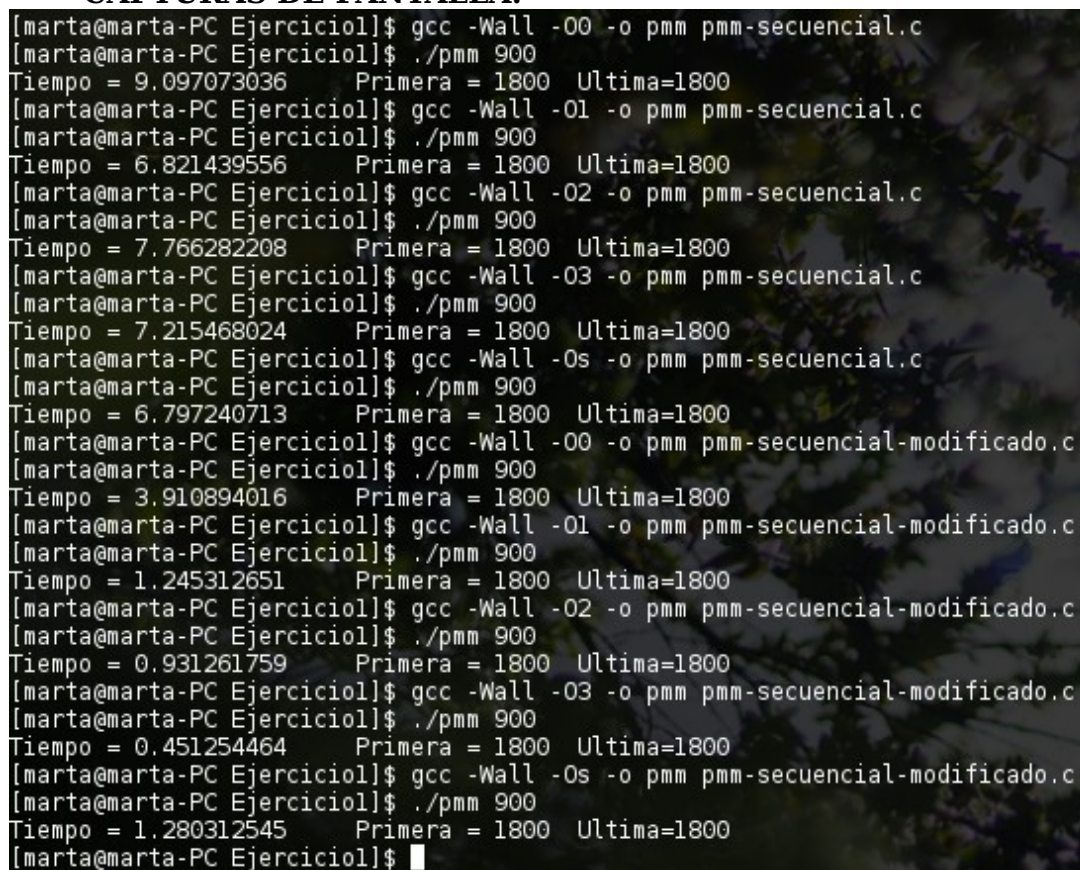
```

```

        leal    (%rax,%rdx), %r8d
        leal    (%rax,%rcx), %r10d
        incl    %eax
        movl    0(%r13,%r8,4), %r8d
        imull   (%r11), %r8d
        addl    %r8d, 0(%rbp,%r10,4)
        jmp     .L13
.L18:
        incl    %esi
        addl    %ebx, %edx
        jmp     .L14
.L12:
        incl    %edi
        addl    %ebx, %ecx
        jmp     .L10

```

CAPTURAS DE PANTALLA:



```

[marta@marta-PC Ejercicio1]$ gcc -Wall -O0 -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 9.097073036      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O1 -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 6.821439556      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O2 -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 7.766282208      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O3 -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 7.215468024      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -Os -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 6.797240713      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O0 -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 3.910894016      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O1 -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 1.245312651      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O2 -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 0.931261759      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O3 -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 0.451254464      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -Os -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 1.280312545      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$

```



