Cuaderno de prácticas
de Arquitectura de Computadores
Grado en Ingeniería Informática

Memoria
Bloque Práctico 4

Alumno: Miguel Sánchez Tello
DNI: 75574961C
Grupo: *B1*

Versión de gcc utilizada: `gcc versión 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)`

Fichero /proc/cpuinfo de la máquina en la que ha tomado las medidas:

(Texto excesivamente largo. La respuesta está en el fichero adjunto *cpuinfo*).

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,...). 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.
    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

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

main()
{
   …
   for (ii=1; ii<=40000;ii++) {
      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;
   }
   …
}
```

Figura 1: Núcleo de programa en C para el ejercicio 1.

A) MULTIPLICACIÓN DE MATRICES:

CÓDIGO FUENTE: pmm-secuencial-modificado.c
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

Para este ejercicio, hay 3 códigos fuente adjuntos en el fichero de entrega de esta práctica:

pmm-secuencial-modificado.c: Se trata del código original.

pmm-secuencial-modificado_accesos.c: Es la versión anterior, mejorando los accesos a memoria cuando inicializamos las matrices *B* y *C*. La forma de hacer esto es dividir el bucle inicializador en dos. Esto implica que habrá el doble de STALLs por salto. Lo he implementado para ilustrar un mal ejemplo de optimización, pese a que se mejore en un aspecto hay que tener en cuenta el impacto que tenemos en el resto del programa.

pmm-secuencial -modificado_desenrollado.c: Se trata de la versión anterior (la 2ª) aplicando un desenrollado de bucles x3. Dicha cifra se ha obtenido por ensayo y error (por ejemplo, un desenrollado x5 es peor en este programa).

TIEMPOS:

Versión original:

```
----------O0----------
Tiempo de inicialización: 0.919238
Tiempo de cálculo: 2.287154
TIEMPO TOTAL: 3.206393
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O1----------
Tiempo de inicialización: 0.541504
Tiempo de cálculo: 1.808802
TIEMPO TOTAL: 2.350305
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O2----------
Tiempo de inicialización: 0.538563
Tiempo de cálculo: 1.800650
TIEMPO TOTAL: 2.339213
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O3----------
Tiempo de inicialización: 0.516308
Tiempo de cálculo: 1.810534
TIEMPO TOTAL: 2.326841
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------Os----------
Tiempo de inicialización: 0.537818
Tiempo de cálculo: 1.800604
TIEMPO TOTAL: 2.338422
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
```

Versión de accesos mejorados

```
----------O0----------
Tiempo de inicialización: 1.053027
Tiempo de cálculo: 2.276340
TIEMPO TOTAL: 3.329367
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O1----------
Tiempo de inicialización: 0.627252
Tiempo de cálculo: 1.835912
TIEMPO TOTAL: 2.463164
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O2----------
Tiempo de inicialización: 0.621598
Tiempo de cálculo: 1.794639
TIEMPO TOTAL: 2.416237
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O3----------
Tiempo de inicialización: 0.515097
Tiempo de cálculo: 1.784562
TIEMPO TOTAL: 2.299660
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------Os----------
Tiempo de inicialización: 0.618702
Tiempo de cálculo: 1.791000
TIEMPO TOTAL: 2.409702
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
```

Versión de desenrollado

