2º curso / 2º cuatr.

Grado Ing. Inform.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 4. Optimización de código

Estudiante (nombre y apellidos): Guillermo Sandoval Schmidt
Grupo de prácticas y profesor de prácticas: C3
Fecha de entrega: 22-mayo-2020

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

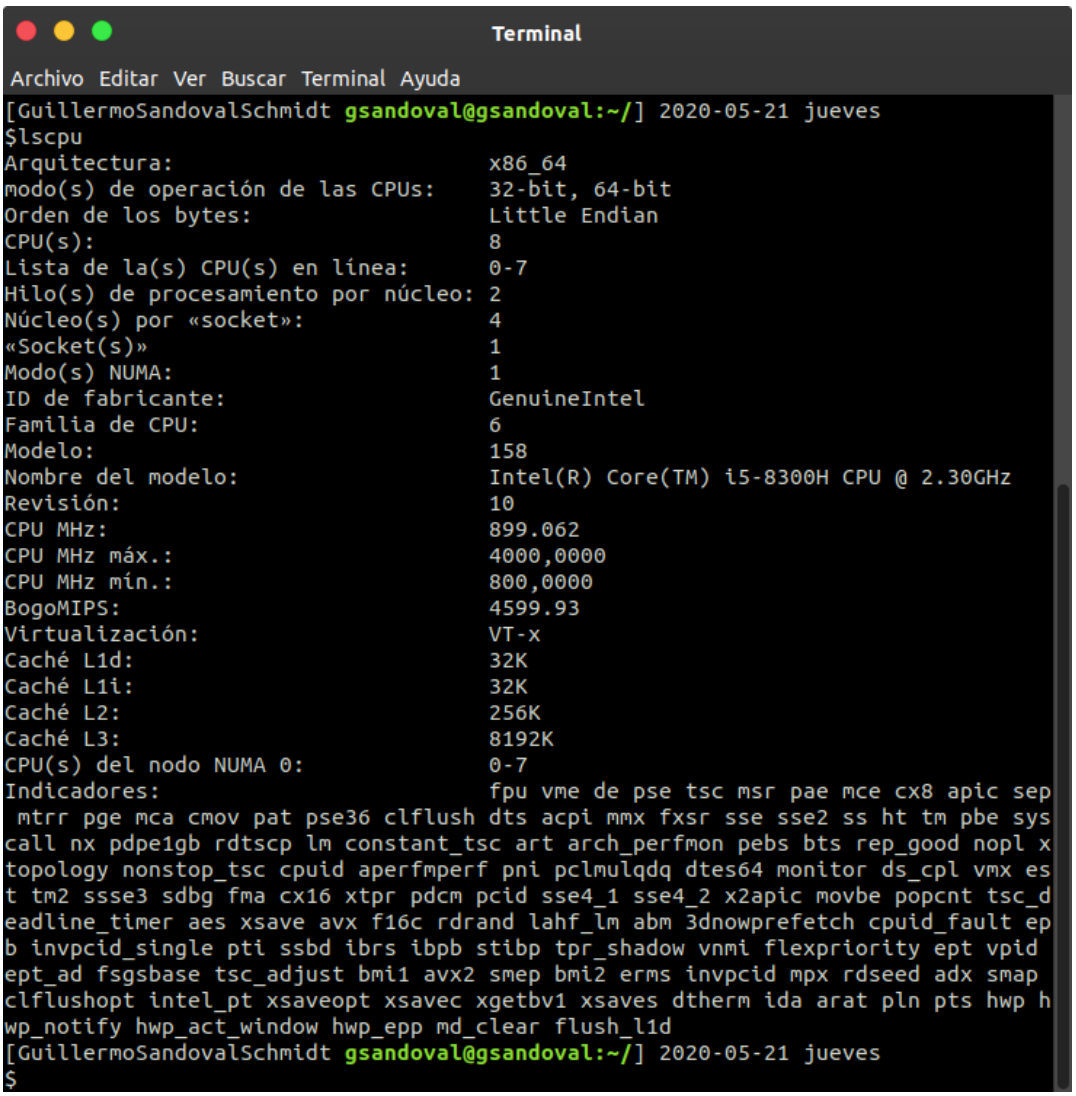
Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo):

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

Sistema operativo utilizado: *Ubuntu 18.04.3 LTS*

Versión de gcc utilizada: *gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0*

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



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.

