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): Antonio Gámiz Delgado
Grupo de prácticas: 2
Fecha de entrega: 30/05/2018
Fecha evaluación en clase: 30/05/2018

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): *Intel ® Core ™ i7-7400 HQ CPU @ 2.80GHz*

Sistema operativo utilizado: *Ubuntu 16.04 LTS*

Versión de gcc utilizada: *5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)*

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

```
antoniogamizdelgado 2018-05-11 viernes  @ antonio ~
└─ $ ▶ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 158
Model name:            Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Stepping:              9
CPU MHz:               800.017
CPU max MHz:           3800,0000
CPU min MHz:           800,0000
BogoMIPS:              5616.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dt
s acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
 rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl
vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes x
save avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti retpoline intel_pt rsb_
ctxsw tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdse
ed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_wind
ow hwp_epp
antoniogamizdelgado 2018-05-11 viernes  @ antonio ~
└─ $ ▶
```

1. Para el núcleo que se muestra en el Figura 1, 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.

   Debido a la larga extensión de los códigos en ensamblador, no los adjunto al cuaderno, sino que los dejo guardados en la carpeta assembly de la práctica.

   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`

```
for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        for(int k=0; k<N; k++)
            m3[i][j]+=m1[i][k] * m2[k][j];
```

Esta es la única parte que nos interesa, el resto se puede ver en el archivo pmm-secuencial.c de la carpeta src.

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: Lo que he hecho es trasponer la matriz m 2 para evitar los fallos de acceso y así recorrer ambas matrices por filas a la hora de hacer el producto.

Modificación b) –explicación-: Aquí he desarrollado el bucle que recorre las filas de m 2, es decir, el segundo bucle de los 3 que hay.

1.1. CÓDIGOS FUENTE MODIFICACIONES

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

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

int aux1, aux2, aux3, aux4;
for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        for(int k=0; k<N; k+=1){
            m3[i][j]+=m1[i][k]*m2[i][k];
        }
```

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

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

int aux1, aux2, aux3, aux4;
for(int i=0; i<N; i++)
    for(int j=0; j<N; j+=4){
        aux1=0; aux2=0; aux3=0; aux4=0;
        for(int k=0; k<N; k+=1){
            aux1+=m1[i][k] * m2[j][k];
            aux2+=m1[i][k] * m2[j+1][k];
            aux3+=m1[i][k] * m2[j+2][k];
            aux4+=m1[i][k] * m2[j+3][k];
        }
        m3[i][j]=aux1;
        m3[i][j+1]=aux2;
        m3[i][j+2]=aux3;
        m3[i][j+3]=aux4;
    }
```

1.1. TIEMPOS:

Modificación	**-O2**
Sin modificar	*45.220127201*
Modificación a)	4.885865804
Modificación b)	3.020288166

```
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -O2 ./src/pmm-secuencial.c -o ./bin/pmm-secuencial; ./bin/pmm-secuencial 2000; gcc -O2 ./src/pmm-secuencial-a.c -o ./bin/pmm-secuencial-a; ./bin/pmm-secuencial-a 2000; gcc -O2 ./src/pmm-secuencial-b.c -o ./bin/pmm-secuencial-b; ./bin/pmm-secuencial-b 2000
Tiempo(seg): 45.220127201       N=2000          (m3[0][0]=2000 m3[1999][1999]=2000)
Tiempo(seg): 4.885865804        N=2000          (m3[0][0]=2000 m3[1999][1999]=2000)
Tiempo(seg): 3.020288166        N=2000          (m3[0][0]=2000 m3[1999][1999]=2000)
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶
```

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

Como claramente se ve en los tiempos, el desenrrollado de bucles es efectivo pero reducir considerablemente el número de fallos de acceso a memoria es crucial para la eficiencia de nuestros programas.