```
----------O0----------
Tiempo de inicialización: 0.885139
Tiempo de cálculo: 1.078425
TIEMPO TOTAL: 1.963563
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O1----------
Tiempo de inicialización: 0.626644
Tiempo de cálculo: 0.846559
TIEMPO TOTAL: 1.473202
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O2----------
Tiempo de inicialización: 0.617400
Tiempo de cálculo: 0.826801
TIEMPO TOTAL: 1.444201
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------O3----------
Tiempo de inicialización: 0.625829
Tiempo de cálculo: 0.812891
TIEMPO TOTAL: 1.438720
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
----------Os----------
Tiempo de inicialización: 0.615628
Tiempo de cálculo: 0.812169
TIEMPO TOTAL: 1.427797
matrizA[0][0]=0.000000 /
matrizA[9999][9999]=399920004.000000  /
```

COMENTARIOS SOBRE LOS RESULTADOS:

- Efectivamente, aunque accedamos de forma más eficiente a cada matriz cuando se inicializa en la versión 2, salimos perdiendo porque ahora tenemos dos bucles for en vez de uno.

- Mejoramos mucho la parte de cálculos con la 3ª versión al introducir el desenrollado de bucles en ambas partes, mientras que no conseguimos compensar la pérdida anterior en la parte de inicialización.

B) CÓDIGO FIGURA 1:

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

Para este ejercicio, hay 3 códigos fuente adjuntos en el fichero de entrega de esta práctica:

figura1-modificado.c: Se trata del código original.

figura1-modificado_accesos.c: Es la versión anterior, mejorándole los accesos a la estructura s. Se hace de tal forma que se recorran los elementos en un orden "abababababab" en vez de "aaaaaaaaabbbbbbb", pues se realizarían muchos saltos en memoria.

figura1-modificado_desen.c: Se trata de la versión anterior (la 2ª) aplicando un desenrollado de bucles x3. Dicha cifra se ha obtenido por ensayo y error (por ejemplo, un desenrollado x5 es peor en este programa).

TIEMPOS:

Versión original:

```
----------O0----------
Tiempo de inicialización: 0.000026
Tiempo de cálculo: 1.046816
TIEMPO TOTAL: 1.046842
R[0]=0 R[40000]=-729000961
----------O1----------
Tiempo de inicialización: 0.000013
Tiempo de cálculo: 0.300207
TIEMPO TOTAL: 0.300219
R[0]=0 R[40000]=-729033728
----------O2----------
Tiempo de inicialización: 0.000014
Tiempo de cálculo: 0.283300
TIEMPO TOTAL: 0.283314
R[0]=0 R[40000]=-724837592
----------O3----------
Tiempo de inicialización: 0.000011
Tiempo de cálculo: 0.197569
TIEMPO TOTAL: 0.197580
R[0]=0 R[40000]=-724837188
----------Os----------
Tiempo de inicialización: 0.000012
Tiempo de cálculo: 0.277850
TIEMPO TOTAL: 0.277862
R[0]=0 R[40000]=-724837612
----------
```

Versión de accesos mejorados

```
----------O0----------
Tiempo de inicialización: 0.000026
Tiempo de cálculo: 0.675161
TIEMPO TOTAL: 0.675187
R[0]=0 R[40000]=-1631691680
----------O1----------
Tiempo de inicialización: 0.000012
Tiempo de cálculo: 0.204717
TIEMPO TOTAL: 0.204729
R[0]=0 R[40000]=-729033728
----------O2----------
Tiempo de inicialización: 0.000013
Tiempo de cálculo: 0.206363
TIEMPO TOTAL: 0.206377
R[0]=0 R[40000]=-724837608
----------O3----------
Tiempo de inicialización: 0.000012
Tiempo de cálculo: 0.182550
TIEMPO TOTAL: 0.182562
R[0]=0 R[40000]=-1115745104
----------Os----------
Tiempo de inicialización: 0.000012
Tiempo de cálculo: 0.201548
TIEMPO TOTAL: 0.201560
R[0]=0 R[40000]=-729033728
----------
```

Versión de desenrollado

