2º curso / 2º cuatr.

Grado Ing. Inform.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 4. Optimización de código

Estudiante (nombre y apellidos): Daniel Monjas Miguélez
Grupo de prácticas y profesor de prácticas: Miércoles, Mancia Anguita
Fecha de entrega: 31 de mayo de 2020
Fecha evaluación en clase: 31 de mayo de 2020

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): *(respuesta)*

model name : Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz

Sistema operativo utilizado: *(respuesta)*

Ubuntu 18.04.4 LTS

Versión de gcc utilizada: *(respuesta)*

gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)

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

```
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS] 2020-05-20 miércoles
$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) i5-8300H CPU @ 2.30GHz
Stepping:            10
CPU MHz:             3437.051
CPU max MHz:         4000,0000
CPU min MHz:         800,0000
BogoMIPS:            4599.93
Virtualisation:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            8192K
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 dts
acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts r
ep_good nopl xtopology nonstop_tsc cpuid aperfmperf 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 xsave avx f16c rdr
and lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi fle
xpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflus
hopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_ep
p 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 (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.

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

#define MAX 10000

double a[MAX][MAX],b[MAX][MAX], res[MAX][MAX];

int main(int argc, char * argv[]){

    if(argc < 2){
        printf("Modo ejercución: <programa> <filas/columnas> \n");
        exit(-1);
    }

    int dimension=atoi(argv[1]);
    struct timespec cgt1,cgt2;
    double ncgt;

    srand((unsigned int) getpid());

    for(int i=0; i < dimension; i++){
        for(int j=0; j < dimension; j++){
            a[i][j]=rand()%10;
            b[i][j]=rand()%10;
        }
    }

    clock_gettime(CLOCK_REALTIME,&cgt1);

    for(int i=0; i < dimension; i++){
        for(int j=0; j < dimension; j++){
            for(int k=0; k < dimension; k++){
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    clock_gettime(CLOCK_REALTIME,&cgt2);

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

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: En la primera modificación antes de realizar el producto de las matrices se realiza la traspuesta de la segunda matriz, y se modifica el código del producto de matrices, para así mantener el resultado correcto del producto de matrices al multiplicar por filas en vez de fila por columna. En mi código para la prueba de funcionamiento incluyo que si el tamaño de la matriz es menor o igual que 5x5 se vuelva a trasponer b, y tras esto se muestran a, b y el resultado del producto.

Modificación b) –explicación-: Para la segunda modificación del código se utiliza desenrollado de bucles, para ello lo que hacemos es ejecutar 5 iteraciones del bucle (por como la he programado) por cada iteración que se produciría con el código secuencial. Resaltar que se hacen las comprobaciones pertinentes para no salirse de la memoria reservada. Por contra parte esto implica un aumento en el tamaño de código. En mi código para la prueba de funcionamiento incluyo que si la matriz es de tamaño menor o igual que 5x5 se muestren las matrices a, b y resultado.

1.1. CÓDIGOS FUENTE MODIFICACIONES

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

```
clock_gettime(CLOCK_REALTIME,&cgt1);

//Se traspone la matriz
for(int i=0; i < dimension; i++){
    for(int j=i; j < dimension; j++){
        if(i!=j){
            acumulador = b[i][j];
            b[i][j] = b[j][i];
            b[j][i] = acumulador;
        }
    }
}
//Se realize el producto de las matrices por filas
for(int i=0; i < dimension; i++){
    for(int j=0; j < dimension; j++){
        for(int k=0; k < dimension; k++){
            res[i][j] += a[i][k] * b[j][k];
        }
    }
}
clock_gettime(CLOCK_REALTIME,&cgt2);

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


//Prueba de funcionamiento
if(dimension <= 5){
    for(int i=0; i < dimension; i++){
        for(int j=0; j < dimension; j++){
            printf("a[%d][%d]=%f ",i,j,a[i][j]);
        }

        printf("\n");
    }
    //Se vuelve a trasponer para que cuando se muestre entera se muestren los coeficientes correctos
    for(int i=0; i < dimension; i++){
        for(int j=i; j < dimension; j++){
            if(i!=j){
                acumulador = b[i][j];
                b[i][j] = b[j][i];
                b[j][i] = acumulador;
            }
        }
    }
```

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

```
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$gcc -O2 -o producto_secuencial_modificado_1 producto_secuencial_modificado_1.c
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$./producto_secuencial_modificado_1 3
a[0][0]=6.000000 a[0][1]=2.000000 a[0][2]=9.000000
a[1][0]=2.000000 a[1][1]=9.000000 a[1][2]=1.000000
a[2][0]=5.000000 a[2][1]=5.000000 a[2][2]=7.000000

b[0][0]=4.000000 b[0][1]=0.000000 b[0][2]=8.000000
b[1][0]=1.000000 b[1][1]=7.000000 b[1][2]=9.000000
b[2][0]=5.000000 b[2][1]=8.000000 b[2][2]=4.000000

res[0][0]=71.000000 res[0][1]=86.000000 res[0][2]=102.000000
res[1][0]=22.000000 res[1][1]=71.000000 res[1][2]=101.000000
res[2][0]=60.000000 res[2][1]=91.000000 res[2][2]=113.000000

Tiempo: 0.000030
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$./producto_secuencial_modificado_1 4
a[0][0]=8.000000 a[0][1]=0.000000 a[0][2]=9.000000 a[0][3]=7.000000
a[1][0]=8.000000 a[1][1]=5.000000 a[1][2]=3.000000 a[1][3]=2.000000
a[2][0]=4.000000 a[2][1]=8.000000 a[2][2]=5.000000 a[2][3]=2.000000
a[3][0]=9.000000 a[3][1]=5.000000 a[3][2]=1.000000 a[3][3]=0.000000

b[0][0]=3.000000 b[0][1]=6.000000 b[0][2]=1.000000 b[0][3]=4.000000
b[1][0]=4.000000 b[1][1]=0.000000 b[1][2]=9.000000 b[1][3]=0.000000
b[2][0]=0.000000 b[2][1]=6.000000 b[2][2]=7.000000 b[2][3]=9.000000
b[3][0]=3.000000 b[3][1]=7.000000 b[3][2]=2.000000 b[3][3]=9.000000

res[0][0]=45.000000 res[0][1]=151.000000 res[0][2]=85.000000 res[0][3]=176.000000
res[1][0]=50.000000 res[1][1]=80.000000 res[1][2]=78.000000 res[1][3]=77.000000
res[2][0]=50.000000 res[2][1]=68.000000 res[2][2]=115.000000 res[2][3]=79.000000
res[3][0]=47.000000 res[3][1]=60.000000 res[3][2]=61.000000 res[3][3]=45.000000