1.2. CÓDIGO EN ENSAMBLADOR DEL ORIGINAL Y DE DOS MODIFICACIONES : (PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR EVALUADA, USE COLORES PARA DESTACAR LAS DIFERENCIAS)

pmm-secuencial.s

```
.L10:
        leaq    (%r14,%r13), %r8
        movq    %rbp, %rdi
        .p2align 4,,10
        .p2align 3
.L8:
        movl    (%r8), %esi
        leaq    (%r10,%rdi), %rax
        movq    %r9, %rcx
        .p2align 4,,10
        .p2align 3
.L7:
        movl    (%rcx), %edx
        addq    $40000, %rax
        addq    $4, %rcx
        imull   -40000(%rax), %edx
        addl    %edx, %esi
        cmpq    %rdi, %rax
        jne     .L7
        movl    %esi, (%r8)
        addq    $4, %r8
        leaq    4(%rax), %rdi
        cmpq    %r8, %r13
        jne     .L8
        addq
```

pmm-secuencial-modificado_b.s

```
.L8:
        leaq    (%r10,%r8), %rdi
        movq    %r8, %rax
        movq    %r9, %rdx
        .p2align 4,,10
        .p2align 3
.L7:
        movl    (%rdx), %ecx
        movl    (%rax), %esi
        addq    $40000, %rax
        addq    $4, %rdx
        movl    %esi, -4(%rdx)
        movl    %ecx, -40000(%rax)
        cmpq    %rdi, %rax
        jne     .L7
        addq    $4, %r8
        addq    $40000, %r9
        cmpq    %r12, %r8
        jne     .L8
        xorl    %esi, %esi
        xorl    %r8d, %r8d
.L13:
        xorl    %edx, %edx
        .p2align 4,,10
        .p2align 3
```

pmm-secuencial-modificado_c.s

```
.L11:
        leaq    40000(%r10), %rbp
        leaq    80000(%r10), %r11
        leaq    120000(%r10), %r9
        xorl    %eax, %eax
        xorl    %r8d, %r8d
        xorl    %edi, %edi
        xorl    %esi, %esi
        xorl    %ecx, %ecx
        .p2align 4,,10
        .p2align 3
.L10:
        movl    m1(%r13,%rax,4), %edx
        movl    m2(%r10,%rax,4), %r12d
        imull   %edx, %r12d
        addl    %r12d, %ecx
        movl    m2(%rbp,%rax,4), %r12d
        imull   %edx, %r12d
        addl    %r12d, %esi
        movl    m2(%r11,%rax,4), %r12d
        imull   %edx, %r12d
        imull   m2(%r9,%rax,4), %edx
        addq    $1, %rax
        addl    %r12d, %edi
        addl    %edx, %r8d
        cmpl    %eax, %ebx
```

```
        $40000, %r13
        addq
        $40000, %r9
        cmpq
        %r11, %r13
        jne
        .L10
```

```
.L11:
                movl
                m3(%rsi,
%rdx,4), %edi
                xorl
                %eax, %eax
                .p2align 4,,10
                .p2align 3
.L10:
                movl
                m1(%rsi,
%rax,4), %ecx
                imull
                m2(%rsi,
%rax,4), %ecx
                addq
                $1, %rax
                addl
                %ecx, %edi
                cmpl
                %eax, %ebx
                jg
                .L10
                movl
                %edi, m3(%rsi,
%rdx,4)
                addq
                $1, %rdx
                cmpl
                %edx, %ebx
                jg
                .L11
                addl
                $1, %r8d
                addq
                $40000, %rsi
                cmpl
                %ebx, %r8d
                jl
                .L13
```

```
        jg
        .L10
        addl
        $4, %r15d
        movl
        %ecx, (%r14)
        movl
        %esi, 4(%r14)
        movl
        %edi, 8(%r14)
        movl
        %r8d, 12(%r14)
        addq
        $160000, %r10
        addq
        $16, %r14
        cmpl
        %ebx, %r15d
        jl
        .L11
        addl
        $1, 8(%rsp)
        addq
        $40000, %r13
        movl
        8(%rsp), %eax
        cmpl
        %ebx, %eax
        jl
        .L13
```

Del primer código al segundo vemos que es bastante más largo, aunque el segundo es mucho más corto, esto se debe a que en la segunda también apareece el código que transpone la matriz m 2. La mejora de tiempo entre la versión normal y la modificada-a no se ve en el código ensamblador ya que se basa en la reducción de loss fallos de memoria.

Del segundo al tercero vemos un poco más de código que en el segundo, esto se debe a que al haber desenrrollado, hay que ajustar más variables dentro del bucle.