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`

```
65    clock_gettime(CLOCK_REALTIME,&ini);
66
67    for(i=0; i<N; i++)
68      for(j=0; j<N; j++)
69        for(k=0; k<N; k++)
70          m3[i][j] += m1[i][k]*m2[k][j];
71
72    clock_gettime(CLOCK_REALTIME,&fin);
73
74    total=(double)(fin.tv_sec-ini.tv_sec) +
75       (double)((fin.tv_nsec-ini.tv_nsec)/(1.e+9));
```

1.1. MODIFICACIONES REALIZADAS:

Modificación a): Desenrollamos el bucle haciendo 4 iteraciones a la vez. Se optimiza el número de iteraciones del bucle, realizando N/4 iteraciones.

Modificación b): Recorremos la matriz por filas en vez de por columnas, optimizando el recorrido de la misma. Se optimiza al encontrarse los datos más cerca en memoria y evitamos fallos de acceso a caché.

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) y b) Captura de pmm-secuencial-a.c y pmm-secuencial-b.c

```
68
69    for(i=0; i<N; i++)
70      for(j=0; j<N; j++){
71        mult0 = mult1 = mult2 = mult3 = 0;
72          for(k=0; k<N; k+=4){
73            mult0 += m1[i][k]*m2[k][j];
74            mult1 += m1[i][k+1]*m2[k+1][j];
75            mult2 += m1[i][k+2]*m2[k+2][j];
76            mult3 += m1[i][k+3]*m2[k+3][j];
77          }
78          m3[i][j] = mult0 + mult1 + mult2 + mult3;
79        }
```

```
64
65    clock_gettime(CLOCK_REALTIME,&ini);
66
67    for(i=0; i<N; i++)
68      for(k=0; k<N; k++)
69        for(j=0; j<N; j++)
70          m3[i][j] += m1[i][k]*m2[k][j];
71
72    clock_gettime(CLOCK_REALTIME,&fin);
73
74    total=(double)(fin.tv_sec-ini.tv_sec) +
75       (double)((fin.tv_nsec-ini.tv_nsec)/(1.e+9));
```

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

```
Terminal
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1A/] 2020-05-21 jueves
$gcc -O2 pmm-secuencial.c -o ejecutable -lrt
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1A/] 2020-05-21 jueves
$gcc -O2 pmm-secuencial-a.c -o ejecutableA -lrt
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1A/] 2020-05-21 jueves
$gcc -O2 pmm-secuencial-b.c -o ejecutableB -lrt
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1A/] 2020-05-21 jueves
$./ejecutable 1000
Tiempo(seg.):2.370429591
 Tamaño matrices:1000
 m1[0][0]+m2[0][0]=m3[0][0](0.600000+3.285714=1971.428571) m1[999][999]+m2[999][999]=m3[999][999](0.600000+3.285714=1971.428571)
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1A/] 2020-05-21 jueves
$./ejecutableA 1000
Tiempo(seg.):2.116482606
 Tamaño matrices:1000
 m1[0][0]+m2[0][0]=m3[0][0](0.600000+3.285714=1971.428571) m1[999][999]+m2[999][999]=m3[999][999](0.600000+3.285714=1971.428571)
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1A/] 2020-05-21 jueves
$./ejecutableB 1000
Tiempo(seg.):1.400555516
 Tamaño matrices:1000
 m1[0][0]+m2[0][0]=m3[0][0](0.600000+3.285714=1971.428571) m1[999][999]+m2[999][999]=m3[999][999](0.600000+3.285714=1971.428571)
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1A/] 2020-05-21 jueves
$
```

1.1. TIEMPOS:

Modificación	Breve descripción de las modificaciones	-O2
Sin modificar	-	*2.370429591 s*
Modificación a)	Desenrollado de bucle	*2.116482606 s*
Modificación b)	Recorremos la matriz por filas en vez de por columnas	*1.400555516 s*

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

Como era de esperar, se mejora el tiempo en ambas modificaciones. También podemos observar que la modificación a, como era de esperar, es mucho peor que la modificación b, ya que esta segunda implica una optimización de mayor dificultad y por lo tanto, mejora mucho más el tiempo obtenido.