Tiempo: 0.000039
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$./producto_secuencial_modificado_1 1500
a[0][0]=9.000000 -- a[1499][1499]=7.000000
b[0][0]=7.000000 -- b[1499][1499]=8.000000
res[0][0]=30157.000000 -- res[1499][1499]=30373.000000
Tiempo: 3.793935
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$
```

b) …

```
int divisor;

srand((unsigned int) getpid());

for(int i=0; i < dimension; i++){
    for(int j=0; j < dimension; j++){
        a[i][j]=rand()%10;
        b[i][j]=rand()%10;
    }
}

clock_gettime(CLOCK_REALTIME,&cgt1);
//Desenrollado de bucle
for(int i=0; i < dimension; i++){
    for(int j=0; j < dimension; j++){
        for(int k=0; k < dimension; k+=5){
            res[i][j] += a[i][k] * b[k][j];

            if(k+1 < dimension)
                res[i][j] += a[i][k+1] * b[k+1][j];

            if(k+2 < dimension)
                res[i][j] += a[i][k+2] * b[k+2][j];

            if(k+3 < dimension)
                res[i][j] += a[i][k+4] * b[k+3][j];

            if(k+4 < dimension)
                res[i][j] += a[i][k+4] * b[k+3][j];
        }
    }
}
clock_gettime(CLOCK_REALTIME,&cgt2);

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

//Prueba de funcionamiento
if(dimension <= 5){
    for(int i=0; i < dimension; i++){
        for(int j=0; j < dimension; j++){
            printf("a[%d][%d]=%f ",i,j,a[i][j]);
        }

        printf("\n");
```

```
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$gcc -O2 -o producto_secuencial_modificado_2 producto_secuencial_modificado_2.c
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$./producto_secuencial_modificado_2 3
a[0][0]=4.000000 a[0][1]=1.000000 a[0][2]=6.000000
a[1][0]=4.000000 a[1][1]=2.000000 a[1][2]=7.000000
a[2][0]=0.000000 a[2][1]=3.000000 a[2][2]=9.000000

b[0][0]=0.000000 b[0][1]=0.000000 b[0][2]=0.000000
b[1][0]=7.000000 b[1][1]=9.000000 b[1][2]=7.000000
b[2][0]=4.000000 b[2][1]=2.000000 b[2][2]=7.000000

res[0][0]=31.000000 res[0][1]=21.000000 res[0][2]=49.000000
res[1][0]=42.000000 res[1][1]=32.000000 res[1][2]=63.000000
res[2][0]=57.000000 res[2][1]=45.000000 res[2][2]=84.000000

Tiempo: 0.000029
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$./producto_secuencial_modificado_2 4
a[0][0]=2.000000 a[0][1]=9.000000 a[0][2]=1.000000 a[0][3]=6.000000
a[1][0]=8.000000 a[1][1]=3.000000 a[1][2]=7.000000 a[1][3]=4.000000
a[2][0]=9.000000 a[2][1]=2.000000 a[2][2]=3.000000 a[2][3]=4.000000
a[3][0]=6.000000 a[3][1]=4.000000 a[3][2]=2.000000 a[3][3]=4.000000

b[0][0]=7.000000 b[0][1]=9.000000 b[0][2]=7.000000 b[0][3]=2.000000
b[1][0]=0.000000 b[1][1]=3.000000 b[1][2]=6.000000 b[1][3]=0.000000
b[2][0]=2.000000 b[2][1]=1.000000 b[2][2]=0.000000 b[2][3]=2.000000
b[3][0]=4.000000 b[3][1]=3.000000 b[3][2]=6.000000 b[3][3]=4.000000

res[0][0]=16.000000 res[0][1]=46.000000 res[0][2]=68.000000 res[0][3]=6.000000
res[1][0]=70.000000 res[1][1]=88.000000 res[1][2]=74.000000 res[1][3]=30.000000
res[2][0]=69.000000 res[2][1]=90.000000 res[2][2]=75.000000 res[2][3]=24.000000
res[3][0]=46.000000 res[3][1]=68.000000 res[3][2]=66.000000 res[3][3]=16.000000

Tiempo: 0.000037
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$./producto_secuencial_modificado_2 1500
a[0][0]=6.000000 -- a[1499][1499]=7.000000
b[0][0]=5.000000 -- b[1499][1499]=7.000000
res[0][0]=30275.000000 -- res[1499][1499]=29659.000000
Tiempo: 5.315578
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$
```

En la explicación de las modificaciones se indica porque para tamaños 3 y 4 se muestran las tres matrices enteras.

1.1. TIEMPOS:

Modificación	**Breve descripción de las modificaciones**	**-O2**
Sin modificar	Se realiza el producto de las matrices por medio de tres bucles anidados.	6.606398 (Tam=1400)
Modificación a)	Se traspone la matriz b y el producto de matrices se hace por filas	3.188413 (Tam =1400)
Modificación b)	En cada iteración del tercer bucle anidado se ejecutan 5 iteraciones de las que se harían en el código secuencial evitando así saltos (con las correspondientes comprobaciones para no salirse de la memoria reservada)	3.861205 (Tam=1400)
...		

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

Como se puede observar para un mismo tamaño de problema los tiempos obtenidos por las modificaciones son significativamente mejores respecto al obtenido con el código secuencial.

En la primera modificación se debe a que al trasponer la matriz y realizar el producto por filas en lugar de por columnas lo que se busca es aprovechar como almacenan los datos los compiladores de intel, que almacenan por filas para así utilizar la localidad.

Por otro lado, en la segunda modificación se aprovecha que al realizar cinco iteraciones secuenciales en una se reducirán los saltos del bucle, reduciendo así la penalización que se puede obtener por estos, además de poder ejecutar en una misma iteración tareas independientes. Por contraparte la segunda modificación aumenta el tamaño del código.

B) CÓDIGO FIGURA 1:

CAPTURA CÓDIGO FUENTE: `figura1-original.c`

daniel > Escritorio > Daniel > AC > PRACTICAS > bp4 > ejer1 > C figura_1.c > main(int, char * [])

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

int R[40000];

int main(int argc, char *argv[]){
    int ii;
    int i=0;
    int x1, x2;
    struct timespec cgt1,cgt2;
    double ncgt;

    for (int i = 0; i < 5000; i++) {
      s[i].a = 1;
      s[i].b = 1;
    }

    clock_gettime(CLOCK_REALTIME,&cgt1);
    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;

    }

    clock_gettime(CLOCK_REALTIME,&cgt2);

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

    printf("Tiempo: %f\n", ncgt);
    printf("R[0]=%d -- R[%d]=%d\n",R[0], 39999, R[39999]);

    return 0;
}
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a) –explicación-: En la modificación 1 lo que he hecho es, aprovechando que los dos bucles dentro del bucle de 40000 iteraciones tienen el mismo número de iteraciones, y que las operaciones dentro de los mismos son completamente independientes, los he unido en uno solo. De forma que por cada iteración de este nuevo bucle se realiza una iteración de cada uno de los bucles anteriores.

Modificación b) –explicación-: He reutilizado la modificación 1 pero además le he añadido al bucle que ejecuta 5000 iteraciones y dos operaciones independientes, desenrollado de bucles, de forma que por cada iteración en el código modificado 2 se realizan 5 iteraciones del código modificado 1 y 5 iteraciones de cada bucle por separado del código secuencial.

…

1.1. CÓDIGOS FUENTE MODIFICACIONES

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

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

int R[40000];

int main(int argc, char *argv[]){
    int ii;
    int i=0;
    int x1, x2;
    struct timespec cgt1,cgt2;
    double ncgt;

    for (i = 0; i < 5000; i++) {
      s.a[i] = 1;
      s.b[i] = 1;
    }

    clock_gettime(CLOCK_REALTIME,&cgt1);
    for (ii = 0; ii < 40000; ii++){
        x1=0; x2=0;

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

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

        else
            R[ii] = x2;

    }

    clock_gettime(CLOCK_REALTIME,&cgt2);

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

    printf("Tiempo: %f\n", ncgt);
    printf("R[0]=%d -- R[%d]=%d\n",R[0], 39999, R[39999]);
```