B) CÓDIGO FIGURA 1:
CAPTURA CÓDIGO FUENTE: `figura1-original.c`

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

int x1, x2;

struct timespec cgt1, cgt2;
double ncgt;
clock_gettime(CLOCK_REALTIME, &cgt1);

for(int ii=0; ii<40000; ii++)
{
    x1=0; x2=0;
    for(int i=0; i<5000; i++) x1+=2*s[i].a+ii;
    for(int i=0; i<5000; i++) x2+=3*s[i].b-ii;

    if( x1<x2 ) R[ii]=x1; else R[ii]=x2;

}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-:

En la primera modificación hemos eliminado uno de los bucles, ya que los dos hacían el mismo recorrido, ahorrándonos 5000 iteraciones por cada ejecución del bucle exterior, que se ejecúta 40000 veces, ahorrándonos entonces 200 millones de iteraciones. Reduciéndo así, como se verá en la captura del final, el tiempo de ejecución a prácticamente la mitad.

Modificación b) –explicación-:

En la segunda modificación he visto que el bucle interior realmente no hacía falta, que lo que hacía era simplemente sumar 5000 veces la misma cantidad, así que con multiplicar esa cantidad por 5000 y sumarla obtendríamos el mismo resultado. De la misma forma con la multiplicación por 2 y 3, he calculado la suma del struct a parte, y luego la he calculado y la guardado en una variable, evitándome así calcularlo en cada iteración del bucle exterior.

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) Captura figura1`-modificado_a.c`

```
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;

}
```

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

```
x1=0; x2=0;
for(int i=0; i<5000; i++)
{
    x1+=s[i].a;
    x2+=s[i].b;
}

x1*=2;
x2*=3;

int x1_aux, x2_aux;
for(int ii=0; ii<40000; ii++)
{
    x1_aux=x1; x2_aux=x2;
    x1_aux+=5000*ii;
    x2_aux-=5000*ii;


    if( x1_aux<x2_aux ) R[ii]=x1_aux; else R[ii]=x2_aux;

}
```

```
antoniogamizdelgado 2018-05-24 jueves  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -O2 ./src/f1.c -o ./bin/f1; ./bin/f1; gcc -O2 ./src/f1_1.c -o ./bin/f1_1; ./bin/f1_1; gcc -O2 ./src/f1_2.c -o ./bin/f1_2; ./bin/f1_2;
Tiempo de ejecución: 0.216326492         / R[0]= 24602044        / R[39999]= -162887433
Tiempo de ejecución: 0.145843147         / R[0]= 24602044        / R[39999]= -162887433
Tiempo de ejecución: 0.000063795         / R[0]= 24602044        / R[39999]= -162887433
antoniogamizdelgado 2018-05-24 jueves  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶
```

1.1. TIEMPOS:

Modificación	**-O2**
Sin modificar	0.216326492
Modificación a)	0.145843147
Modificación b)	0.000063795

1.1. COMENTARIOS SOBRE LOS RESULTADOS:

Con este ejercicio vemos la importancia de optimizar los programas secuenciales también y no sólo intentar ejecutarlos en paralelo, ya que sin paralelizar hemos obtenido una mejora en la eficiencia unas 1000 veces mayor analizando qué pasaba en el código.

1.2. CÓDIGO EN ENSAMBLADOR DEL ORIGINAL Y DE DOS MODIFICACIONES: (PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR EVALUADA, USE COLORES PARA DESTACAR LAS DIFERENCIAS)

figura1.s

```
.L3:
                movl    %r8d, %edi
                movl    $s, %eax
                xorl    %esi, %esi
                .p2align 4,,10
                .p2align 3
.L4:
                movl    (%rax), %edx
                addq    $8, %rax
                leal    (%rdi,%rdx,2),
%edx
                addl    %edx, %esi
                cmpq    $s+40000, %rax
                jne     .L4
                movl    $s+4, %eax
                xorl    %ecx, %ecx
                .p2align 4,,10
                .p2align 3
.L5:
                movl    (%rax), %edx
                addq    $8, %rax
                leal    (%rdx,%rdx,2),
%edx
                subl    %edi, %edx
                addl    %edx, %ecx
                cmpq    $s+40004, %rax
                jne     .L5
                cmpl    %ecx, %esi
                cmovl   %esi, %ecx
                movl
```