Modificación a): Desenrollamos el bucle haciendo 4 iteraciones a la vez. Se optimiza el número de iteraciones del bucle, realizando N/4 iteraciones.

Modificación b): Recorremos la matriz por filas en vez de por columnas, optimizando el recorrido de la misma. Se optimiza al encontrarse los datos más cerca en memoria y evitamos fallos de acceso a caché.

B) CÓDIGO FIGURA 1:

CAPTURA CÓDIGO FUENTE: `figura1.c`

```
19    clock_gettime(CLOCK_REALTIME, &ini);
20
21    for (ii = 0; ii < TAM; ii++) {
22      X1 = 0;
23      X2 = 0;
24      for (i = 0; i < TAM2; i++)
25        X1 += 2 * s[i].a + ii;
26      for (i = 0; i < TAM2; i++)
27        X2 += 3 * s[i].b - ii;
28      if (X1 < X2)
29        R[ii] = X1;
30      else
31        R[ii] = X2;
32    }
33    clock_gettime(CLOCK_REALTIME, &fin);
34
35    total = (double)(fin.tv_sec - ini.tv_sec) +
36          (double)((fin.tv_nsec - ini.tv_nsec) / (1.e+9));
```

1.1. MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación a): Se ha unificado dos bucles que tenían el mismo recorrido, evitando repetir código.

Modificación b): Se ha desenrollado un bucle y se ha mantenido la modificación anterior.

1.1. CÓDIGOS FUENTE MODIFICACIONES

a) y b) Captura figura1-a.c y figura1-b.c

```c
  clock_gettime(CLOCK_REALTIME, &ini);

  for (ii = 0; ii < TAM; ii++) {
    X1 = 0;
    X2 = 0;
    for (i = 0; i < TAM2; i++){
      X1 += 2 * s[i].a + ii;
      X2 += 3 * s[i].b - ii;
    }
    if (X1 < X2)
      R[ii] = X1;
    else
      R[ii] = X2;
  }
  clock_gettime(CLOCK_REALTIME, &fin);
  total = (double)(fin.tv_sec - ini.tv_sec) +
          (double)((fin.tv_nsec - ini.tv_nsec) / (1.e+9));

  printf("\nTiempo (seg): %11.9f\n", total);

  free(R);
}
```

```c
  for (ii = 0; ii < TAM; ii++) {
    X1 = 0;
    X2 = 0;

    for (i = 0; i < TAM2; 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;
    }
    if (X1 < X2)
      R[ii] = X1;
    else
      R[ii] = X2;
  }
  clock_gettime(CLOCK_REALTIME, &fin);
```

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

```
Terminal
Archivo  Editar  Ver  Buscar  Terminal  Ayuda

[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1B/] 2020-05-21 jueves
$gcc -O2 figura1.c -o ejecutable -lrt
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1B/] 2020-05-21 jueves
$gcc -O2 figura1-a.c -o ejecutableA -lrt
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1B/] 2020-05-21 jueves
$gcc -O2 figura1-b.c -o ejecutableB -lrt
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1B/] 2020-05-21 jueves
$./ejecutable

Tiempo (seg): 0.312842123
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1B/] 2020-05-21 jueves
$./ejecutableA

Tiempo (seg): 0.214945505
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1B/] 2020-05-21 jueves
$./ejecutableB

Tiempo (seg): 0.178407850
[GuillermoSandovalSchmidt gsandoval@gsandoval:~/Documentos/Github/AC-UGR/bp4/ejer1B/] 2020-05-21 jueves
$
```

1.1. TIEMPOS:

Modificación	Breve descripción de las modificaciones	-O2
Sin modificar	-	*0.312842123 s*
Modificación a)	Se ha unificado dos bucles que tenían el mismo recorrido	0.214945505 s
Modificación b)	Se ha desenrollado un bucle (+ modificación A)	0.178407850 s