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

```
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$gcc -O2 -o figura_1_mod_1 figura_1_mod_1.c
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$./figura_1_mod_1
Tiempo: 0.125720
R[0]=10000 -- R[39999]=-199980000
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$
```

El resultado es correcto, ya que para ii=0, se tiene que x1=5000*2*1+0 *5000 y x2=5000*3*1-0*5000, ya que se

realiza 2*1 y 3*1 en todas las iteraciones, luego x1 es menor que x2 y por tanto R[0]=x1=10000. Por otro lado en la última iteración se tiene x1=5000*1*2+39999*5000 y x2=5000*1*3-39999*5000, luego claramente x2 < x1 y por tanto, R[39999]=x2=-199980000, que es lo que se ha obtenido en la salida de la ejecución.

b) …

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

int R[40000];

int main(int argc, char *argv[]){
    int ii;
    int i=0;
    int x1, x2;
    struct timespec cgt1,cgt2;
    double ncgt;

    for (int i = 0; i < 5000; i++) {
      s[i].a = 1;
      s[i].b = 1;
    }

    clock_gettime(CLOCK_REALTIME,&cgt1);
    for (ii = 0; ii < 40000; ii++){
        x1=0; x2=0;
        //Desenrollado de bucle
        for(i = 0; i < 5000; i+=5){
            x1 += 2*s[i].a+ii;
            x1 += 2*s[i+1].a+ii;
            x1 += 2*s[i+2].a+ii;
            x1 += 2*s[i+3].a+ii;
            x1 += 2*s[i+4].a+ii;

            x2 += 3*s[i].b-ii;
            x2 += 3*s[i+1].b-ii;
            x2 += 3*s[i+2].b-ii;
            x2 += 3*s[i+3].b-ii;
            x2 += 3*s[i+4].b-ii;
        }

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

        else
            R[ii] = x2;

    }
```

```
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$gcc -O2 -o figura_1_mod_2 figura_1_mod_2.c
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$./figura_1_mod_2
Tiempo: 0.099705
R[0]=10000 -- R[39999]=-199980000
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer1] 2020-05-27 mi
ércoles
$
```

De la explicación anterior se obtiene que el resultado es correcto.

1.1. TIEMPOS:

Modificación	**Breve descripción de las modificaciones**	**-O2**
Sin modificar	Se ejecuta secuencialmente un bucle de 40000 iteraciones que contiene dos bucles independientes de 5000 iteraciones seguido de un if-else	0.185035
Modificación a)	Se ejecuta un bucle de 40000 iteraciones que contiene un bucle de 5000 iteraciones que realiza las tareas independientes antes divididas en dos bucles de 5000 iteraciones	0.125720
Modificación b)	Se utiliza la misma estructura que en el caso anterior, con la diferencia que el bucle de 5000 iteraciones en cada iteración realiza 5 iteraciones del bucle de la modificación 1 y 5 iteraciones de cada bucle independiente del código secuencial.	0.099705
...		

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

Atendiendo a los tiempos ambas modificaciones mejoran el tiempo obtenido por el código secuencial.

En la modificación 1, se reduce el tiempo de ejecución, ya que al juntar dos bucles con tareas independientes a 1 sólo bucle que realice las tareas de los dos bucles separados se reducen en buena medida la cantidad de saltos condicionales, además de las penalización que un salto condicional conlleva.

En la modificación 2, a parte de las ventajas que se obtienen gracias a la modificación 1 se le añade el desenrollado de bucle, dando lugar a que en cada iteración del bucle de 5000 iteraciones se ejecuten cinco iteraciones del bucle de 5000 iteraciones de la modificación 1 reduciendo aún más la cantidad de saltos condicionales y sus consecuencias. Por otro lado conlleva un código más largo.

A) MULTIPLICACIÓN DE MATRICES:

1.2 ANÁLISIS CÓDIGO ENSAMBLADOR.

CAPTURA CÓDIGO ENSAMBLADOR: `pmm-secuencial.s`

pmm-secuencial.s	`pmm-secuencial-modificado_a.s`	`pmm-secuencial-modificado_b.s`
Call clock_gettime@PLT **imulq $80000, 24(%rsp), %r10** **leaq 80000(%r14), %rax** **leaq 0(,%r15,8), %r9** **movq %r14, %rcx** **movq %r14, 8(%rsp)** **movq %r13, %rdi** **addq %rax, %r10** **.p2align 4,,10** **.p2align 3** **.L21:**	**Call clock_gettime@PLT** **leaq 80000+b(%rip), %r9** **movl 24(%rsp), %r8d** **movl 48(%rsp), %r10d** **movl $1, %edi** **leaq -80000(%r9), %rax** **movq %r9, 16(%rsp)** **movq %rax, %rcx** **movq %rax, 56(%rsp)** **movq %rax, %r14** **.p2align 4,,10**	**call clock_gettime@PLT** **imulq $80000, 24(%rsp), %r11** **movq 32(%rsp), %rbp** **leaq 80000(%r12), %rax** **movq %r13, %r10** **movq %r13, 8(%rsp)** **movq %r12, %r9** **salq $3, %rbp** **addq %rax, %r11** **.p2align 4,,10** **.p2align 3**

```
xorl %esi, %esi
.p2align 4,,10
.p2align 3
.L10:
movsd (%rdi,%rsi), %xmm1
leaq b(%rip), %r12
leaq 0(%rbp,%rsi), %rdx
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L7:
movsd(%rcx,%rax,8), %xmm0
addq $1, %rax

addq $80000, %rdx

mulsd -80000(%rdx), %xmm0
cmpq %r15, %rax
addsd %xmm0, %xmm1
jne .L7
movsd %xmm1, (%rdi,%rsi)
addq $8, %rsi
cmpq %r9, %rsi
jne .L10
addq $80000, %rcx
addq $80000, %rdi
cmpq %r10, %rcx
jne .L21
leaq 64(%rsp), %rsi
xorl %edi, %edi
xorl %ebx, %ebx
leaq .LC1(%rip), %r15
call clock_gettime@PLT
```