```

[marta@marta-PC Ejercicio1]$ gcc -Wall -O0 -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 9.097073036      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O1 -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 6.821439556      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O2 -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 7.766282208      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O3 -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 7.215468024      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -Os -o pmm pmm-secuencial.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 6.797240713      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O0 -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 3.910894016      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O1 -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 1.245312651      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O2 -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 0.931261759      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -O3 -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 0.451254464      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$ gcc -Wall -Os -o pmm pmm-secuencial-modificado.c
[marta@marta-PC Ejercicio1]$ ./pmm 900
Tiempo = 1.280312545      Primera = 1800  Ultima=1800
[marta@marta-PC Ejercicio1]$

```

B) CÓDIGO FIGURA 1:**CÓDIGO FUENTE:** figura1-modificado.c**(ADJUNTAR CÓDIGO FUENTE AL .ZIP)**

```

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;
}

```

MODIFICACIONES REALIZADAS:

Modificación a) -explicación-: He unido los dos bucles for que había inicialmente en uno sólo pues ambos eran iguales.

Modificación b) -explicación-: He cambiado la estructura if else que había al final del bucle por una estructura usando el operador ternario así ahorramos instrucciones de salto. Aunque como se aprecia en la tabla inferior, se obtienen peores resultados con el operador ternario que sin él.

Modificación c) -explicación-: He desenrollado el bucle más interno

para tener menos iteraciones y hacerlo, por tanto, más eficiente.

Modificación	-O0	-O1	-O2	-O3	-Os
Sin modificar	1.166604427	0.355399931	0.353267023	0.163762494	0.313044189
Modificación a)	0.751596482	0.226692690	0.248963953	0.136337018	0.227854896
Modificación b)	0.728572708	0.285495063	0.274505428	0.137173200	0.255367189
Modificación c)	0.623292787	0.209795447	0.216518448	0.181059524	0.173372602

En -O0, usamos direcciones de memoria relativas a %rbp, mientras que en -O1, usamos registros de la arquitectura. Por tanto, nos ahorramos tener que hacer operaciones como `movl -8(%rbp), %eax`. Así, para realizar las dos primeras sumas del bucle más interno, en -O0 hacemos lo siguiente:

```

movl -8(%rbp), %eax      # guardamos i en eax
cltq                    # extendemos su valor a 64 bits
movl s(,%rax,8), %eax    # cogemos 2*s[i].a
leal (%rax,%rax), %edx   # edx = s[i].a
movl -4(%rbp), %eax      # eax = ii
addl %edx, %eax          # sumamos ambos valores
addl %eax, -12(%rbp)     # guardamos el valor en X1
movl -8(%rbp), %eax      # copiamos i en eax
cltq                    # extendemos su valor a 64bits
movl s+4(,%rax,8), %edx  # cogemos s[i].b
movl %edx, %eax          # lo guardamos en eax
addl %eax, %eax          # lo sumamos consigo mismo
addl %edx, %eax          # y lo sumamos con lo inicial
subl -4(%rbp), %eax      # le restamos ii
addl %eax, -16(%rbp)     # lo guardamos en X2

```

Mientras que en -O1, hacemos esto:

```

movl (%rax), %edi        # s[i].a en edi
leal (%rcx,%rdi,2), %edi # edi = 2*s[i].a
addl %edi, %edx          # X1 += edi
movl 4(%rax), %edi       # s[i].b en edi
leal (%rdi,%rdi,2), %r8d # rd8 = 3*edi
subl %ecx, %r8d          # rd8 = rd8 - ii
addl %r8d, %esi          # X2 += rd8

```

Y como decía, en -O1 se usan registros de la arquitectura como %rd8, mientras que en -O0 no. Así en -O1 ahorramos bastantes operaciones.

Para acceder a la siguiente posición de s, en -O0 hacemos esto:

```

addl $1, %eax            # y le sumamos 1
cltq                    # extendemos a 64 bits
movl s(,%rax,8), %eax    # cogemos s[i+1].a

```

Mientras que en -O1 accedemos a s con desplazamientos:

```

movl 8(%rax), %edi       # edi = s[i+1].a

```

Otra diferencia, es que para saber si es el fin del bucle o no, en -O0 comparamos i con 4999, y en -O1, comparamos la posición de s actual con la posición final de s:

```

cmpl $4999, -8(%rbp)     # comparamos i y 4999
jle .L4                  # si es menor o igual vamos a L4

addq $32, %rax           # desplazamos s 32
cmpq %r9, %rax           # si s es menor que el fin de s
jne .L3                  # volvemos al bucle.si es distinto

```

Previamente en el código hemos hecho:

