Generación y optimización de código

Compiladores e Intérpretes

19/12/2014

Brais López Yáñez

Índice

Objetivo de la práctica

La finalidad de esta práctica es la generación y optimización de código en lenguaje C, para esto se utilizarán diferentes formas de compilar, con gcc: -O0, -O1, -O2, -O3, -Os. Además en algunos casos se generará código objeto y código en lenguaje ensamblador, que será comentado.

Descripción del ordenador empleado

La realización de esta práctica se ha realizado en una máquina virtual de Ubuntu, se ha utilizado el VirtualBox, como software de virtualización.

Estas son las características del sistema operativo empleado:

2. Compilación con gcc.

Apartado h

En este apartado vamos a compilar con las opciones -O0, -O1, -O2, -O3, -Os. Después compararemos el tamaño de los códigos objeto, de cada compilación. Luego compararemos los tiempos de ejecución, de las opciones anteriormente comentadas. Por último haremos una breve comparación de los códigos en ensamblador.

Código en C

```
#include <sys/time.h>
#include <stdio.h>
#include <math.h>
#define Nmax 600

void producto(float x, float y, float *z){
*z=x*y;}

main()
```

```
{
struct timeval inicio, final;
double tiempo;
float A[Nmax][Nmax],B[Nmax][Nmax],C[Nmax][Nmax],t,r;
int i,j,k;

gettimeofday(&inicio,NULL);

for(i=0;i<Nmax;i++)/*Valores de las matrices*/
        for(j=0;j<Nmax;j++){
                A[i][j]=(i+j)/(j+1.1);
                B[i][j]=(i-j)/(j+2.1);}
for(i=0;i<Nmax;i++)
        for(j=0;j<Nmax;j++){
                t=0;
                for(k=0;k<Nmax;k++){
                        producto(A[i][k],B[k][j],&r);
                        t+=r;}
                C[i][j]=t;}


gettimeofday(&final,NULL);
for(i=0;i<Nmax;i++)
        for(j=0;j<Nmax;j++){
printf("%f", C[i][j]);

}
tiempo=(final.tv_sec-inicio.tv_sec+(final.tv_usec-inicio.tv_usec)/1.e6);

printf("\nEl tiempo es: %f\n",tiempo);
}
```

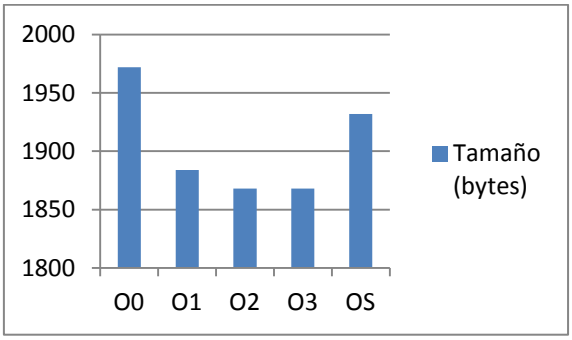
Para empezar este ejercicio 2, vemos que el código está compuesto por varios bucles anidados, donde inicializan varias matrices. Esto es lo que se va a medir también para el tiempo de ejecución.

Tamaño de los códigos objeto

Una vez compilamos con la opción gcc -c, generamos el código objeto para cada opción de optimización.

Con la opción -O0, no se genera ningún tipo de mejora y generamos el mayor tamaño, como se puede contemplar en la tabla y el gráfico adjunto. A medida que ejecutamos las demás opciones de optimización conseguimos mejorar los tamaños. Sin embargo con ´-O2 y -O3, el tamaño se incrementa un poco, quedando muy parejo. Con la opción de optimización -Os, el tamaño del código objeto vuelve a aumentar.

-O0	-O1	-O2	-O3	-Os
1972 Bytes	1884 Bytes	1868 Bytes	1868 Bytes	1932 Bytes

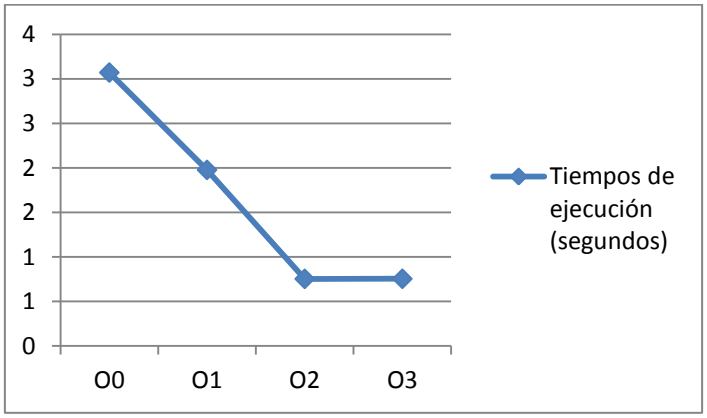


Después de comprobar los archivos de cada código objeto, compilamos y ejecutamos para ver los tiempos de ejecución. Aquí tras hacer las mediciones, y calcular su media, obtenemos que por cada opción de optimización el tiempo se va reduciendo, aunque a excepción de la última optimización empleada, puesto que el tiempo es muy similar a la opción -O2.