1.2. ENSAMBLADOR:

A) MULTIPLICACIÓN DE MATRICES:

pmm-secuencial.s

```
        call    clock_gettime@PLT
        xorl    %r8d, %r8d
        .p2align 4,,10
        .p2align 3
.L16:
        movq    0(%rbp,%r8), %rdi
        movq    (%r12,%r8), %rsi
        xorl    %ecx, %ecx
        .p2align 4,,10
        .p2align 3
.L13:
        movsd   (%rdi,%rcx), %xmm1
        xorl    %eax, %eax
        .p2align 4,,10
        .p2align 3
.L10:
        movq    (%r14,%rax), %rdx
        movsd   (%rdx,%rcx), %xmm0
        mulsd   (%rsi,%rax), %xmm0
        addq    $8, %rax
        cmpq    %rax, %r15
        addsd   %xmm0, %xmm1
        jne     .L10
        movsd   %xmm1, (%rdi,%rcx)
        addq    $8, %rcx
        cmpq    %rcx, %r15
        jne     .L13
        addq    $8, %r8
        cmpq    %r8, %r15
        jne     .L16
.L12:
        leaq    32(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```

pmm-secuencial-a.s

```
        call    clock_gettime@PLT
        movl    8(%rsp), %eax
        xorl    %r10d, %r10d
        shrl    $2, %eax
        salq    $5, %rax
        leaq    32(%rax), %r11
        .p2align 4,,10
        .p2align 3
.L17:
        movq    0(%r13,%r10), %r8
        movq    (%r12,%r10), %r9
        xorl    %ecx, %ecx
        leaq    (%r11,%r8), %rdi
        .p2align 4,,10
        .p2align 3
.L13:
        pxor    %xmm3, %xmm3
        movq    %rbx, %rdx
        movq    %r8, %rax
        pxor    %xmm2, %xmm2
        pxor    %xmm1, %xmm1
        pxor    %xmm0, %xmm0
        .p2align 4,,10
        .p2align 3
.L10:
        movq    (%rdx), %rsi
        addq    $32, %rax
        addq    $32, %rdx
        movsd   (%rsi,%rcx), %xmm4
        movq    -24(%rdx), %rsi
        mulsd   -32(%rax), %xmm4
        addsd   %xmm4, %xmm0
        movsd   (%rsi,%rcx), %xmm4
        movq    -16(%rdx), %rsi
        mulsd   -24(%rax), %xmm4
        addsd   %xmm4, %xmm1
        movsd   (%rsi,%rcx), %xmm4
        movq    -8(%rdx), %rsi
        mulsd   -16(%rax), %xmm4
        addsd   %xmm4, %xmm2
        movsd   (%rsi,%rcx), %xmm4
        mulsd   -8(%rax), %xmm4
        cmpq    %rax, %rdi
        addsd   %xmm4, %xmm3
        jne     .L10
        addsd   %xmm1, %xmm0
        addsd   %xmm0, %xmm2
        addsd   %xmm2, %xmm3
        movsd   %xmm3, (%r9,%rcx)
        addq    $8, %rcx
        cmpq    %rcx, %r14
        jne     .L13
        addq    $8, %r10
        cmpq    %r10, %r14
        jne     .L17
.L12:
        leaq    32(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```

pmm-secuencial-b.s

```
        call    clock_gettime@PLT
        xorl    %r8d, %r8d
        .p2align 4,,10
        .p2align 3
.L16:
        movq    0(%rbp,%r8), %rdx
        movq    (%r12,%r8), %rdi
        xorl    %esi, %esi
        .p2align 4,,10
        .p2align 3
.L13:
        movq    (%r14,%rsi), %rcx
        movsd   (%rdi,%rsi), %xmm1
        xorl    %eax, %eax
        .p2align 4,,10
        .p2align 3
.L10:
        movsd   (%rcx,%rax), %xmm0
        mulsd   %xmm1, %xmm0
        addsd   (%rdx,%rax), %xmm0
        movsd   %xmm0, (%rdx,%rax)
        addq    $8, %rax
        cmpq    %rax, %r15
        jne     .L10
        addq    $8, %rsi
        cmpq    %rsi, %r15
        jne     .L13
        addq    $8, %r8
        cmpq    %r8, %r15
        jne     .L16
.L12:
        leaq    32(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```

Podemos observar como el código inicial y el B son muy similares ya que realmente los cambios que estamos haciendo solo implican una manera distinta de acceder a los datos en memoria, provocando menos fallos en caché. En el caso del código A, vemos como se obtiene un código ensamblador más largo debido a que, al haber realizado un desenrrollado de bucle, dentro del mismo tendremos que utilizar más registros en una sola iteración, pero realizamos menos iteraciones a cambio, logrando una mejora del tiempo.

B) CÓDIGO FIGURA 1:

figura1.s

