Generación y optimización de código

María Luz Mosteiro Del Pilar

10 de diciembre de 2017

Índice

1. Introducción

Este informe tiene como objetivo analizar el efecto de distintos niveles de optimización y técnicas, como por ejemplo, el desenrrollamiento de lazos.

Todos los códigos han sido compilados con gcc 5.4.0 en una máquina con sistema operativo ubuntu 16.04 de 64 bits.

2. Caso 1. Multiplicación de matrices

2.1. Preprocesado

La opción -E, realiza solamente el preprocesamiento, enviando el resultado a la salida estándar.

```
$ gcc -E ejercicio2.c
```

En el código tenemos la macro `#DEFINE Nmax 600`, que determina la dimensión máxima de nuestras matrices. La salida que obtenemos es el código expandido, sin compilar, en el que se sustituyen las macros. Así pues, aquellos sitios en los que aparecía la macro `Nmax`, ahora aparece el valor 600.

```
1 int main(int argc, char *argv[]){
2   float A[600][600], B[600][600], C[600][600], t, r;  //Vemos como se sustituye el
        tamano del array
```

```
3     int i,j,k, maxIt;
4
5     if (argc<2){
6       printf("[ERROR] Indicar numero de iteraciones como argumento\n");
7       return -1;
8     } else{
9       maxIt = atoi(argv[1]);
10      tiempo=0;
11    }
12
13    for(i=0;i<600;i++){  // Aqui se sustituye por el valor maximo de i
14      for(j=0;j<600;j++) { // Aqui se sustituye por el valor maximo de j
15        A[i][j]=(i+j)/(j+1.1);
16        B[i][j]=(i-j)/(j+2.1);
17      }
18    }
19
20    for(int it=0; it<maxIt; it++){
21      gettimeofday(&inicio,
22      # 32 "ejercicio2.c" 3 4
23                       ((void *)0)
24      # 32 "ejercicio2.c"
25                           );
26
27      for(i=0;i<600;i++){  // Igual que el caso anterior, con i
28        for(j=0;j<600;j++) { // Igual que el caso anterior, con j
29          t=0;
30          for (k=0;k<600;k++) { // Igual que el caso anterior, con k
31            producto(A[i][k],B[k][j],&r);
32            t+=r;
33          }
34          C[i][j]=t;
35        }
36      }
```

2.2. Código ensamblador

La opción -S genera únicamente el código ensamblador propio del procesador de nuestra máquina. No es frecuente realizar sólo el ensamblado, lo usual es realizar todas las etapas anteriores hasta obtener el código objeto.

```
$ gcc -S ejercicio2.c
```

En está sección nos centraremos en tres aspectos interesantes del código ensamblador que nos proporciona el compilador: cómo se implementan los lazos, cómo se hacen las llamadas a funciones y cómo son las operaciones en punto flotante. Además, se analizará la sintaxis de las instrucciones más representativas dentro de cada caso.

2.2.1. Implementación de lazos

En el código ensamblador mostrado a continuación podemos destacar el uso de instrucciones de salto y comparación para implementar los lazos. El **salto incondicional**, `jmp` se realiza siempre. Sin embargo, el **salto condicional** se ejecuta solo si se cumple algún requisito. En ese caso podemos ver que primero se realiza una comparacion, `cmp`, y luego se verifica si la condición se cumple o no en la instrucción de salto. Existen bastantes tipos de saltos condicionales: `je` (si igual), `jne` (si no igual), `jg` (si mayor), `jge` (si mayor o igual), `jl` (si menor), `jle` (si menor o igual), `ja` (si mayor sin signo), `jae` (si mayor o igual sin signo), `jb` (si menor sin signo), `jbe` (si menor o igual sin signo), etcétera. En este caso la instrucción que se usa es la de menor o igual.