```
----------O0----------
Tiempo de inicialización: 0.000021
Tiempo de cálculo: 0.546559
TIEMPO TOTAL: 0.546580
R[0]=0 R[40000]=-1253086176
----------O1----------
Tiempo de inicialización: 0.000012
Tiempo de cálculo: 0.179506
TIEMPO TOTAL: 0.179519
R[0]=0 R[40000]=-1253086176
----------O2----------
Tiempo de inicialización: 0.000013
Tiempo de cálculo: 0.170520
TIEMPO TOTAL: 0.170534
R[0]=0 R[40000]=-1253086176
----------O3----------
Tiempo de inicialización: 0.000010
Tiempo de cálculo: 0.174656
TIEMPO TOTAL: 0.174666
R[0]=0 R[40000]=-1253086176
----------Os----------
Tiempo de inicialización: 0.000015
Tiempo de cálculo: 0.154625
TIEMPO TOTAL: 0.154640
R[0]=0 R[40000]=-1253086176
----------
```

COMENTARIOS SOBRE LOS RESULTADOS:

- La única diferencia entre los códigos de la 1ª y 2ª versión está en la parte de los cálculos, haciendo que se acceda de forma mucho más eficiente a los elementos de *s*. Los tiempos de la inicialización son iguales. De forma aproximada, obtenemos 1/3 más de rendimiento con la segunda versión (rendimiento atendiendo a los tiempos).

- La 3ª versión afecta a ambas partes: mejoramos entre 0.03 y 0.05 unidades de tiempo en la parte del cálculo en sí, mientras que no es apreciable una mejoría o pérdida en la parte de inicialización.

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];

a. 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 acarrean.

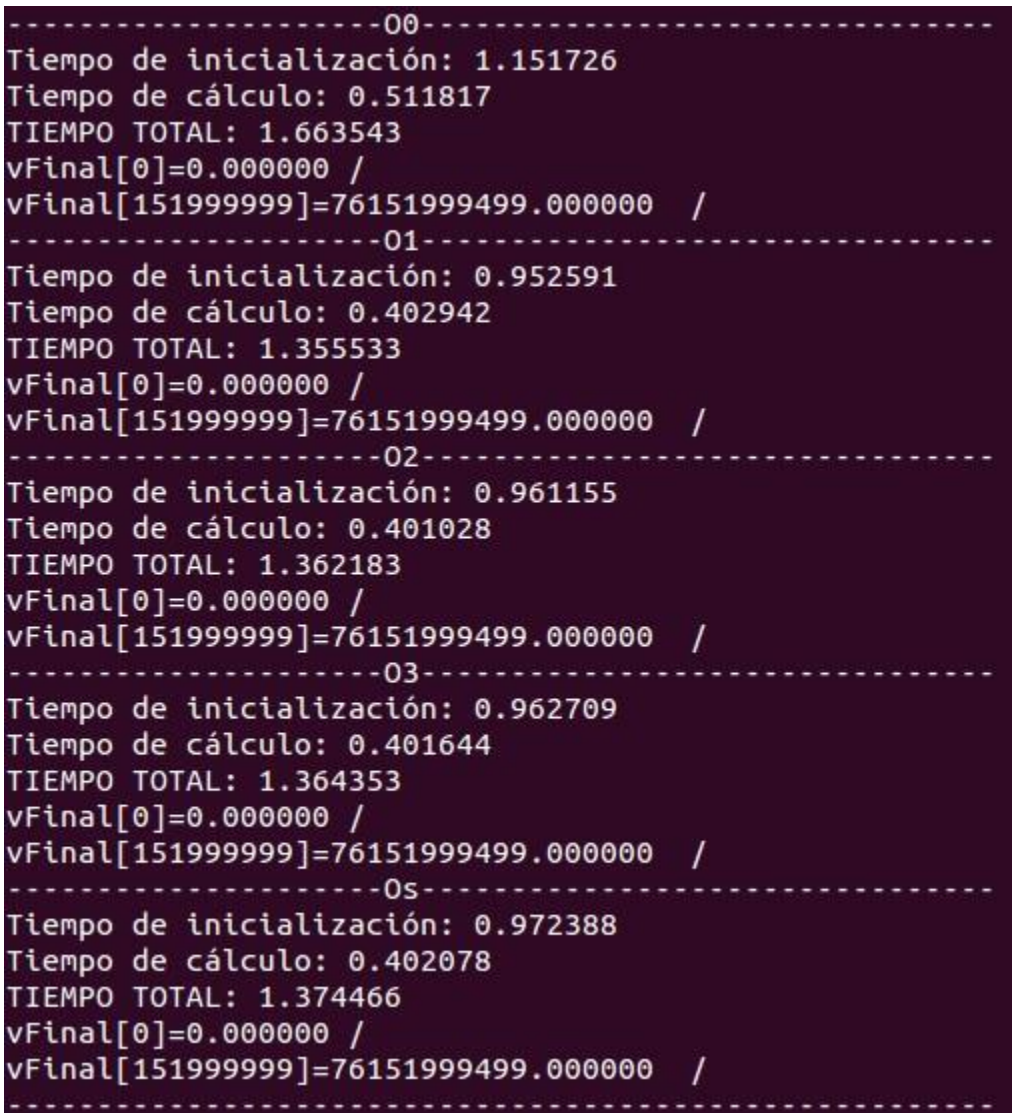
b. (Ejercicio EXTRA) Para la mejor de las opciones, obtenga los tiempos de ejecución con distintos valores de N y determine para su sistema los valores de Rmax (valor máximo del número de operaciones en coma flotante por unidad de tiempo), Nmax (valor de N para el que se consigue Rmax), y N1/2 (valor 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) y compárela con el valor obtenido para Rmax.

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