```
        call    clock_gettime@PLT
        leaq    40004+s(%rip), %r9
        xorl    %r10d, %r10d
        leaq    -4(%r9), %r8
        .p2align 4,,10
        .p2align 3
.L2:
        leaq    s(%rip), %rax
        movl    %r10d, %edi
        xorl    %esi, %esi
        .p2align 4,,10
        .p2align 3
.L3:
        movl    (%rax), %edx
        addq    $8, %rax
        leal    (%rdi,%rdx,2), %edx
        addl    %edx, %esi
        cmpq    %rax, %r8
        jne     .L3
        leaq    4+s(%rip), %rax
        xorl    %ecx, %ecx
        .p2align 4,,10
        .p2align 3
.L4:
        movl    (%rax), %edx
        addq    $8, %rax
        leal    (%rdx,%rdx,2), %edx
        subl    %edi, %edx
        addl    %edx, %ecx
        cmpq    %rax, %r9
        jne     .L4
        cmpl    %esi, %ecx
        cmovg   %esi, %ecx
        movl    %ecx, (%rbx,%r10,4)
        addq    $1, %r10
        cmpq    $40000, %r10
        jne     .L2
        leaq    16(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```

figura1-a.s

```
        call    clock_gettime@PLT
        leaq    40000+s(%rip), %r8
        xorl    %r9d, %r9d
        .p2align 4,,10
        .p2align 3
.L2:
        leaq    s(%rip), %rax
        movl    %r9d, %edi
        xorl    %ecx, %ecx
        xorl    %esi, %esi
        .p2align 4,,10
        .p2align 3
.L3:
        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    %rax, %r8
        jne     .L3
        cmpl    %ecx, %esi
        cmovl   %esi, %ecx
        movl    %ecx, (%rbx,%r9,4)
        addq    $1, %r9
        cmpq    $40000, %r9
        jne     .L2
        leaq    16(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```

figura1-b.s

```
        call    clock_gettime@PLT
        leaq    40000+s(%rip), %rdi
        xorl    %r8d, %r8d
        .p2align 4,,10
        .p2align 3
.L2:
        leaq    s(%rip), %rax
        movl    %r8d, %edx
        xorl    %ecx, %ecx
        xorl    %r9d, %r9d
        .p2align 4,,10
        .p2align 3
.L3:
        movl    (%rax), %esi
        addq    $32, %rax
        leal    (%rdx,%rsi,2), %r10d
        movl    -28(%rax), %esi
        addl    %r9d, %r10d
        leal    (%rsi,%rsi,2), %esi
        subl    %edx, %esi
        addl    %esi, %ecx
        movl    -24(%rax), %esi
        leal    (%rdx,%rsi,2), %r9d
        movl    -20(%rax), %esi
        addl    %r9d, %r10d
        leal    (%rsi,%rsi,2), %esi
        subl    %edx, %esi
        addl    %ecx, %esi
        movl    -16(%rax), %ecx
        leal    (%rdx,%rcx,2), %r9d
        movl    -12(%rax), %ecx
        addl    %r10d, %r9d
        leal    (%rcx,%rcx,2), %ecx
        subl    %edx, %ecx
        addl    %ecx, %esi
        movl    -8(%rax), %ecx
        leal    (%rdx,%rcx,2), %ecx
        addl    %ecx, %r9d
        movl    -4(%rax), %ecx
        leal    (%rcx,%rcx,2), %ecx
        subl    %edx, %ecx
        addl    %esi, %ecx
        cmpq    %rax, %rdi
        jne     .L3
        cmpl    %ecx, %r9d
        cmovl   %r9d, %ecx
        movl    %ecx, (%rbx,%r8,4)
        addq    $1, %r8
        cmpq    $40000, %r8
        jne     .L2
        leaq    16(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```