```
.p2align 3
.L9:

cmpl %r15d, %r8d
jle .L11
movl %r10d, %esi
movq %r9, %rdx
movq %rdi, %rax
subl %edi, %esi
addq %rdi, %rsi
jmp .L12
.p2align 4,,10
.p2align 3
.L8:
cmpl %eax, %r15d
je .L7
movsd (%rcx,%rax,8), %xmm0
movsd (%rdx), %xmm1
movsd %xmm1, (%rcx,%rax,8)
movsd %xmm0, (%rdx)
.L7
addq $1, %rax
addq $80000, %rdx
.L12:
cmpq %rsi, %rax
jne .L8
.L11:
addl $1, %r15d
addq $1, %rdi
addq $80008, %r9
addq $80000, %rcx
cmpl %r8d, %r15d
jne .L9
imulq $80000, 32(%rsp), %rax
leaq a(%rip), %r13
leaq b(%rip), %rbx
movq 40(%rsp), %rsi
leaq res(%rip), %r12
leaq 80000(%r13), %r11
movq %r13, %rbp
movq %r13, %rcx
salq $3, %rsi
movq %r12, %r10
leaq 80000(%rbx,%rax), %r9
addq %rax, %r11
.p2align 4,,10
.p2align 3

.L10:
leaq b(%rip), %rdx
movq %r10, %rdi
.p2align 4,,10
.p2align 3
.L16
movsd (%rdi), %xmm1
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L13:
movsd (%rcx,%rax), %xmm0
mulsd (%rdx,%rax), %xmm0
addq $8, %rax
cmpq %rsi, %rax
addsd %xmm0, %xmm1
jne .L13
```

```
.L24:
xorl %edi, %edi
.p2align 4,,10
.p2align 3
.L14:
movsd (%r9,%rdi), %xmm0
leaq b(%rip), %r14
leaq (%rbx,%rdi), %rcx
movq %r10, %rdx
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L11:
movsd (%rdx), %xmm1
leal 1(%rax), %esi
mulsd (%rcx), %xmm1
cmpl %r15d, %esi
addsd %xmm1, %xmm0
jge .L7
movsd 8(%rdx), %xmm1
mulsd 80000(%rcx), %xmm1
addsd %xmm1, %xmm0
.L7:
leal 2(%rax), %esi
cmpl %esi, %r15d

jle .L8
movsd 16(%rdx), %xmm1
mulsd 160000(%rcx), %xmm1
addsd %xmm1, %xmm0
.L8:
leal 3(%rax), %esi
cmpl %esi, %r15d

jle.L9
movsd 32(%rdx), %xmm1
mulsd 240000(%rcx), %xmm1
addsd %xmm1, %xmm0
.L9:
leal 4(%rax), %esi
cmpl %esi, %r15d

jle .L10
movsd 32(%rdx), %xmm1
mulsd 240000(%rcx), %xmm1
addsd %xmm1, %xmm0

.L10:
addl $5, %eax
addq $40, %rdx
addq $400000, %rcx
cmpl %r15d, %eax
jl .L11
movsd %xmm0, (%r9,%rdi)
addq $8, %rdi
cmpq %rbp, %rdi
jne .L14
addq $80000, %r9
addq $80000, %r10
cmpq %r11, %r9
jne .L24
leaq 64(%rsp), %rsi
xorl %edi, %edi
xorl %ebp, %ebp
call clock_gettime@PLT
```

	addq $80000, %rdx
	movsd %xmm1, (%rdi)
	addq $8, %rdi
	cmpq %r9, %rdx
	jne .L16
	addq $80000, %rcx
	addq $80000, %r10
	cmpq %r11, %rcx
	jne .L10
	leaq 80(%rsp), %rsi
	xorl %edi, %edi
	xorl %ebx, %ebx
	call clock_gettime@PLT	

El color rojo indican las operaciones de suma y producto para cualquier iteración en el código secuencial y la modificación 1, y para i en la modificación 2. El código verde en la modificación 2 indica el código ensamblador para la trasposición de la matriz. El código morado indican las operaciones de multiplicación y escritura para i+1,i+2,…,i+4, pues recordemos que se usa desenrollado de bucle. Todos estos códigos son los que en c están contenidos entre los clock_gettime. Se aprecia a simple vista que tanto la modificación 1 como la 2 tienen más código ensamblador que el código secuencial. También se aprecia que se realizan operaciones de suma con los registros %xmm0 y %xmm1 que son registros vectoriales. Mi ordenador en concreto usa instrucciones sd en lugar de pd para operaciones con elementos de los vectores.

B) FIGURA 1:
1.2 ANÁLISIS CÓDIGO ENSAMBLADOR.

figura1-original.s	figura1_modificado_a.s	figura1_modificado_b.s

Call clock_gettime@PLT
leaq 40004(%rbp), %r8
leaq R(%rip), %r10
xorl %r9d, %r9d
.p2align 4,,10
.p2align 3
.L3:
movl %r9d, %edi
movq %rbp, %rax
xorl %ecx, %ecx
.p2align 4,,10
.p2align 3
.L4:
movl (%rax), %edx
addq $8, %rax
leal (%rdi,%rdx,2), %edx
addl %edx, %ecx
cmpq %rbx, %rax
jne .L4
leaq 4+s(%rip), %rax
xorl %esi, %esi
.p2align 4,,10
.p2align 3
.L5:
movl (%rax), %edx
addq $8, %rax
leal (%rdx,%rdx,2), %edx
subl %edi, %edx
addl %edx, %esi
cmpq %r8, %rax
jne.L5
cmpl %ecx, %esi
jle .L6
movl %ecx,(%r10,%r9,4)
	Call clock_gettime@PLT
	leaq R(%rip), %r9
	xorl %r8d, %r8d
	.p2align 4,,10
	.p2align 3
	.L3:
	movl %r8d, %edi
	movq %rbp, %rax
	xorl %esi, %esi
	xorl %ecx, %ecx
	.p2align 4,,10
	.p2align 3
	.L4:
	movl (%rax), %edx
	addq $4, %rax
	leal (%rdi,%rdx,2), %edx
	addl %edx, %ecx
	movl 19996(%rax), %edx
	leal (%rdx,%rdx,2), %edx
	subl %edi, %edx
	addl %edx, %esi
	cmpq %rbx, %rax
	jne .L4
	cmpl %esi, %ecx
	jge .L5
	movl %ecx, (%r9,%r8,4)
	.L6:
	addq $1, %r8
	cmpq $40000, %r8
	jne .L3
	leaq 16(%rsp), %rsi
	xorl %edi, %edi
	call clock_gettime@PLT
	.L5:
		Call clock_gettime@PLT
		leaq R(%rip), %r9
		xorl %r8d, %r8d
		.p2align 4,,10
		.p2align 3
		.L3:
		movl %r8d, %edx
		movq %rbp, %rax
		xorl %ecx, %ecx
		xorl %edi, %edi
		.p2align 4,,10
		.p2align 3
		.L4:
		movl (%rax), %esi
		addq $40, %rax
		leal (%rdx,%rsi,2), %esi
		addl %esi, %edi
		movl -32(%rax), %esi
		leal (%rdx,%rsi,2), %esi
		addl %esi, %edi
		movl -24(%rax), %esi
		leal (%rdx,%rsi,2), %esi
		addl %edi, %esi
		movl -16(%rax), %edi
		leal (%rdx,%rdi,2), %edi
		addl %edi, %esi
		movl -8(%rax), %edi
		leal (%rdx,%rdi,2), %edi
		addl %esi, %edi
		movl -36(%rax), %esi
		leal (%rsi,%rsi,2), %esi
		subl %edx, %esi
		addl %esi, %ecx
		movl -28(%rax), %esi