```

movl $s+40000, %r9d      # rd9 = ultima direccion de s

```

si no, no funcionaría.

En -O2, empiezan a verse optimizaciones a nivel de bit con operaciones como xor

con un registro consigo mismo para inicializarse a 0:

```
xorl    %esi, %esi          # X2 = 0
xorl    %edx, %edx          # X1 = 0
```

También, aunque la manera de sumar es la misma que en -O1, las operaciones empiezan a desordenarse. Además, hacemos los desplazamientos en negativo (sumamos 32 a *s* al principio en vez de al final del bucle como hacíamos en -O1):

```
movl    (%rax), %edi        # s[i].a en edi
addq    $32, %rax           # desplazamos rax 32
leal    (%rcx,%rdi,2), %edi  # edi = 2*s[i].a + ii
addl    %edi, %edx          # X1 += edi
movl    -28(%rax), %edi     # s[i].b en edi
leal    (%rdi,%rdi,2), %r8d  # rd8 = 3*s[i].b
movl    -24(%rax), %edi     # s[i+1].a en edi
subl    %ecx, %r8d          # rd8 -= ii
leal    (%rcx,%rdi,2), %edi  # edi = 2*s[i+1].a
addl    %r8d, %esi          # X2 += rd8
```

Como se ve, antes de restar *ii* a $3*s[i].b$, ya estamos guardando el valor $s[i+1].a$ y, antes de acumular dicho valor en *X2*, ya estamos calculando el que acumularemos en *X1*.

Entre -O2 y -O3 en este caso no hay ninguna diferencia, porque no tenemos funciones *inline* en este código, por tanto, aplica las mismas optimizaciones.

En -Os, para reducir el tamaño del código, en vez de usar instrucciones *lea* o *add*, directamente usa la instrucción *imull* para obtener los valores del vector multiplicados, en concreto los valores que implican multiplicar por tres:

```
imull    $3, -20(%rcx), %r9d  # rd9 = 3*s[i].b
```

Por lo demás, es bastante parecido al código obtenido en -O2.

CAPTURAS DE PANTALLA:

```
[marta@marta-PC Ejercicio1]$ gcc -Wall -O0 -o fig1 figural-modificado.c
[marta@marta-PC Ejercicio1]$ ./fig1
R[0] = 0, R[39999] = -199995000

Tiempo (seg.) = 0.623292787
[marta@marta-PC Ejercicio1]$ gcc -Wall -O1 -o fig1 figural-modificado.c
[marta@marta-PC Ejercicio1]$ ./fig1
R[0] = 0, R[39999] = -199995000

Tiempo (seg.) = 0.209795447
[marta@marta-PC Ejercicio1]$ gcc -Wall -O2 -o fig1 figural-modificado.c
[marta@marta-PC Ejercicio1]$ ./fig1
R[0] = 0, R[39999] = -199995000

Tiempo (seg.) = 0.216518448
[marta@marta-PC Ejercicio1]$ gcc -Wall -O3 -o fig1 figural-modificado.c
[marta@marta-PC Ejercicio1]$ ./fig1
R[0] = 0, R[39999] = -199995000

Tiempo (seg.) = 0.181059524
[marta@marta-PC Ejercicio1]$ gcc -Wall -Os -o fig1 figural-modificado.c
[marta@marta-PC Ejercicio1]$ ./fig1
R[0] = 0, R[39999] = -199995000

Tiempo (seg.) = 0.173372602
```

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 *lpmm*:

3. .LFB3ista de los 500 computadores más rápidos del mundo (el Top500 Report). El núcleo de este programa es una rutina denominada DAXPY (*Double precision- real Alpha X Plus 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];
```

- Genere los programas en ensamblador para cada una de las opciones de optimización del compilador (-O1, -O2,...) y explique las diferencias que se observan en el código justificando las mejoras en velocidad que acarorean. Incorpore los códigos al cuaderno de prácticas y destaque las diferencias entre ellos.
- (Ejercicio EXTRA) Para la mejor de las opciones, obtenga los tiempos de ejecución con distintos valores de N y determine para su sistema los valores de Rmax (valor máximo del número de operaciones en coma flotante por unidad de tiempo), Nmax (valor de N para el que se consigue Rmax), y N1/2 (valor de N para el que se obtiene Rmax/2). Estime el valor de la velocidad pico (Rpico) del procesador (consulte en [4] el número de ciclos por instrucción punto flotante para la familia y modelo de procesador que está utilizando) y compárela con el valor obtenido para Rmax. -Consulte la Lección 3 del Tema 1.