Además, podemos notar que la mayoría de las instrucciones están acompañadas por un sufijo: `movq`, `movl` o `addq`. En realidad, hay mnemónicos (abreviatura con la que se hace referencia a cada instrucción) diferentes para distintos tamaños de datos (q=quadword=64bits; l=long=32 bits; w=word=16 bits; b=byte=8 bits).

```
1
2   .L3:
3           movq    -4320080(%rbp), %rax
4           addq    $8, %rax
5           movq    (%rax), %rax
6           movq    %rax, %rdi
7           call    atoi
8           movl    %eax, -4320028(%rbp)
9           pxor    %xmm0, %xmm0
10          movsd   %xmm0, tiempo(%rip)
11          movl    $0, -4320044(%rbp) // Asignar 0 a donde parece que esta la variable i
12          jmp     .L5 // Salto incondicional a L5
13  .L8:
14          movl    $0, -4320040(%rbp) // Asignar 0 a donde parece que esta la variable j
15          jmp     .L6 // Salto incondicional a L6
16  .L7: // Operaciones correspondientes a la asignacion de valores
17          movl    -4320044(%rbp), %ed
18          movl    -4320040(%rbp), %eax
19          addl    %edx, %eax
20          .
21          .
22          .
23          addq    %rdx, %rax
24          movss   %xmm0, -2880016(%rbp,%rax,4)
25          addl    $1, -4320040(%rbp) // Suma 1 a la variable j
26  .L6:
27          cmpl    $599, -4320040(%rbp) // Comprobacion para salir del bucle de j
28          jle     .L7 // Salto condicional a L7 si el valor es menor o igual
29          addl    $1, -4320044(%rbp) // Sumamos 1 a la variable i
30  .L5:
31          cmpl    $599, -4320044(%rbp) // Comprobacion para salir del bucle de i
32          jle     .L8 // Salto condicional a L8 si se cumple
33          movl    $0, -4320032(%rbp) // Si no se cumple, salimos.
34          jmp     .L9 // Salto a L9, ya corresponde al siguiente bucle
```

2.2.2. Llamadas a funciones

En el código ensamblador mostrado a continuación podemos observar como es que se llama a una subrutina. Para ello se utilizan instrucciones para el manejo del stack, de llamada y retorno de subrutinas y de transferencia de datos.

En primer lugar, podemos notar los registros `rbp`, *base pointer*, que apunta a la base del stack actual y a partir del cual podemos direccionar los parámetro o variables locales; y `rsp`, *stack pointer*, que apunta a la cima del stack. Podemos ver como se almacenan o se retiran del stack con las instrucciones `push` y `pop`.

Por otro lado, el uso de la instrucción `mov` antes de llamar a la subrutina tiene lugar pues es necesario almacenar el valor de los registros que pueden ser machacados por esta, de forma que su valor pueda ser restaurado al acabar la misma.

Por último, las instrucciones en ensamblador para llamar a la subrutina o volver a la rutina invocadora son `call` y `ret`, respectivamente.

```
1   producto:       // Etiqueta de la funcion producto
2   .LFB2:
3           .cfi_startproc
4           pushq   %rbp // Almacena el actual base pointer en la cima de la pila
```

```
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp      // El base pointer ahora es el stack pointer
        .cfi_def_cfa_register 6
        movss   %xmm0, -4(%rbp)
        movss   %xmm1, -8(%rbp)
        movq    %rdi, -16(%rbp)
        movss   -4(%rbp), %xmm0
        mulss   -8(%rbp), %xmm0
        movq    -16(%rbp), %rax
        movss   %xmm0, (%rax)
        nop
        popq    %rbp // Saca el base pointer de la pila
        .cfi_def_cfa 7, 8
        ret // Retorno a la instruccion donde se hizo la llamada
        .cfi_endproc

// CODIGO ENSABLADOR EXTRA AQUI//

.L13:
        movl    -4320040(%rbp), %eax    // Preparacion de los parametros
        cltq
        movl    -4320036(%rbp), %edx
        movslq  %edx, %rdx
        imulq   $600, %rdx, %rdx
        addq    %rdx, %rax
        movss   -2880016(%rbp,%rax,4), %xmm0
        movl    -4320036(%rbp), %eax
        cltq
        movl    -4320044(%rbp), %edx
        movslq  %edx, %rdx
        imulq   $600, %rdx, %rdx
        addq    %rdx, %rax
        movl    -4320016(%rbp,%rax,4), %eax
        leaq    -4320052(%rbp), %rdx
        movq    %rdx, %rdi
        movaps  %xmm0, %xmm1
        movl    %eax, -4320088(%rbp)
        movss   -4320088(%rbp), %xmm0
        call    producto        // Llamada a la funcion
        movss   -4320052(%rbp), %xmm0
        movss   -4320048(%rbp), %xmm1
        addss   %xmm1, %xmm0
        movss   %xmm0, -4320048(%rbp)
        addl    $1, -4320036(%rbp)
```

2.2.3. Operaciones en punto flotante

En el código ensamblador mostrado a continuación se muestra como se realizan las operaciones en punto flotante. Podemos destacar el uso de registros especiales, estos utilizan una definición como la siguiente `xmm0`, `xmm1`, `xmm7`. Mediante ellos se pueden hacer operaciones con lo la unidad de punto flotante, FPU.