Podemos observar entre el código inicial y el A que al unificar dos bucles con el mismo recorrido, nos ahorramos parte del código ensamblador que estaba repetido, mejorando en consecuencia el tiempo. Respecto al código B, al mantener la mejora anterior y añadir el desenrollado del bucle, ocurre igual que en el problema anterior, se obtiene un código ensamblador más largo al utilizar más registros en una sola iteración, pero realizamos menos iteraciones a cambio, logrando una mejora del tiempo.

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];
```

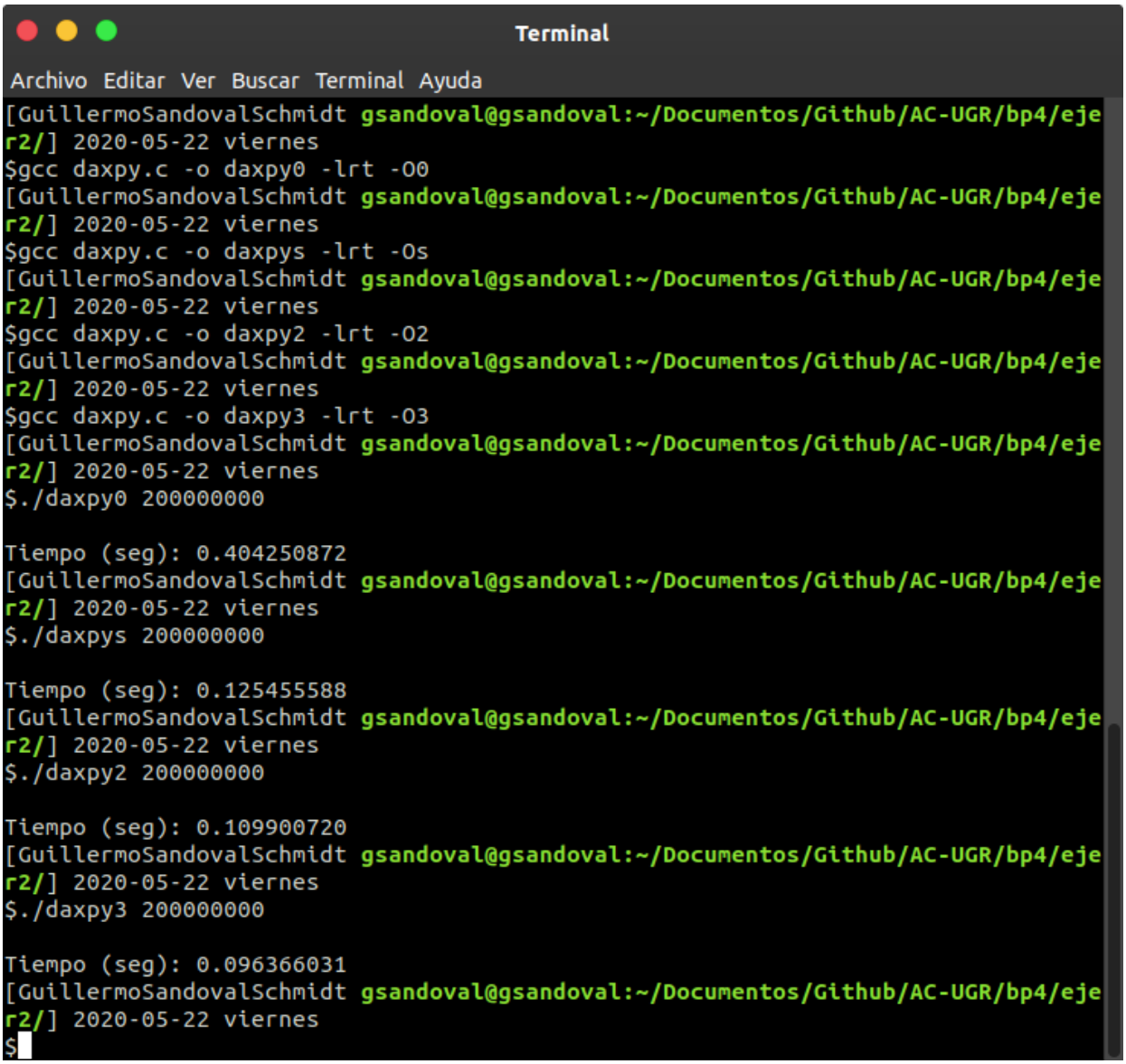
2.1. Genere los programas en ensamblador para cada una de las siguientes opciones de optimización del compilador: -O0, -Os, -O2, -O3. Explique las diferencias que se observan en el código justificando al mismo tiempo las mejoras en velocidad que acarrean. Incorpore los códigos al cuaderno de prácticas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

CAPTURA CÓDIGO FUENTE: `daxpy.c`

daxpy.c

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

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

  if (argc != 2) {
    printf("Falta tamanio\n");
    exit(-1);
  }

  const int N = atoi(argv[1])>INT_MAX ? INT_MAX : atoi(argv[1]);
  const float a = 5.15445455445;
  int i;
  float *x, *y;
  x = (float *)malloc(N * sizeof(float));
  y = (float *)malloc(N * sizeof(float));

  for (i = 0; i < N; i++) {
    x[i] = 23.1313455445513;
    y[i] = 54.54455544554;
  }

  struct timespec ini, fin;
  double total;

  clock_gettime(CLOCK_REALTIME, &ini);

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

  clock_gettime(CLOCK_REALTIME, &fin);

  total = (double)(fin.tv_sec - ini.tv_sec) +
          (double)((fin.tv_nsec - ini.tv_nsec) / (1.e+9));

  printf("\nTiempo (seg): %11.9f\n", total);

  free(x);
  free(y);
}
```