```
// gcc daxpy.c -o daxpy-0 -lrt
// gcc -O1 daxpy.c -o daxpy-1 -lrt
// gcc -O2 daxpy.c -o daxpy-2 -lrt
// gcc -O3 daxpy.c -o daxpy-3 -lrt
// gcc -Os daxpy.c -o daxpy-4 -lrt

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

int main(int argc, char **argv){
            int n;
```

```
            int i;
            struct timespec cgt1_inic,cgt2_inic; double ncgt_inic;
//Para la inicialización.
            struct timespec cgt1_calc,cgt2_calc; double ncgt_calc;
//Para los cálculos.

            n =152000000;
            //    |  |

            double *vector;
            double *vFinal;
            double alpha=500;

            //Creación de vectores
            vector = (double*) malloc(n*sizeof(double));
            vFinal  = (double*) malloc(n*sizeof(double));

            //Inicialización
            clock_gettime(CLOCK_REALTIME,&cgt1_inic);

            for(i=0;i<n;i+=3){vector[i]=i; vector[i+1]=i+1;
vector[i+2]=i+2;}
            for(i=0;i<n;i+=3){vFinal[i]=i; vFinal[i+1]=i+1;
vFinal[i+2]=i+2;}

            clock_gettime(CLOCK_REALTIME,&cgt2_inic);
            ncgt_inic=  (double) (cgt2_inic.tv_sec-cgt1_inic.tv_sec)+
                        (double) ((cgt2_inic.tv_nsec-
cgt1_inic.tv_nsec)
                        /(1.e+9));

            clock_gettime(CLOCK_REALTIME,&cgt1_calc);
/* ------------CUERPO DEL PROGRAMA----------  */
            for(i=0;i<n;i+=3){
                        vFinal[i] = alpha*vector[i] + vFinal[i];
                        vFinal[i+1] = alpha*vector[i+1] + vFinal[i+1];
                        vFinal[i+2] = alpha*vector[i+2] + vFinal[i+2];
            }
/* ---------------------------------------- */
            clock_gettime(CLOCK_REALTIME,&cgt2_calc);
            ncgt_calc=  (double) (cgt2_calc.tv_sec-cgt1_calc.tv_sec)+
                        (double) ((cgt2_calc.tv_nsec-
cgt1_calc.tv_nsec)
                        /(1.e+9));
            printf("Tiempo de inicialización: %f \n", ncgt_inic);
            printf("Tiempo de cálculo: %f \n", ncgt_calc);
            printf("TIEMPO TOTAL: %f \n", ncgt_inic+ncgt_calc);

/* -- */
            printf("vFinal[0]=%f /\nvFinal[%d]=%f  /\n",vFinal[0],n-
1,vFinal[n-1]);