figura1-modificado_a.s

```
.L3:
                movl    %r8d, %edi
                movl    $s, %eax
                xorl    %ecx, %ecx
                xorl    %esi, %esi
                .p2align 4,,10
                .p2align 3
.L4:
                movl    (%rax), %edx
                addq    $8, %rax
                leal    (%rdi,%rdx,2),
%edx
                addl    %edx, %esi
                movl    -4(%rax), %edx
                leal    (%rdx,%rdx,2),
%edx
                subl    %edi, %edx
                addl    %edx, %ecx
                cmpq    $s+40000, %rax
                jne     .L4
                cmpl    %ecx, %esi
                cmovl   %esi, %ecx
                movl    %ecx, R(,%r8,4)
                addq    $1, %r8
                cmpq    $40000, %r8
                jne     .L3
                leaq    16(%rsp), %rsi
                xorl    %edi, %edi
```

figura1-modificado_b.s

```
.L3:
                addl    0(%rbp), %eax
                addl    4(%rbp), %edx
                addq    $8, %rbp
                cmpq    $s+40000, %rbp
                jne     .L3
                leal    (%rdx,%rdx,2),
%ecx
                addl    %eax, %eax
                movl    $R, %edx
                movl    $R+160000, %esi
                jmp     .L6
                .p2align 4,,10
                .p2align 3
.L14:
                movl    %eax, (%rdx)
.L5:
                addq    $4, %rdx
                addl    $5000, %eax
                subl    $5000, %ecx
                cmpq    %rdx, %rsi
                je      .L13
.L6:
                cmpl    %ecx, %eax
                jl      .L14
                movl    %ecx, (%rdx)
                jmp     .L5
                .p2align 4,,10
                .p2align 3
```

%ecx, R(,%r8,4) addq $1, %r8 cmpq $40000, %r8 jne .L3 leaq 16(%rsp), %rsi xorl %edi, %edi		

De la primera columna a la segunda, vemos que, hay un bucle menos, es decir, todo el códigoo de .L4 no es ejecutado en cada iteración del bucle grande por lo que nos ahorramos bastante tiempo.
Del segundo al tercero vemos que hay el mismo número de bucles, pero la cantidad de instrucciones que tiene que ejecutar cada bucle es bastante inferior a la de los bucles en el segundo.

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 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.

2.2. (Ejercicio EXTRA) Para la mejor de las opciones, obtenga los tiempos de ejecución con distintos valores de N y determine para su sistema los valores de Rmax (valor máximo del número de operaciones en coma flotante por unidad de tiempo), Nmax (valor de N para el que se consigue Rmax), y N1/2 (valor de N para el que se obtiene Rmax/2). Estime el valor de la velocidad pico (Rpico) del procesador (consulte en [4] el número de ciclos por instrucción punto flotante para la familia y modelo de procesador que está utilizando) y compárela con el valor obtenido para Rmax. -Consulte la Lección 3 del Tema 1.

CAPTURA CÓDIGO FUENTE: `daxpy.c`

```c
#define MAX 300000000
float x[MAX], y[MAX];

#define ALPHA 1.5

int main(int argc, char **argv) {
    if( argc != 2 ) {
        printf("Formato: %s <N> \n", argv[0]);
        exit(-1);
    }

    int N = atoi(argv[1]); if( N > MAX ) N=MAX;

    for(int i=0; i<N; i++) x[i]=i+0.1;

    struct timespec cgt1, cgt2;
    double ncgt;

    clock_gettime(CLOCK_REALTIME, &cgt1);

    for(int i=0; i<N; i++) y[i] = ALPHA * x[i] + y[i];

    clock_gettime(CLOCK_REALTIME, &cgt2);

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

    printf("T(s) %11.9f \t / y[0]=%5.3f / y[%d]=%5.3f\n", ncgt, y[0], N, y[N-1]);

}
```

Tiempos ejec.	-O0	-Os	-O2	-O3
	2.722250981	1.712244910	1.835209745	1.284168224

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

```
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -O0 ./src/daxpy.c -o ./bin/daxpy-O0; ./bin/daxpy-O0 300000000
T(s) 2.722250981         / y[0]=0.150 / y[300000000]=450000000.000
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -Os ./src/daxpy.c -o ./bin/daxpy-Os; ./bin/daxpy-Os 300000000
T(s) 1.712244910         / y[0]=0.150 / y[300000000]=450000000.000
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -O2 ./src/daxpy.c -o ./bin/daxpy-O2; ./bin/daxpy-O2 300000000
T(s) 1.835209745         / y[0]=0.150 / y[300000000]=450000000.000
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -O3 ./src/daxpy.c -o ./bin/daxpy-O3; ./bin/daxpy-O3 300000000
T(s) 1.284168224         / y[0]=0.150 / y[300000000]=450000000.000
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶
```