CÓDIGO FUENTE: daxpy.c

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

```
void DAXPY (int *y, int *x, int a, unsigned n) {
    unsigned i;
    for (i=0; i<n; i++)
        y[i] += a*x[i];
}
```

Tiempos ejec.	-O0	-O1	-O2	-O3	-Os
	0.396215297	0.143110587	0.111823235	0.085888037	0.144529917

CAPTURAS DE PANTALLA:

```

[marta@marta-PC Ejercicio2]$ gcc -Wall -O0 -o daxpy daxpy.c
[marta@marta-PC Ejercicio2]$ ./daxpy 123456789 3
y[0] = 2, y[123456788] = 864197518

Tiempo (seg.) = 0.396215297
[marta@marta-PC Ejercicio2]$ gcc -Wall -O1 -o daxpy daxpy.c
[marta@marta-PC Ejercicio2]$ ./daxpy 123456789 3
y[0] = 2, y[123456788] = 864197518

Tiempo (seg.) = 0.143110587
[marta@marta-PC Ejercicio2]$ gcc -Wall -O2 -o daxpy daxpy.c
[marta@marta-PC Ejercicio2]$ ./daxpy 123456789 3
y[0] = 2, y[123456788] = 864197518

Tiempo (seg.) = 0.111823235
[marta@marta-PC Ejercicio2]$ gcc -Wall -O3 -o daxpy daxpy.c
[marta@marta-PC Ejercicio2]$ ./daxpy 123456789 3
y[0] = 2, y[123456788] = 864197518

Tiempo (seg.) = 0.085888037
[marta@marta-PC Ejercicio2]$ gcc -Wall -Os -o daxpy daxpy.c
[marta@marta-PC Ejercicio2]$ ./daxpy 123456789 3
y[0] = 2, y[123456788] = 864197518

Tiempo (seg.) = 0.144529917
[marta@marta-PC Ejercicio2]$ █

```

COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:

Nada más examinar el código obtenido con O0 y el código obtenido con O1 se ve una clara diferencia: con O0 usamos direcciones relativas a la pila y con O1, registros de la arquitectura. Así, ahorramos muchas operaciones move innecesarias y, como se puede ver abajo, obtenemos un código mucho más pequeño para O1 que para O0, donde estamos moviendo a registros de la arquitectura direcciones relativas a la pila y operando con esas direcciones.

Otra diferencia entre O0 y O1, está al inicio de la función: en O1 miramos si el valor de n es negativo o 0, y en caso de serlo, ni se molesta en ejecutar el bucle, directamente abandona la función.

Entre O1 y O2 no hay tanta diferencia, de hecho, la única que veo es al calcular el valor de $x[i]*a$.

En Os, la diferencia es que para incrementar el valor de i usa la instrucción `incq`.

Por último, en O3, el compilador ha hecho un desenrollado del bucle de cuatro iteraciones (porque a la variable i le suma 16 (el tamaño de cuatro enteros) en cada iteración). Para ello, empaqueta cuatro enteros en los registros de coma flotante (con tamaño 128 bits) y los procesa paralelamente con instrucciones multimedia utilizando las extensiones SIMD del procesador.

CÓDIGO EN ENSAMBLADOR: (ADJUNTAR AL .ZIP) (LIMITAR AQUÍ EL CÓDIGO INCLUIDO A LA ZONA DEL CÓDIGO ENSAMBLADOR DONDE SE REALIZA LA OPERACIÓN CON VECTORES)

daxpy00.s

DAXPY:
.LFB2:

```

.cfi_startproc
pushq          %rbp                    # metemos los argumentos en la pila
.cfi_def_cfa_offset 16