/* -- */
□□          return 0;□}
```

CAPTURAS DE PANTALLA:



COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:
(Las diferencias van junto a cada código, más abajo).
He implementado el programa tal y como se nos pide en el enunciado, así como en la transparencia del Tema1. En el fichero inicial de esta práctica venía una tabla con tiempos a rellenar, con una especificación de tiempo: tenía que ser mínimo 10 segundos con la versión sin optimización. He trabajado con tipos de dato *doublé* e *int*, pero con ninguno de ellos consigo llegar a los 2 segundos de ejecución. He utilizado la versión con los *double*. Utilizando 152 millones de elementos en cada vector he llegado a llenar casi por completo los 3.8gb de memoria disponible del sistema, y apenas llegar al segundo y medio de tiempo de ejecución.
No sé qué es lo que debo de estar haciendo mal, no hay mucho donde equivocarse, el núcleo del programa es simple.

CÓDIGO EN ENSAMBLADOR:

daxpy00.s

```
.L7:
        movl    -8(%rbp), %eax
        cltq
        salq    $3, %rax
        addq    -32(%rbp), %rax
        movl    -8(%rbp), %edx
        movslq  %edx, %rdx
        salq    $3, %rdx
        addq    -40(%rbp), %rdx
        movsd   (%rdx), %xmm0
        mulsd   -48(%rbp), %xmm0
        movl    -8(%rbp), %edx
        movslq  %edx, %rdx
        salq    $3, %rdx
        addq    -32(%rbp), %rdx
        movsd   (%rdx), %xmm1
        addsd   %xmm1, %xmm0
        movsd   %xmm0, (%rax)
        movl    -8(%rbp), %eax
        cltq
        addq    $1, %rax
        salq    $3, %rax
        addq    -32(%rbp), %rax
        movl    -8(%rbp), %edx
        movslq  %edx, %rdx
        addq    $1, %rdx
        salq    $3, %rdx
        addq    -40(%rbp), %rdx
        movsd   (%rdx), %xmm0
        mulsd   -48(%rbp), %xmm0
        movl    -8(%rbp), %edx
        movslq  %edx, %rdx
        addq    $1, %rdx
        salq    $3, %rdx
        addq    -32(%rbp), %rdx
        movsd   (%rdx), %xmm1
        addsd   %xmm1, %xmm0
        movsd   %xmm0, (%rax)
        movl    -8(%rbp), %eax
        cltq
        addq    $2, %rax
        salq    $3, %rax
        addq    -32(%rbp), %rax
        movl    -8(%rbp), %edx
        movslq  %edx, %rdx
        addq    $2, %rdx
        salq    $3, %rdx
        addq    -40(%rbp), %rdx
        movsd   (%rdx), %xmm0
        mulsd   -48(%rbp), %xmm0
        movl    -8(%rbp), %edx
        movslq  %edx, %rdx
        addq    $2, %rdx
        salq    $3, %rdx
        addq    -32(%rbp), %rdx
        movsd   (%rdx), %xmm1
        addsd   %xmm1, %xmm0
```

```
            movsd       %xmm0, (%rax)
            addl        $3, -8(%rbp)
```

daxpy01.s

```
.L4:
//Primer elemento
            movsd       0(%rbp,%rax), %xmm1
            mulsd       %xmm0, %xmm1
            addsd       (%rbx,%rax), %xmm1
            movsd       %xmm1, (%rbx,%rax)
//Segundo elemento
            movsd       8(%rbp,%rax), %xmm1
            mulsd       %xmm0, %xmm1
            addsd       8(%rbx,%rax), %xmm1
            movsd       %xmm1, 8(%rbx,%rax)
//Segundo elemento
            movsd       16(%rbp,%rax), %xmm1
            mulsd       %xmm0, %xmm1
            addsd       16(%rbx,%rax), %xmm1
            movsd       %xmm1, 16(%rbx,%rax)
//Contador y salto
            addq        $24, %rax
            cmpq        $1216000008, %rax
            jne         .L4
            leaq        64(%rsp), %rsi
            movl        $0, %edi
```

daxpy02.s: La única diferencia con el anterior está en que se utilizan otros registros. Seguramente la diferencia que hace que esta versión sea más eficiente esté fuera de esta sección de código