COMENTARIOS QUE EXPLIQUEN LAS DIFERENCIAS EN ENSAMBLADOR:

Generación del código ensamblador:

```
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -S -O0 ./src/daxpy.c
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -S -Os ./src/daxpy.c
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -S -O2 ./src/daxpy.c
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶ gcc -S -O3 ./src/daxpy.c
antoniogamizdelgado 2018-05-25 viernes  @ antonio ~/ArquitecturaDeComputadores/Práctica 5 (master)
└─ $ ▶
```

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

```
.L7:
        movl    -60(%rbp), %eax
        cltq
        movss   x(,%rax,4), %xmm0
        cvtss2sd        %xmm0, %xmm0
        movsd   .LC2(%rip), %xmm1
        mulsd   %xmm0, %xmm1
        movl    -60(%rbp), %eax
        cltq
        movss   y(,%rax,4), %xmm0
        cvtss2sd        %xmm0, %xmm0
        addsd   %xmm1, %xmm0
        cvtsd2ss        %xmm0, %xmm0
        movl    -60(%rbp), %eax
        cltq
        movss   %xmm0, y(,%rax,4)
        addl    $1, -60(%rbp)
.L6:
        movl    -60(%rbp), %eax
        cmpl    -68(%rbp), %eax
        jl      .L7
```

daxpyOs.s

```
.L5:
        cmpl    %eax, %ebx
        jle     .L11
        cvtss2sd        x(,%rax,4), %xmm0
        cvtss2sd        y(,%rax,4), %xmm1
        mulsd   %xmm2, %xmm0
        addsd   %xmm1, %xmm0
        cvtsd2ss        %xmm0, %xmm4
        movss   %xmm4, y(,%rax,4)
        incq    %rax
        jmp     .L5
.L11:
        leaq    24(%rsp), %rsi
        xorl    %edi, %edi
```

daxpy02.s

```
.L6:
        pxor    %xmm0, %xmm0
        addq    $4, %rax
        pxor    %xmm1, %xmm1
        pxor    %xmm4, %xmm4
        cvtss2sd    x-4(%rax), %xmm0
        cvtss2sd    y-4(%rax), %xmm1
        mulsd   %xmm2, %xmm0
        addsd   %xmm1, %xmm0
        cvtsd2ss    %xmm0, %xmm4
        movss   %xmm4, y-4(%rax)
        cmpq    %rax, %rbx
        jne     .L6
```

daxpy03.s

```
.L10:
        addl    $1, %edx
        addq    $16, %rax
        movaps  x-16(%rax), %xmm2
        cvtps2pd    %xmm2, %xmm0
        movaps  y-16(%rax), %xmm4
        mulpd   %xmm3, %xmm0
        movhps  %xmm2, (%rsp)
        movhps  %xmm4, 16(%rsp)
        movapd  %xmm0, %xmm1
        cvtps2pd    %xmm4, %xmm0
        addpd   %xmm1, %xmm0
        cvtps2pd    (%rsp), %xmm1
        cvtpd2ps    %xmm0, %xmm0
        movapd  %xmm1, %xmm2
        cvtps2pd    16(%rsp), %xmm1
        mulpd   %xmm3, %xmm2
        addpd   %xmm2, %xmm1
        cvtpd2ps    %xmm1, %xmm1
        movlhps %xmm1, %xmm0
        movaps  %xmm0, y-16(%rax)
        cmpl    %edx, %r13d
        ja      .L10
        cmpl    %ebp, %ebx
        movl    %ebp, %eax
        je      .L15
```

En la anterior tabla podemos ver los 4 códigos máquina generados por el compilador para el bucle daxpy. Como vemos, en el primer código el cálculo de la multiplicación se realiza en más instrucciones máquina y además se llama a clqt, para cambiar un entero a un entero de 64 bits. En el segundo vemos que ésto último se usa y que además cálcula la multiplicación exactamente igual que en el tercer código, es decir, entre el segundo y el tercero, la única diferencia es la forma de actualizar los índices para las iteraciones. En el cuarto código, con -O3, el código ensamblador generado es mucho más largo (no solo el trozo del bucle, si no todo el código en general: ver los archivos en la carpeta assembly). La verdad es que no sé porqué este último código sale bastante más rápido que todos los otros si tiene muchas más instrucciones, supongo que es porque utiliza instrucciones que requieran menos ciclos o alguna optimización de ese estilo.