```
.L7:
addq $1, %r9
cmpq $40000, %r9
jne .L3
leaq 16(%rsp), %rsi
xorl %edi, %edi
call clock_gettime@PLT
//Codigo entre L7 y L6
.L6:
.cfi_restore_state
movl %esi, (%r10,%r9,4)
jmp .L7
```

```
.cfi_restore_state
movl %esi, (%r9,%r8,4)
jmp .L6
```

```
leal (%rsi,%rsi,2), %esi
subl %edx, %esi
addl %esi, %ecx
movl -20(%rax), %esi
leal (%rsi,%rsi,2), %esi
subl %edx, %esi
addl %ecx, %esi
movl -12(%rax), %ecx
leal (%rcx,%rcx,2), %ecx
subl %edx, %ecx
addl %ecx, %esi
movl -4(%rax), %ecx
leal (%rcx,%rcx,2), %ecx
subl %edx, %ecx
addl %esi, %ecx
cmpq %rax, %rbx
jne .L4
cmpl %ecx, %edi
jge .L5
movl %edi, (%r9,%r8,4)
.L6:
addq $1, %r8
cmpq $40000, %r8
jne .L3
leaq 16(%rsp), %rsi
xorl %edi, %edi
call clock_gettime@PLT
.L5:
.cfi_restore_state
movl %ecx, (%r9,%r8,4)
jmp .L6
```

Para este segundo análisis de código ensamblador he utilizado, morado para el if-else que se realiza al final para asignar valor en cada iteración del primer bucle(menos profundo). En la opción sin modificaciones los colores rojo y azul indican los bucles de 5000 iteraciones, y el color verde indican el bucle de 40000 iteraciones. Obviamente lo rojo y lo azul es también verde por ser los dos bucles que se encuentran dentro del bucle de 40000 iteraciones. Para las modificaciones 1 y 2 he utilizado el color rojo para el nuevo bucle que une los bucles rojo y azul del codigo sin modificaciones y he seguido utilizando el verde para el bucle de 40000 iteraciones. Al igual que en el caso anterior lo rojo es también verde. Por último hay que destacar que la parte roja de la modificación 2 es mucho más grande que en la modificación 1, esto se debe a que la modificación 2 utiliza desenrollado de bucle, luego las operaciones independientes que se realizan en cada iteración equivalen a 5 iteraciones de la modificación 1 y eso se refleja en un aumento en el código ensamblador. En este caso destacar que solo se utilizan registros vectoriales e instrucciones con el sufijo sd en el código sin modificaciones.

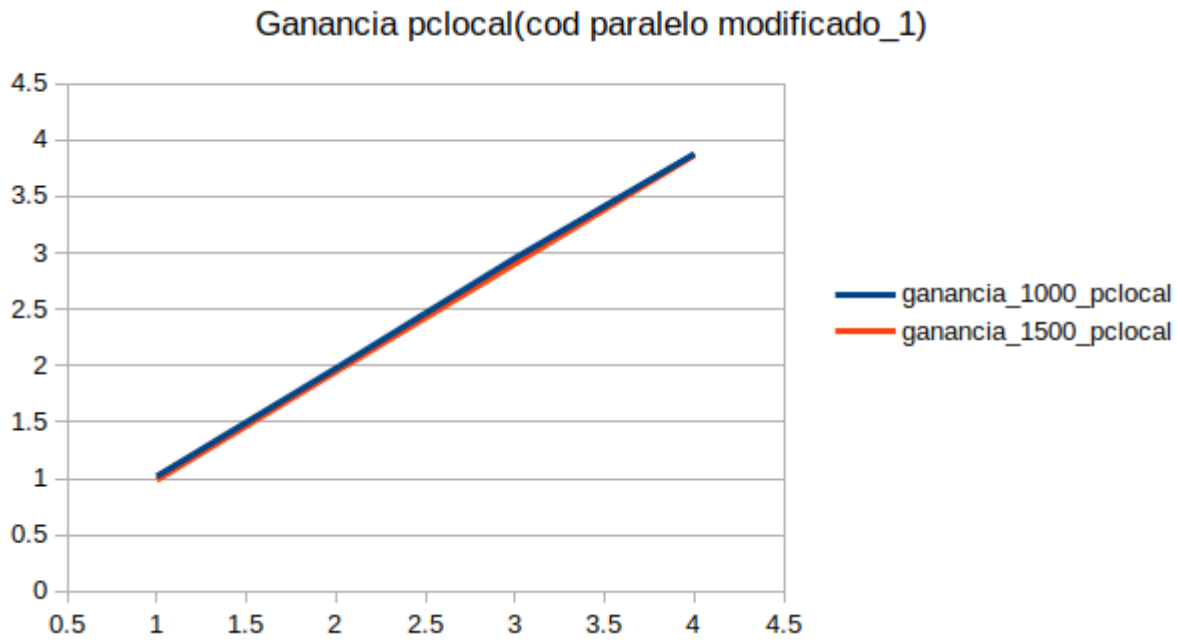
Ejercicio Extra (Estudio escalabilidad entre codigo_modificado_ 1 y su correspondiente código paralelo):

Comparacion con codigo paralelo pmm_mod_1					
	pclocal				
Tamaño	1	2	3	4	Secuencial
1000	1.120581	0.57809	0.386717	0.294767	1.143126
1500	4.014242	2.032833	1.362333	1.023704	3.960044

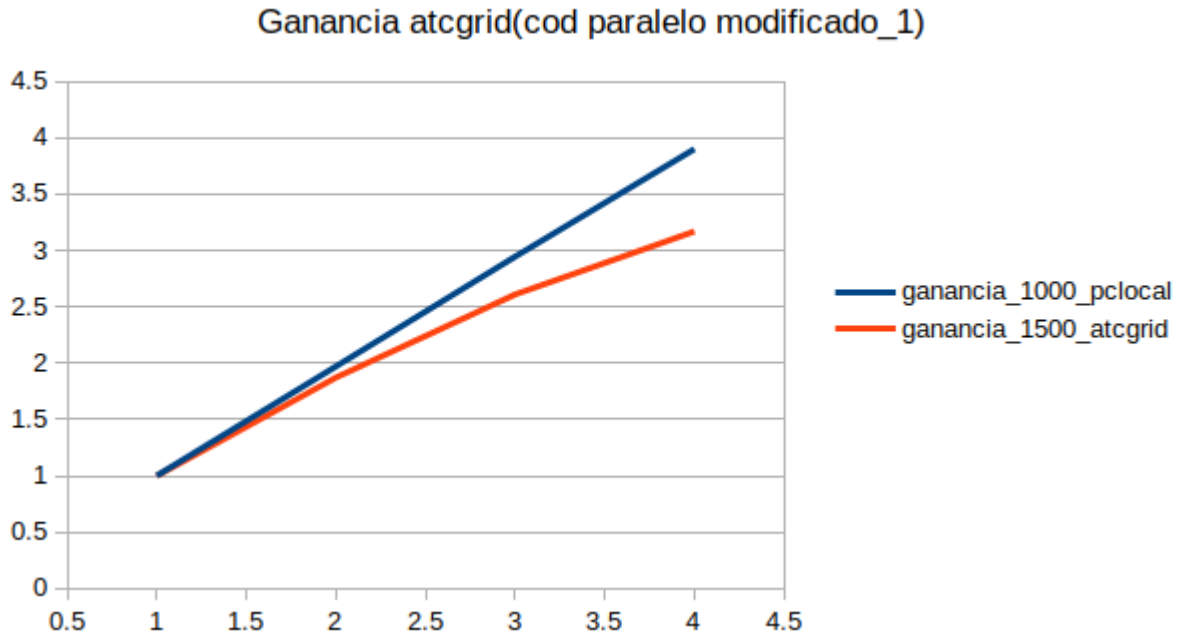
Ganancia pclocal				
Tamaño	1	2	3	4
1000	1.020119028	1.977418741	2.955975558	3.878066405
1500	0.986498572	1.948041969	2.906810596	3.868348663

	atcgrid				
Tamaño	1	2	3	4	Secuencial
1000	1.325541	0.673181	0.450424	0.340252	1.327398
1500	4.665487	2.490451	1.786258	1.47126	4.664062

Ganancia atcgrid				
Tamaño	1	2	3	4
1000	1.001400937	1.97182927	2.946996608	3.901220272
1500	0.999694566	1.872778063	2.611079698	3.170114052



Se puede observar que la ganancia en mi portátil para los distintos números de núcleos es prácticamente lineal y se aproxima bastante a la ganancia máxima teórica, pues para 1 core físico se obtiene ganancia 1, para 2 cores físicos ganancia cercana a 2, y así sucesivamente hasta 4 cores físicos. Esto se verifica para ambos tamaños utilizados 1000 y 1500.



Al igual que en el caso anterior para tamaño 1000 la ganancia es lineal y la ganancia es cercana a la máxima teórica. Por otro lado para tamaño 1500 se ve como según aumentan los cores la ganancia sigue mejorando pero no al mismo ritmo, es decir, la ganancia ya no es lineal y para números de cores superiores a 2 ya no se obtiene una ganancia tan cercana a la máxima esperada.

Ahora por otro lado estudiaremos la mejora del codigo modificado respecto al código secuencia original y veremos la ganancia que se obtiene a partir de dicha mejora. Utilizaremos como tamaños 1000 y 1500 como hemos hecho para el anterior estudio de escalabilidad.

Tamaño	Codigo_secuencial_original	Codigo_secuencial_modificado
1000	1.911227	1.193821
1500	10.576989	4.092271

Tamaño	Ganancia
1000	1.600932636
1500	2.584625749

Como se puede observar la modificación realizada sobre el codigo secuencial obtiene una ganancia mayor que uno por lo que mejora el tiempo, es más, cuanto mayor sea el tamaño del problema introducido mayor será la ganancia obtenida.

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

unsigned int N;
double *x, *y, a;

int main(int argc, char * argv[]){
    int i;

    if(argc < 3){
        printf("Modo ejecución: <programa> <dimensión> <constante>\n");
        exit(-1);
    }
    struct timespec cgt1,cgt2;
    double ncgt;

    N = atoi(argv[1]);
    a = atof(argv[2]);

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

    for(i = 0; i < N; i++){
        x[i] = 1;
        y[i] = 4;
    }

    clock_gettime(CLOCK_REALTIME,&cgt1);

    for(i = 0; i < N; i++){
        y[i] = a*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("Tiempo: %f\n",ncgt);

    if(N <= 5){
        for (i = 0; i < N; i++)
        {
            printf("r[%d] = %f ",i,y[i]);
        }

        printf("\n");
    }
```