```
.L4:
//Primer elemento
            movsd       0(%rbp,%rax), %xmm0
            mulsd       %xmm1, %xmm0
            addsd       (%rbx,%rax), %xmm0
            movsd       %xmm0, (%rbx,%rax)
//Segundo elemento
            movsd       8(%rbp,%rax), %xmm0
            mulsd       %xmm1, %xmm0
            addsd       8(%rbx,%rax), %xmm0
            movsd       %xmm0, 8(%rbx,%rax)
//Tercer elemento
            movsd       16(%rbp,%rax), %xmm0
            mulsd       %xmm1, %xmm0
            addsd       16(%rbx,%rax), %xmm0
            movsd       %xmm0, 16(%rbx,%rax)
//Contador y salto
            addq        $24, %rax
            cmpq        $1216000008, %rax
            jne         .L4
            leaq        80(%rsp), %rsi
            xorl        %edi, %edi
```

daxpy03.s: Se utilizan instrucciones de mover double-precision floating-point con funcionalidad avanzada para realizar las sumas, productos y almacenamientos, aunque se sigue utilizando el viejo remedio de comparar y saltar según el resultado de comparar.

```
.L4:
        movsd   16(%rbp,%rax), %xmm4
        movsd   16(%rbx,%rax), %xmm5
        movhpd  24(%rbp,%rax), %xmm4
        movsd   32(%rbp,%rax), %xmm2
        movhpd  24(%rbx,%rax), %xmm5
        movapd  %xmm4, %xmm1
        movhpd  40(%rbp,%rax), %xmm2
        movsd   32(%rbx,%rax), %xmm4
        mulpd   %xmm3, %xmm1
        movapd  %xmm2, %xmm0
        movhpd  40(%rbx,%rax), %xmm4
        mulpd   %xmm3, %xmm0
        addpd   %xmm5, %xmm1
        movsd   0(%rbp,%rax), %xmm5
        addpd   %xmm4, %xmm0
        movhpd  8(%rbp,%rax), %xmm5
        movapd  %xmm5, %xmm2
        movlpd  %xmm1, 16(%rbx,%rax)
        movsd   (%rbx,%rax), %xmm5
        mulpd   %xmm3, %xmm2
        movhpd  %xmm1, 24(%rbx,%rax)
        movhpd  8(%rbx,%rax), %xmm5
        movlpd  %xmm0, 32(%rbx,%rax)
        movhpd  %xmm0, 40(%rbx,%rax)
        addpd   %xmm5, %xmm2
        movlpd  %xmm2, (%rbx,%rax)
        movhpd  %xmm2, 8(%rbx,%rax)
        addq    $48, %rax
        cmpq    $1215999984, %rax
        jne     .L4
        movsd   .LC2(%rip), %xmm0
        leaq    80(%rsp), %rsi
        movsd   1215999984(%rbp), %xmm1
        xorl    %edi, %edi
        mulsd   %xmm0, %xmm1
        addsd   1215999984(%rbx), %xmm1
        movsd   %xmm1, 1215999984(%rbx)
        movsd   1215999992(%rbp), %xmm1
        mulsd   %xmm0, %xmm1
        mulsd   1216000000(%rbp), %xmm0
        addsd   1215999992(%rbx), %xmm1
        addsd   1216000000(%rbx), %xmm0
        movsd   %xmm1, 1215999992(%rbx)
        movsd   %xmm0, 1216000000(%rbx)
```

daxpyOs.s: equivalente a las versiones -1 y -2

```
.L4:
                movsd           0(%rbp,%rax), %xmm1
                mulsd           %xmm0, %xmm1
                addsd           (%rbx,%rax), %xmm1
                movsd           %xmm1, (%rbx,%rax)
                movsd           8(%rbp,%rax), %xmm1
                mulsd           %xmm0, %xmm1
                addsd           8(%rbx,%rax), %xmm1
                movsd           %xmm1, 8(%rbx,%rax)
                movsd           16(%rbp,%rax), %xmm1
                mulsd           %xmm0, %xmm1
                addsd           16(%rbx,%rax), %xmm1
                movsd           %xmm1, 16(%rbx,%rax)
                addq            $24, %rax
                cmpq            $1216000008, %rax
                jne             .L4
                leaq            80(%rsp), %rsi
                xorl            %edi, %edi
```