Tambien se utilizan instrucciones especificas para tratar con datos de más de 32 bits, por ejemplo `MOVSS`, *move scalar single-precision floating-point value*, copia los 32 bits menos significativos del primer registro al segundo.

```
producto:
.LFB2:
        .cfi_startproc
```

```
4           pushq   %rbp
5           .cfi_def_cfa_offset 16
6           .cfi_offset 6, -16
7           movq    %rsp, %rbp
8           .cfi_def_cfa_register 6
9           movss   %xmm0, -4(%rbp)     // Uso de registros especificos de punto flotante
10          movss   %xmm1, -8(%rbp)     // Uso de registros especificos de punto flotante
11          movq    %rdi, -16(%rbp)
12          movss   -4(%rbp), %xmm0
13          mulss   -8(%rbp), %xmm0     // instruccion de multiplicacion
14          movq    -16(%rbp), %rax
15          movss   %xmm0, (%rax)
16          nop
17          popq    %rbp
18          .cfi_def_cfa 7, 8
19          ret
20          .cfi_endproc
```

2.3. Código objeto

La opción -c, realiza el preprocesamiento y compilación, obteniendo el archivo en código objeto; no realiza el enlazado.

```
$ gcc -c ejercicio2.c
```

En general, el código objeto es una secuencia de instrucciones en un lenguaje que entiende el ordenador directamente, y por lo tanto es difícil de entender para los humanos. Podemos comprobar fácilmente que, efectivamente, lo que el compilador genera con esta opción es código objeto.

En primer lugar, si abrimos el archivo, en este caso *objeto.out*, con un editor de texto, podemos apreciar inmediatamente que el contenido no es legible para un ser humano. A continuación se muestran las primeras 15 lineas del archivo.

```
7f45 4c46 0201 0100 0000 0000 0000 0000
0100 3e00 0100 0000 0000 0000 0000 0000
0000 0000 0000 0000 300a 0000 0000 0000
0000 0000 4000 0000 0000 4000 0d00 0a00
5548 89e5 f30f 1145 fcf3 0f11 4df8 4889
7df0 f30f 1045 fcf3 0f59 45f8 488b 45f0
f30f 1100 905d c355 4889 e548 81ec 60eb
4100 89bd bc14 beff 4889 b5b0 14be ff64
488b 0425 2800 0000 4889 45f8 31c0 83bd
bc14 beff 017f 14bf 0000 0000 e800 0000
00b8 ffff ffff e95b 0300 0048 8b85 b014
beff 4883 c008 488b 0048 89c7 e800 0000
0089 85e4 14be ff66 0fef c0f2 0f11 0500
0000 00c7 85d4 14be ff00 0000 00e9 df00
0000 c785 d814 beff 0000 0000 e9b9 0000
```

Por otro lado, para finalizar todas las etapas hasta obtener un ejecutable, podemos proveer a gcc con el código objeto directamente, y posteriormente ejecutar nuestro programa.

```
$ gcc -o ejecutable objeto.out
$ ./ejecutable
```

2.4. Enlazado estático

Con la opción -static, se realiza el enlazado estático de las librerías. Si no se especifica esta opción, el enlace es dinámico.

```
$ gcc -static ejercicio2.c
```

Existen dos modos de realizar el enlace: el estático, donde se incorporan al fichero ejecutable el código de las librerías necesarias; y el dinámico, caso en el que el ejecutable cargará en memoria la librería y ejecutará la parte de código correspondiente en el momento de correr el programa. Si bien el enlazado dinámico genera un ejecutable más pequeño, el enlazado estático genera ficheros autónomos, aunque de mayor tamaño.