Tiempos ejec.	**-O0**	**-Os**	**-O2**	**-O3**
	2.587367	1.365490	1.259693	1.204346

Para las ejecuciones he usado tamaño 1000000000 y constante 2.5.

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

```
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer2] 2020-05-30 sábado
$gcc -o daxpy daxpy.c
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer2] 2020-05-30 sábado
$./daxpy 5 2.5
Tiempo: 0.000000
r[0] = 6.500000 r[1] = 6.500000 r[2] = 6.500000 r[3] = 6.500000 r[4] = 6.500000
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer2] 2020-05-30 sábado
$./daxpy 1000 2.5
Tiempo: 0.000009
r[0]=6.500000 -- r[999] = 6.500000
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer2] 2020-05-30 sábado
$./daxpy 1000000 2.5
Tiempo: 0.010790
r[0]=6.500000 -- r[999999] = 6.500000
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer2] 2020-05-30 sábado
$./daxpy 100000000 2.5
Tiempo: 0.345887
r[0]=6.500000 -- r[99999999] = 6.500000
[DanielMonjasMiguelez daniel@daniel-XPS-15-9570:~/Escritorio/Daniel/AC/PRACTICAS/bp4/ejer2] 2020-05-30 sábado
$
```

En mi código todos los valores de x es 1 y los valores de y son 4. Como la constante es 2.5, los resultados almacenados en el vector resultado son todos iguales y con valor 1*2.5+4=6.5, luego el código funciona correctamente.

COMENTARIOS QUE EXPLIQUEN LAS DIFERENCIAS EN ENSAMBLADOR:

En rojo se han marcado los códigos que realizan el producto de un elemento del vector x por un double y la suma del resultado con un elemento del vector y, así como el almacenamiento del resultado de esta suma. En azul he marcado la comprobación de que se verifica la condición del bucle for y el salto al principio del mismo. En verde he marcado la comprobación antes de entrar al bucle que la condición del bucle se verifica, en el daxpy0s.s se ha marcado en azul en vez de en verde, pues ese cmp y ese jbe hacen la comprobación tanto antes de entrar como iteración a itreación.