```

```

.cfi_offset 6, -16
movq    %rsp, %rbp                # movemos el puntero de pila a rbp
.cfi_def_cfa_register 6
movq    %rdi, -24(%rbp)           # -24(%rbp) = y
movq    %rsi, -32(%rbp)           # -32(%rbp) = x
movl    %edx, -36(%rbp)           # -36(%rbp) = a
movl    %ecx, -40(%rbp)           # -40(%rbp) = n
movl    $0, -4(%rbp)              # i = 0
jmp     .L2

.L3:
movl    -4(%rbp), %eax            # eax = i
leaq    0(,%rax,4), %rdx          # obtener la posicion de memoria
movq    -24(%rbp), %rax          # rax = y
addq    %rax, %rdx               # llevar el puntero a la posicion de memoria
movl    -4(%rbp), %eax            # eax = i
leaq    0(,%rax,4), %rcx          # rcx = 4*i
movq    -24(%rbp), %rax          # rax = y
addq    %rcx, %rax               # rax = 4*i + y
movl    (%rax), %ecx             # ecx = rax
movl    -4(%rbp), %eax            # eax = i
leaq    0(,%rax,4), %rsi          # calculamos la posicion
movq    -32(%rbp), %rax          # guardamos x en rax
addq    %rsi, %rax               # llevar el puntero a la posicion de memoria
movl    (%rax), %eax             # guardamos el puntero en eax
imull   -36(%rbp), %eax          # eax = x[i]*a
addl    %ecx, %eax               # eax = x[i]*a + y
movl    %eax, (%rdx)             # rdx = eax
addl    $1, -4(%rbp)             # i++

.L2:
movl    -4(%rbp), %eax            # eax = i
cmpl    -40(%rbp), %eax          # compare n : i
jb      .L3                      # if i < n goto L3
nop
popq    %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc

```

daxpy01.s

```

DAXPY:
.LFB21:
.cfi_startproc
testl   %ecx, %ecx               # check n > 0
je      .L1                      # if n < 0 fin de la funcion
movl    $0, %eax                 # i = 0
.L3:                                     # bucle for
movl    %edx, %r8d               # r8d = a
imull   (%rsi,%rax,4), %r8d      # r8d *= 4*i + x
addl    %r8d, (%rdi,%rax,4)      # y[i] += r8d
addq    $1, %rax                 # i++
cmpl    %eax, %ecx               # compare i:n
ja      .L3                      # if n>i goto L3
.L1:
rep ret
.cfi_endproc

```

daxpy02.s

DAXPY:

```

.LFB21:
        .cfi_startproc
        xorl    %eax, %eax           # i = 0
        testl   %ecx, %ecx          # check n > 0
        je      .L1                 # if n < 0 fin de la funcion
        .p2align 4,,10
        .p2align 3

.L5:
        movl    (%rsi,%rax,4), %r8d  # bucle for
        imull    %edx, %r8d          # rd8 = x[i]
        addl     %r8d, (%rdi,%rax,4) # rd8 *= a
        addq     $1, %rax            # y[i] += rd8
        cmpl     %eax, %ecx          # i++
        ja      .L5                 # compare i:n
                                       # if n>i goto L5

.L1:
        rep ret
        .cfi_endproc

```

daxpy03.s

```

#-----
# utilizamos las instrucciones multimedia para desenrollar cuatro
# iteraciones del bucle empaquetando los datos en los registros de
# coma flotante y utilizando las extensiones SIMD del procesador
.L6:
        movdqu   (%rax,%rcx), %xmm0
        addl     $1, %r9d
        movdqa   %xmm0, %xmm1
        psrlq    $32, %xmm0
        pmuludq  %xmm3, %xmm0
        pshufd   $8, %xmm0, %xmm0
        pmuludq  %xmm2, %xmm1
        pshufd   $8, %xmm1, %xmm1
        punpckldq %xmm0, %xmm1
        movdqa   (%r10,%rcx), %xmm0
        paddb    %xmm1, %xmm0
        movaps   %xmm0, (%r10,%rcx)
        addq     $16, %rcx
        cmpl     %esi, %r9d
        jbe      .L6
#-----

```

daxpy0s.s

```

DAXPY:
.LFB2:
        .cfi_startproc
        xorl    %eax, %eax           # i = 0

.L2:
        cmpl     %eax, %ecx          # compare i:n
        jbe      .L5                 # if i<= n goto L5
        movl     (%rsi,%rax,4), %r8d  # r8d = x[i]
        imull    %edx, %r8d          # rd8 *= a
        addl     %r8d, (%rdi,%rax,4) # y[i] += rd8
        incq     %rax                # i++
        jmp      .L2                 # goto L2

.L5:
        ret
        .cfi_endproc

```