Estático	**Dinámico**
912904	8800

2.5. Optimizaciones

Se realizaron las optimizaciones O0, O1, O2, O3 y Os. Sin ninguna opción de optimización, el objetivo del compilador es reducir el costo de la compilación y hacer que la depuración produzca los resultados esperados. Al activar los indicadores de optimización, el compilador intenta mejorar el rendimiento y / o el tamaño del código a expensas del tiempo de compilación y posiblemente la capacidad de depurar el programa.

2.5.1. Análisis de tiempos

Para la medida del tiempo de ejecución se toman en cuenta únicamente las operaciones correspondientes al producto de matrices. Además, para disminuir el efecto de datos atípicos provocados por interrupciones del sistema y bloqueos, se han realizado 100 iteraciones sobre la operación completa de multiplicación de matrices. Como valor representativo del tiempo de ejecución consideraremos la media. El tamaño se mide en bytes y el tiempo en segundos.

Optimización	**Tamaño**	**Tiempo**
O0	2832	2.325104
O1	2704	0.439123
O2	2400	0.000005
O3	2400	0.000005
Os	2256	0.000005

Podemos observar que, sin duda, el tiempo mejora cuanto mayor es el nivel de optimización. Sin embargo, como veremos en la siguiente sección 2.5.2, es difícil identificar las zonas de código ensamblador en las que se realizan las operaciones correspondientes con el producto de matrices, por lo cual, es probable que el código generado no produzca resultados correctos.

Por otro lado, también se puede observar cómo existe una relación entre el el tamaño del objeto y el nivel de optimización. Además la optimización Os, especifica para optimizar tamaño, genera un código objeto con un tamaño muy inferior al que genera el compilador por defecto.

2.5.2. Análisis código ensamblador

Es importante mencionar que cuanto mayor es el nivel de optimización, más difícil es interpretar el código ensamblador, por lo que algunas de las instrucciones pueden estar mal interpretadas. Es probable, también, que los niveles más altos no generen resultados correctos, pues son optimizaciones muy agresivas.

-O0, predeterminado. Reduce el tiempo de compilación y hace que la depuración produzca los resultados esperados. Consideraremos este caso como el punto de referencia para realizar las comparaciones respecto al tiempo de ejecución y al tamaño del código objeto. Todo lo destacable del código ensamblador coincide con lo mencionado en la sección 2.2.

-O1, primer nivel de optimización. La optimización de la compilación requiere algo más de tiempo y mucha más memoria para una función grande.El compilador intenta reducir el tamaño del código y el tiempo de ejecución, sin realizar ninguna optimización que requiera una gran cantidad de tiempo de compilación.

En el código ensamblador podemos interpretar algunas cosas interesantes. En primer lugar, las operaciones de la subrutina `producto` que veíamos en el caso de la optimización O0 ahora se hacen directamente con el uso de etiquetas, como podemos ver en `.L10`. Se preparan los datos para trabajar directamente con las instrucciones para datos en punto flotante (`movss`, `mulss`, `addss`) usando los registros especificos `xmm`, `xmm0` y `xmm1`. También se utiliza la instrucción `leaq` para calcular las direcciones, *Load Effective Address*, es una forma de invocar la lógica en la AGU, *Unidad de Generación de Direcciones* y hacer que calcule expresiones simples.

```
.L14:
        movl    $0, %esi
        movl    $inicio, %edi
        call    gettimeofday // Llamada a gettimeofday
        leaq    2882416(%rsp), %r8
        leaq    16(%rsp), %rdi
        jmp     .L9 //Salto a L9
.L10:
        movss   (%rax), %xmm0 // Uso de registros
        mulss   (%rdx), %xmm //especiales de punto flotante
        addss   %xmm0, %xmm1
        addq    $2400, %rax
        addq    $4, %rdx
        cmpq    %rax, %rcx // implementacion del bucle
        jne     .L10 // saltando a la misma etiqueta
        movss   %xmm1, (%rsi) // vuelta a los registros de tipo r
        addq    $4, %rsi // aqui se suma 1 a i y a j
        addq    $4, %rcx
        cmpq    %rsi, %r8 // en r8 esta el maximo de iteraciones
        je      .L11
.L13:
        leaq    -1440000(%rcx), %rax
        movq    %rdi, %rdx
        pxor    %xmm1, %xmm // uso del xor con los registros de pf
        jmp     .L10 //Salto para realizar los calculos de producto
.L11:
        addq    $2400, %r8
        addq    $2400, %rdi
        cmpq    %rdi, %r12
        je      .L12 // Fin de las operaciones
.L9:
        leaq    -2400(%r8), %rsi
        leaq    2880016(%rsp), %rcx
        jmp     .L13
```

-O2. GCC realiza casi todas las optimizaciones admitidas que no implican una relación de sacrificio entre velocidad y memoria. En comparación con -O1, esta opción aumenta tanto el tiempo de compilación como el rendimiento del código generado.