Las diferencias en ensamblador son de daxpyO0.s a daxpyOs.s se ve que el código rojo, es decir, el que ejecuta las operaciones aritméticas y el almacenamiento de resultados es menor en daxpyOs que en daxpyO0, lo que conlleva una reducción en el tiempo pues para tamaños grandes se requiere de la ejecución de muchos menos instrucciones. Además daxpyOs.s incluye un cmpl y un jbe (marcados en azul) que al entar en la primer iteración impiden que esta se ejecute si no se cumple la condición del bucle for.

De daxpyOs.s a daxpyO2.s no hay practicamente ninguna diferencia en el código rojo salvo que el incremento del indice en daxpyOs se hace con incq y en daxpyO2 se hace con addq. Por otra parte, en daxpyO2 la comprobación que se hace en la primer iteración del bucle se realiza con una orden test y permite que la comprobación del resto de iteraciones sea independiente, con lo que se evita que en la última iteración haya que ejecutar un salto de más como se hace en daxpyOs.s (se salta de jmp a cmp se ve que la condición no se cumple y se salta a .L18).

Finalmente el mayor cambio se ve en daxpyO3.s que tiene un código mucho más largo. En este el código rojo es prácticamente idéntico al del daxpyOs y daxpyO2, menos una parte marcada en naranja donde se sustituye el sufijo sd por pd. Al igual que en daxpyO2.s se tiene un test que comprueba si se cumple la condición del bucle para la primera iteración del mismo. Además daxpyO3.s incluye varias opciones, es decir, se realizan varias comprobaciones antes de empezar con las operaciones aritméticas y en función de los resultados de estas comprobaciones el bucle se puede ejecutar por medio de .L13 o .L11, donde este último tiene varias comprobaciones extra en las que en función de los resultados se realizan más o menos operaciones o se termina el bucle.

En todos los códigos se ve el uso de registros vectoriales xmm- y instrucciones con sufijo sd. Además en el daxpyO3.s también se aprecia instrucciones con sufijo pd.

En todos los códigos ensamblador utilizados en este ejercicio se aprecia el uso de registros vectoriales,

instrucciones con sufijo sd y además en el daxpyO3.s se observan instrucciones con sufijo pd.

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

```
Call clock_gettime@PLT
movl $0, -60(%rbp)
jmp .L5
.L6:
movq x(%rip), %rax
movl -60(%rbp), %edx
movslq %edx, %rdx
salq $3, %rdx
addq %rdx, %rax
movsd (%rax), %xmm1
movsd a(%rip), %xmm0
mulsd %xmm1, %xmm0
movq y(%rip), %rax
movl -60(%rbp), %edx
movslq %edx, %rdx
salq $3, %rdx
addq %rdx, %rax
movsd (%rax), %xmm1
movq y(%rip), %rax
movl -60(%rbp), %edx
movslq %edx, %rdx
salq $3, %rdx
addq %rdx, %rax
addsd %xmm1, %xmm0
movsd %xmm0, (%rax)
addl $1, -60(%rbp)
.L5:
movl -60(%rbp), %edx
movl N(%rip), %eax
cmpl %eax, %edx
jb .L6
leaq -32(%rbp), %rax
movq %rax, %rsi
movl $0, %edi
call clock_gettime@PLT
```

daxpyOs.s

```
call clock_gettime@PLT
movl N(%rip), %ecx
movq x(%rip), %rsi
xorl %eax, %eax
movq y(%rip), %rdx
.L5:
cmpl %eax, %ecx
jbe .L18
movsd (%rsi,%rax,8),%xmm0
mulsd a(%rip), %xmm0
addsd (%rdx,%rax,8),%xmm0
movsd %xmm0,(%rdx,%rax,8)
incq %rax
jmp .L5
.L18:
leaq 24(%rsp), %rsi
xorl %edi, %edi
xorl %ebx, %ebx
leaq .LC5(%rip), %rbp
call clock_gettime@PLT
```

daxpyO2.s

```
Call clock_gettime@PLT
movl N(%rip), %eax
testl %eax, %eax
je .L5
movq x(%rip), %rsi
movq y(%rip), %rdx
subl $1, %eax
leaq 8(,%rax,8), %rcx
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L6:
movsd (%rsi,%rax), %xmm0
mulsd a(%rip), %xmm0
addsd (%rdx,%rax), %xmm0
movsd %xmm0, (%rdx,%rax)
addq $8, %rax
cmpq %rax, %rcx
jne .L6
.L5:
leaq 16(%rsp), %rsi
xorl %edi, %edi
call clock_gettime@PLT
```