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



TIEMPOS:

Tiempos ejec.	**-O0**	**-Os**	**-O2**	**-O3**
	0.404250872 s	0.125455588 s	0.10990072 s	0.096366031 s

COMENTARIOS QUE EXPLIQUEN LAS DIFERENCIAS EN ENSAMBLADOR:

-O0: optimización por defecto, es decir, como si compilaramos sin opciones de optimización.
-Os: se centra en optimizar el tamaño. Utiliza también las opciones de -O2 (menos algunas específicas) por eso se parecen tanto ambas opciones.*****
-O2: optimiza aún más, sin utilizar optimizaciones que supongan una pérdida de espacio a cambio de un menor tiempo.*****
-O3: el tamaño del código aumenta al utilizar todas las opciones de optimización posibles.

*****Ambas opciones optimizan las instrucciones utilizadas para reducir el tamaño y tiempo de ejecución.

CÓDIGO EN ENSAMBLADOR :

daxpy00.s

```
        call    clock_gettime@PLT
        movl    $0, -84(%rbp)
        jmp     .L5
.L6:
        movl    -84(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -72(%rbp), %rax
        addq    %rdx, %rax
        movss   (%rax), %xmm0
        mulss   -76(%rbp), %xmm0
        movl    -84(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -64(%rbp), %rax
        addq    %rdx, %rax
        movss   (%rax), %xmm1
        movl    -84(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -64(%rbp), %rax
        addq    %rdx, %rax
        addss   %xmm1, %xmm0
        movss   %xmm0, (%rax)
        addl    $1, -84(%rbp)
.L5:
        movl    -84(%rbp), %eax
        cmpl    -80(%rbp), %eax
        jl      .L6
        leaq    -32(%rbp), %rax
        movq    %rax, %rsi
        movl    $0, %edi
        call    clock_gettime@PLT
```

daxpy0s.s

```
        call    clock_gettime@PLT
        xorl    %eax, %eax
        movss   .LC3(%rip), %xmm1
.L5:
        cmpl    %eax, %r12d
        jle     .L11
        movss   (%rbx,%rax,4), %xmm0
        mulss   %xmm1, %xmm0
        addss   0(%rbp,%rax,4), %xmm0
        movss   %xmm0, 0(%rbp,%rax,4)
        incq    %rax
        jmp     .L5
.L11:
        leaq    24(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```

daxpy02.s