Podemos notar que claramente en este caso realizar la correspondencia entre código de alto nivel y código ensamblador resulta mucho más complicado. Podemos ver que tampoco hay una llamada explicita a la subrutina `producto`. Algunos de los saltos condicionales se realizan con un `xor` en lugar de con una instrucción `cmp`. Además, podemos intuir que se usan instrucciones del tipo `movq%fs:40,%rax` para gestionar los elementos de la matriz, realizando un offset (desplazamiento).

```
main:
.LFB39:
        .cfi_startproc // directiva para debug, marca inicio
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        subq    $1440016, %rsp
        .cfi_def_cfa_offset 1440032
        movq    %fs:40, %rax    // offset
        movq    %rax, 1440008(%rsp)
        xorl    %eax, %eax      // Parece que en eax y en edi estan los indices
        cmpl    $1, %edi        //  que controla la matrix
        jle     .L8
        movq    8(%rsi), %rdi
```

```
        ... OPERACIONES PARA EL PRINTF ...
.L4:
        movq    1440008(%rsp), %rcx
        xorq    %fs:40, %rcx    // Este salto condicional se realiza con
        jne     .L9                 // un xor
        addq    $1440016, %rsp
        .cfi_remember_state // directiva para debug, estado anterior
        .cfi_def_cfa_offset 16
        popq    %rbx // hacemos pop
        .cfi_def_cfa_offset 8
        ret  // FINALIZA EJECUCION
.L8:
        .cfi_restore_state
        movl    $.LC1, %edi
        call    puts
        movl    $-1, %eax
        jmp     .L4
```

-O3. Activa todas las optimizaciones especificadas por -O2 y además otras relacionadas con bucles, redundancias entre iteraciones de bucles, vectores, etcétera. En este caso, si analizamos ambos archivos podemos ver que no existe ninguna diferencia entre las dos optimizaciones, por lo que podemos asegurar que el compilador no fue capaz de mejorar el rendimiento. Para comparar las diferencias entre archivos basta con ejecutar el siguiente código en una terminal.

```
$ diff ejercicio2_O2_ensamblador.out ejercicio2_O3_ensamblador.out
```

-Os, optimizar para el tamaño. -Os habilita todas las optimizaciones de -O2 que normalmente no aumentan el tamaño del código. También realiza optimizaciones adicionales diseñadas para reducir el tamaño del código. En el código ensamblador podemos encontrar algunas diferencias, entre ellas, el uso de instrucciones para decrementar como `decl` que sirven para realizar los saltos condicionales, el uso de un número menor de etiquetas en el código o el uso de la llamada a `atoi` en lugar de `strtol`.

```
main:
.LFB21:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        subq    $1440016, %rsp
        .cfi_def_cfa_offset 1440032
        movq    %fs:40, %rax
        movq    %rax, 1440008(%rsp)
        xorl    %eax, %eax
        decl    %edi // instruccion para decrementar 1
        jg      .L3
        movl    $.LC1, %edi
        call    puts
        orl     $-1, %eax
        jmp     .L4
.L3:
        movq    8(%rsi), %rdi
        call    atoi // en lugar de llamar a strtol
        cvtss2sd        1442408(%rsp), %xmm0
        ...INSTRUCCIONES PARA PRINTF...
.L4:
        movq    1440008(%rsp), %rcx
        xorq    %fs:40, %rcx
        je      .L5             //Salto a finalizacion de ejecucion
        call    __stack_chk_fail // en caso de  fallo, no hay etiqueta extra
```

```
.L5: // no hay directivas de debug del estilo remember state o restore state
        addq    $1440016, %rsp
        .cfi_def_cfa_offset 16
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

3. Caso 2. Análisis de optimización de lazos

En este caso se optimizó el mismo código con la opción `-O1` y la opción `-O1 -funroll-loops`. Analizaremos las principales diferencias en cuanto a código ensamblador y su comportamiento en termino de tiempo de ejecución para distintos valores de N.

3.1. Análisis del código ensamblador

En primer lugar, analicemos el caso de primer bucle. Con la optimización `-O1`, podemos ver que se implementa con comparaciones y saltos a etiquetas para realizar el numero de iteraciones necesarias. En general, podemos decir que este es el comportamiento por defecto.