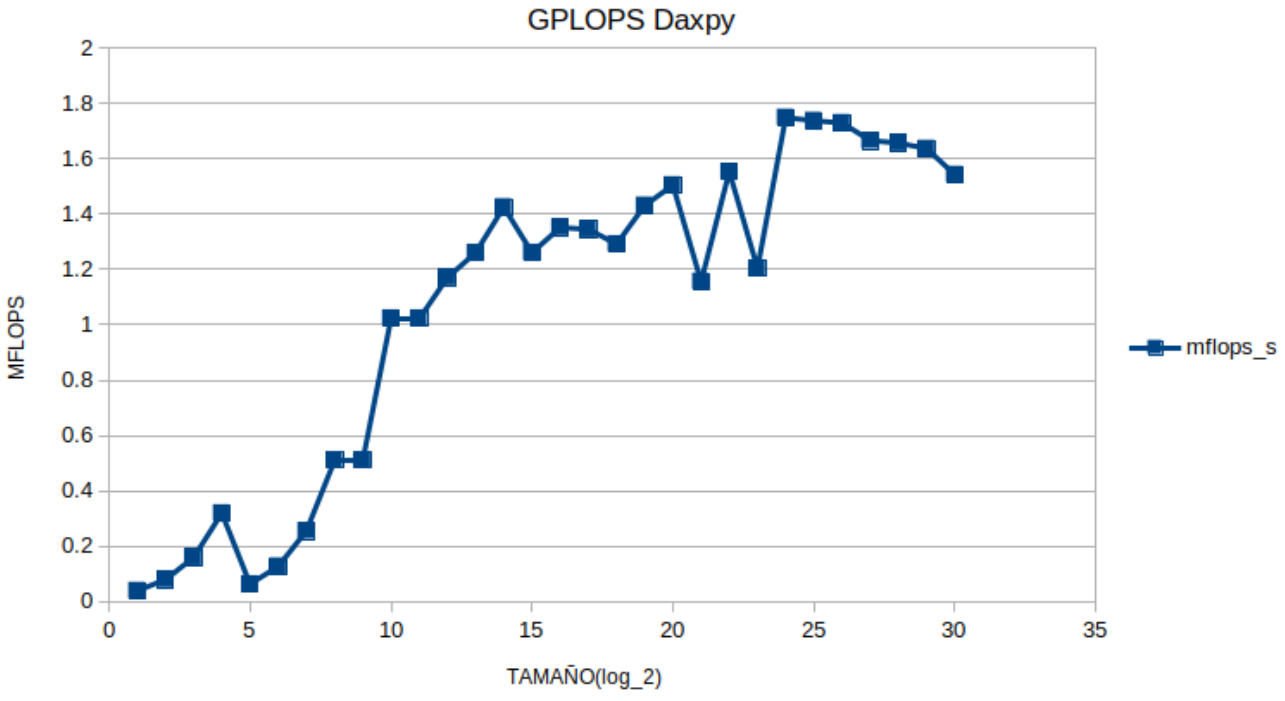
daxpyO3.s

```
call clock_gettime@PLT
movl N(%rip), %r9d
testl %r9d, %r9d
je .L8
movq y(%rip), %rsi
movq x(%rip), %r8
leaq 16(%rsi), %rax
leaq 16(%r8), %rdx
cmpq %rax, %r8
setnb %al
cmpq %rdx, %rsi
setnb %dl
orl %edx, %eax
cmpl $8, %r9d
seta %dl
testb %dl, %al
je .L9
movl %r9d, %eax
leaq (%rsi,%rax,8), %rdx
leaq a(%rip), %rax
cmpq %rax, %rdx
setbe %dl
addq $8, %rax
cmpq %rax, %rsi
setnb %al
orb %al, %dl
je .L9
movq %rsi, %rdx
xorl %edi, %edi
shrq $3, %rdx
andl $1, %edx
je .L10
movsd a(%rip), %xmm0
movl $1, %edi
mulsd (%r8), %xmm0
addsd (%rsi), %xmm0
movsd %xmm0, (%rsi)
.L10:
movsd a(%rip), %xmm1
movl %r9d, %ebx
xorl %eax, %eax
subl %edx, %ebx
movl %edx, %edx
xorl %ecx, %ecx
unpcklpd %xmm1, %xmm1
salq $3, %rdx
movl %ebx, %r11d
leaq (%r8,%rdx), %r10
addq %rsi, %rdx
shrl %r11d
.p2align 4,,10
.p2align 3
.L11:
movupd (%r10,%rax), %xmm0
addl $1, %ecx
mulpd %xmm1, %xmm0
addpd (%rdx,%rax), %xmm0
movaps %xmm0, (%rdx,%rax)
addq $16, %rax
```

			cmpl %ecx, %r11d
			ja .L11
			movl %ebx, %eax
			andl $-2, %eax
			addl %eax, %edi
			cmpl %eax, %ebx
			je .L8
			movslq %edi, %rdx
			addl $1, %edi
			movsd (%r8,%rdx,8), %xmm0
			leaq (%rsi,%rdx,8), %rax
			cmpl %r9d, %edi
			mulsd a(%rip), %xmm0
			addsd (%rax), %xmm0
			movsd %xmm0, (%rax)
			jnb .L8
			movslq %edi, %rdi
			movsd (%r8,%rdi,8), %xmm0
			leaq (%rsi,%rdi,8), %rax
			mulsd a(%rip), %xmm0
			addsd (%rax), %xmm0
			movsd %xmm0, (%rax)
			.L8:
			leaq 16(%rsp), %rsi
			xorl %edi, %edi
			call clock_gettime@PLT
			.L9:
			leal -1(%r9), %eax
			leaq 8(,%rax,8), %rdx
			xorl %eax, %eax
			.p2align 4,,10
			.p2align 3
			.L13:
			movsd a(%rip), %xmm0
			mulsd (%r8,%rax), %xmm0
			addsd (%rsi,%rax), %xmm0
			movsd %xmm0, (%rsi,%rax)
			addq $8, %rax
			cmpq %rdx, %rax
			jne .L13
			jmp .L8

Ejercicio extra:

Nº Ejecucion	Tamaño	Tiempo	GFLOPS
1	2	1E-07	0.04
2	4	1E-07	0.08
3	8	1E-07	0.16
4	16	1E-07	0.32
5	32	1E-06	0.064
6	64	1E-06	0.128
7	128	1E-06	0.256
8	256	1E-06	0.512
9	512	2E-06	0.512
10	1024	2E-06	1.024
11	2048	4E-06	1.024
12	4096	7E-06	1.170285714
13	8192	1.3E-05	1.260307692
14	16384	2.3E-05	1.424695652
15	32768	5.2E-05	1.260307692
16	65536	9.7E-05	1.351257732
17	131072	0.000195	1.344328205
18	262144	0.000406	1.291349754
19	524288	0.000733	1.430526603
20	1048576	0.001394	1.504413199
21	2097152	0.003628	1.156092613
22	4194304	0.005402	1.552870789
23	8388608	0.01393	1.204394544
24	16777216	0.019198	1.74780873
25	33554432	0.038644	1.736592071
26	67108864	0.077633	1.72887468
27	134217728	0.161242	1.664798601
28	268435456	0.324136	1.656313745
29	536870912	0.656426	1.635739328
30	1073741824	1.392773	1.541876277



Mirando a la tabla y el gráfico observamos que al ejecutar el programa daxpyO3 en mi ordenador para 30 tamaños distintos, se obtiene un Rmax de 1.747 GFLOPS, y el Nmax en el que se consigue Rmax es 1677716.

Ahora buscaremos N1/2 que es el tamaño en el que se obtiene R1/20=0.8735 aproximadamente. En la tabla vemos que este valor se encuentra entre los tamaños 1024 y 512, luego probamos distintos valores. Este se obtiene para el tamaño 873, que obtiene un tiempo de entre 0.000001 y 0.000002, más proximo de 0.000002, se tendrá que sus gigaflops estarán entre 873 y 970, tirando más hacia 873, pues el tiempo se aproxima más a 0.000002. Para el cálculo del Rpeak tomo IPC=1/5=0.2, ya que las instrucciones mulsd y mulpd tardan 5 ciclos en ejecutarse según el manual de Intel. Por otro lado mi ordenador utiliza registros xmm por tanto OPI=2. Por último la frecuencia pico de mi ordenador es de 4GHz (utilizando el boost) por consiguiente tengo que el numero de operaciones por segundo es Rpeak=0.2*2*4*10^9=1.6*10^9 ops/seg = 1.6 GFLOPS. Comparando esto con lo obtenido prácticamente se ve que el Rpeak es un poco más pequeño que el Rmax, lo que se puede deber que para el cálculo de Rpeak no hemos considerado el número de unidades punto flotante tiene el núcleo de procesamiento concreto que ha ejecutado el programa.