```
        call    clock_gettime@PLT
        xorl    %eax, %eax
        movss   .LC3(%rip), %xmm1
        .p2align 4,,10
        .p2align 3
.L6:
        movss   (%r12,%rax), %xmm0
        mulss   %xmm1, %xmm0
        addss   0(%rbp,%rax), %xmm0
        movss   %xmm0, 0(%rbp,%rax)
        addq    $4, %rax
        cmpq    %rax, %rbx
        jne     .L6
.L7:
        leaq    16(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```

daxpy03.s

```
        call    clock_gettime@PLT
        movq    %rbp, %rdx
        shrq    $2, %rdx
        negq    %rdx
        andl    $3, %edx
        leal    3(%rdx), %eax
        cmpl    %r14d, %eax
        ja      .L21
        testl   %edx, %edx
        je      .L22
        movss   .LC5(%rip), %xmm0
        cmpl    $1, %edx
        movss   (%r12), %xmm1
        mulss   %xmm0, %xmm1
        addss   0(%rbp), %xmm1
        movss   %xmm1, 0(%rbp)
        je      .L23
        movss   4(%r12), %xmm1
        cmpl    $3, %edx
        mulss   %xmm0, %xmm1
        addss   4(%rbp), %xmm1
        movss   %xmm1, 4(%rbp)
        jne     .L13
        mulss   8(%r12), %xmm0
        movl    $3, %ebx
        addss   8(%rbp), %xmm0
        movss   %xmm0, 8(%rbp)
.L13:
        movl    %r13d, %r8d
        movaps  .LC6(%rip), %xmm1
        subl    %edx, %r8d
        salq    $2, %rdx
        xorl    %eax, %eax
        movl    %r8d, %edi
        leaq    (%r12,%rdx), %rsi
        xorl    %ecx, %ecx
        shrl    $2, %edi
        addq    %rbp, %rdx
        .p2align 4,,10
        .p2align 3
.L10:
        movups  (%rsi,%rax), %xmm0
        addl    $1, %ecx
        mulps   %xmm1, %xmm0
        addps   (%rdx,%rax), %xmm0
        movaps  %xmm0, (%rdx,%rax)
        addq    $16, %rax
        cmpl    %edi, %ecx
        jb      .L10
        movl    %r8d, %eax
        andl    $-4, %eax
        addl    %eax, %ebx
        cmpl    %eax, %r8d
        je      .L15
.L12:
        movslq  %ebx, %rdx
        movss   .LC5(%rip), %xmm0
        movss   (%r12,%rdx,4), %xmm1
        leaq    0(%rbp,%rdx,4), %rax
        mulss   %xmm0, %xmm1
        addss   (%rax), %xmm1
        movss   %xmm1, (%rax)
        leal    1(%rbx), %eax
        cmpl    %r13d, %eax
        jge     .L15
        cltq
        movss   (%r12,%rax,4), %xmm1
        leaq    0(%rbp,%rax,4), %rdx
        leal    2(%rbx), %eax
        mulss   %xmm0, %xmm1
        cmpl    %r13d, %eax
        addss   (%rdx), %xmm1
        movss   %xmm1, (%rdx)
        jge     .L15
        cltq
        movss   (%r12,%rax,4), %xmm1
        leaq    0(%rbp,%rax,4), %rdx
        leal    3(%rbx), %eax
        mulss   %xmm0, %xmm1
        cmpl    %eax, %r13d
        addss   (%rdx), %xmm1
        movss   %xmm1, (%rdx)
        jle     .L15
        cltq
        addl    $4, %ebx
        movss   (%r12,%rax,4), %xmm1
        leaq    0(%rbp,%rax,4), %rdx
        cmpl    %ebx, %r13d
        mulss   %xmm0, %xmm1
        addss   (%rdx), %xmm1
        movss   %xmm1, (%rdx)
        jle     .L15
        movslq  %ebx, %rbx
        mulss   (%r12,%rbx,4), %xmm0
        leaq    0(%rbp,%rbx,4), %rax
        addss   (%rax), %xmm0
        movss   %xmm0, (%rax)
.L15:
        leaq    16(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime@PLT
```