```
.L2:
        pxor    %xmm0, %xmm0
        cvtsi2sd        %eax, %xmm0
        mulsd   %xmm1, %xmm0
        movsd   %xmm0, res(,%rax,8)
        addq    $1, %rax
        cmpq    $10000, %rax    // Primer bucle
        jne     .L2 // salto a la misma etiqueta
        movl    $res, %eax
        movl    $res+80000, %ecx
        movsd   .LC1(%rip), %xmm2
        movsd   .LC2(%rip), %xmm4
        movsd   .LC0(%rip), %xmm3
```

Por otro lado, con la opción `-O1 -funroll-loops`, podemos ver que el compilador *desenrolla* el bucle, escribiendo el mismo grupo de instrucciones un numero determinado de veces, en este caso, 8 iteraciones. Así el numero de saltos que tiene que realizar a la etiqueta `.L2` es menor.

```
.L2:
        pxor    %xmm1, %xmm1
        cvtsi2sd        %eax, %xmm1
        mulsd   %xmm0, %xmm1
        movsd   %xmm1, res(,%rax,8)
        addq    $1, %rax // 1 unroll
        pxor    %xmm2, %xmm2
        cvtsi2sd        %eax, %xmm2
        mulsd   %xmm0, %xmm2
        movsd   %xmm2, res(,%rax,8)
        leaq    1(%rax), %rcx // 2 unroll
        pxor    %xmm3, %xmm3
        cvtsi2sd        %ecx, %xmm3
        mulsd   %xmm0, %xmm3
        movsd   %xmm3, res(,%rcx,8)
        leaq    2(%rax), %rsi // 3 unroll

... HAEC UNROLL DE 8 ITERACIONES ...
```

```
19          pxor    %xmm8, %xmm8
20          cvtsi2sd         %r10d, %xmm8
21          mulsd   %xmm0, %xmm8
22          movsd   %xmm8, res(,%r10,8)
23          addq    $7, %rax // 8 unroll
24          cmpq    $10000, %rax
25          jne     .L2  // Salto a la misma etiqueta
26          movl    $res, %edx
27          movl    $res+80000, %r11d
28          movsd   .LC1(%rip), %xmm9
29          movsd   .LC2(%rip), %xmm10
30          movsd   .LC0(%rip), %xmm11
```

El segundo bucle es interesante por la estructura *if-else* que hay en su interior. Al igual que el caso inicial del primer bucle, en este caso se implementa el lazo con instrucciones de comparación `cmp` y se salto `jne`. Si observamos con cuidado podemos identificar las instrucciones que corresponden con la estructura *if-else*

```
1   .L6:    //inicio segundo bucle
2           movq    %rax, %rdx
3           movsd   (%rax), %xmm0
4           ucomisd %xmm0, %xmm2
5           jbe     .L10 // salto al else
6           movapd  %xmm0, %xmm1 // caso del if
7           mulsd   %xmm0, %xmm1
8           addsd   %xmm3, %xmm1
9           jmp     .L5
10  .L10: // caso del else
11          movapd  %xmm0, %xmm1
12          subsd   %xmm4, %xmm1
13  .L5:
14          addsd   %xmm1, %xmm0
15          movsd   %xmm0, (%rdx)
16          addq    $8, %rax
17          cmpq    %rcx, %rax
18          jne     .L6 //salto a l6, inicio del segundo bucle
19          subq    $8, %rsp
```

Con la opción `-O1 -funroll-loops`, al contrario de lo que podríamos pensar, podemos ver que el compilador también *desenrolla* el bucle, escribiendo el grupo de instrucciones completo (incluidas las relacionadas con el *if-else*) 8 iteraciones. De nuevo, el numero de saltos que tiene que realizar a la etiqueta `.L6` es menor.

```
1   .L6: // INICIO DEL BUCLE
2           movq    %rdx, %rax
3           movsd   (%rdx), %xmm12
4           ucomisd %xmm12, %xmm9
5           jbe     .L31  // salta al else
6           movapd  %xmm12, %xmm13 // caso del if
7           mulsd   %xmm12, %xmm13
8           addsd   %xmm11, %xmm13
9           jmp     .L5
10  .L31:  // caso del else
11          movapd  %xmm12, %xmm13
12          subsd   %xmm10, %xmm13
13  .L5: // ITERACION 2
14          addsd   %xmm13, %xmm12
15          movsd   %xmm12, (%rax)
16          leaq    8(%rdx), %rcx
17          movsd   8(%rdx), %xmm14
18          ucomisd %xmm14, %xmm9
```

```
19          ja      .L10 // Salta al if
20          jmp     .L32 // salta al else
21  .L40:
22          .cfi_def_cfa_offset 16
23          movsd   res+79992(%rip), %xmm0
24          movl    $.LC3, %esi
25          movl    $1, %edi
26          movl    $1, %eax
27          call    __printf_chk
28          movl    $0, %eax
29          addq    $8, %rsp
30          .cfi_def_cfa_offset 8
31          ret // FIN DE PROGRAMA
32  .L32: // caso del ELSE
33          movapd  %xmm14, %xmm15
34          subsd   %xmm10, %xmm15
35          jmp     .L33
36  .L10: // caso del IF
37          movapd  %xmm14, %xmm15
38          mulsd   %xmm14, %xmm15
39          addsd   %xmm11, %xmm15
40  .L33: // ITERACION 3
41          addsd   %xmm15, %xmm14
42          movsd   %xmm14, (%rcx)
43          movsd   8(%rcx), %xmm0
44          ucomisd %xmm0, %xmm9
45          ja      .L12 // salta al if
46          movapd  %xmm0, %xmm1
47          subsd   %xmm10, %xmm1 // caso del else
48          jmp     .L34
49  .L12: // caso del if
50          movapd  %xmm0, %xmm1
51          mulsd   %xmm0, %xmm1
52          addsd   %xmm11, %xmm1
53  .L34: // ITERACION 4
54  ... HAEC UNROLL DE 8 ITERACIONES ...
55  .L39:
56          addsd   %xmm14, %xmm13
57          movsd   %xmm13, 48(%rcx)
58          leaq    56(%rcx), %rdx
59          cmpq    %r11, %rdx
60          jne     .L6 // salto a inicio del bucle
61          subq    $8, %rsp
62          .cfi_def_cfa_offset 16
63          jmp     .L40
64          .cfi_endproc
```

3.2. Análisis de tiempos

N	-O1	-O1 -funroll-loops
10	0.000005	0.000005
100	0.000005	0.000005
1000	0.000008	0.000006
10000	0.000052	0.000017
100000	0.000528	0.000193
1000000	0.005011	0.001878
10000000	0.046881	0.016250
100000000	0.457550	0.141985

Para cada N, se han realizado 50 iteraciones. Así pues, el valor elegido como tiempo de ejecución es la media de todos resultados. De este modo se intenta disminuir el efecto de valores atípicos en la medición causados por interrupciones u bloqueos.

Como se puede apreciar en los tiempos, la opción `-funroll-loops` mejora el rendimiento para tamaños de N grandes mientras que para tamaños pequeños los prácticamente iguales. Esto quiere decir que no es rentable aumentar el número de operaciones por cada iteración del bucle cuando tenemos pocas iteraciones, pues sacrificamos mucha memoria por una mejora apenas apreciable.

4. Bibliografia

- A. González Barbone Víctor. *El compilador GCC.* Instituto de Ingeniería Eléctrica - Facultad de Ingeniería - Montevideo, Uruguay. *(https://iie.fing.edu.uy/ vagonbar/gcc-make/gcc.htm)*
- 3.10 Options That Control Optimization *(https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html#Optimize-Options)*
- *Lenguajes del computador:alto nivel, ensamblador y máquina.* Departamento de Ingeniería y Tecnología de Computadores. Universidad de Murcia. *(http://ditec.um.es/ jpujante/documentos/Tema5-slides.pdf)*