Sin embargo estos tiempos obtenidos, pueden variar de un código a otro, beneficiándose más de alguna opción de optimización.

Comparación de tiempos

-O0	-O1	-O2	-O3
3,07066 s	1,97716156 s	0,75362689 s	0,75438856 s



Para acabar con el ejercicio 2, haré una breve comparación entre los códigos ensamblador generados. Con la opción -O0, no se generan mejoras, ya que no hay optimización. Se realizan saltos que se pueden reducir, lo que significa que es un código ineficiente. Con la optimización -O1, se mejora cuando se accede a memoria, dando lugar a un código más optimizado.

Con las optimizaciones -O2 y -O3, se ordena el código recolocando bloques, y se disminuye la realización de saltos, en definitiva, se genera un código más secuencial.

3.Comparación de los códigos ensamblador.

En este ejercicio, vamos a generar diferentes códigos ensamblador a raíz de compilar con gcc -s con las opciones: -O0,O1, -O2, -O3.

Código en C

```
#include <stdio.h>
#define Nmax 10

main(){

int i, j, k, t;

j=1;
t=1;
```

```
for(i=0;i<Nmax;i++){
        t++;
        j=j+i;
        k=j-t;
        k=k+t;
        j=j+k;
        printf("Resultado = %d\n",j);}
}
```

Este código, contiene un bucle en el que se realizan diferentes tipos de operaciones.

Al generar el código ensamblador con las diferentes opciones de optimización, podemos ver que la versión de nuestro GCC es la 4.6.3., esto nos va a dar una explicación más detallada de las diferentes opciones que tiene esta versión de gcc. Y con las que vamos a explicar los diferentes códigos.

Optimización: -O0

Para este código ensamblador no habría ningún tipo de optimización, ya que es la opción por defecto del gcc.

```
        .file   "3.c"
        .intel syntax noprefix
        .section        .rodata
.LC0:
        .string "Resultado = %d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        push    ebp
        .cfi def cfa offset 8
        .cfi offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        and     esp, -16
        sub     esp, 32
        mov     DWORD PTR [esp+20], 1
        mov     DWORD PTR [esp+24], 1
        mov     DWORD PTR [esp+16], 0
        jmp     .L2
.L3:
        add     DWORD PTR [esp+24], 1
        mov     eax, DWORD PTR [esp+16]
        add     DWORD PTR [esp+20], eax
        mov     eax, DWORD PTR [esp+24]
        mov     edx, DWORD PTR [esp+20]
        mov     ecx, edx
        sub     ecx, eax
        mov     eax, ecx
        mov     DWORD PTR [esp+28], eax
        mov     eax, DWORD PTR [esp+24]
        add     DWORD PTR [esp+28], eax
        mov     eax, DWORD PTR [esp+28]
        add     DWORD PTR [esp+20], eax
        mov     eax, OFFSET FLAT:.LC0
        mov     edx, DWORD PTR [esp+20]
        mov     DWORD PTR [esp+4], edx
        mov     DWORD PTR [esp], eax
        call    printf
        add     DWORD PTR [esp+16], 1
.L2:
        cmp     DWORD PTR [esp+16], 9
        jle     .L3
```

```
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

Optimización: -O1

Para este código ensamblador si se produce optimización, y se reducen las operaciones que se encuentran dentro del bucle, acortando de manera drástica las operaciones que se encuentran en dentro de la etiqueta .L2. Además se hace uso de "printf_chk", que comprueba si hay desbordamiento en la pila, antes de calcular el resultado.

```
        .file   "3.c"
        .intel_syntax noprefix
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Resultado = %d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB22:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        push    esi
        push    ebx
        and     esp, -16
        sub     esp, 16
        mov     esi, 1
        .cfi_offset 3, -16
        .cfi_offset 6, -12
        mov     ebx, 0
.L2:
        add     esi, ebx
        add     esi, esi
        mov     DWORD PTR [esp+8], esi
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        add     ebx, 1
        cmp     ebx, 10
        jne     .L2
        lea     esp, [ebp-8]
        pop     ebx
        .cfi_restore 3
        pop     esi
        .cfi_restore 6
        pop     ebp
        .cfi_def_cfa 4, 4
        .cfi_restore 5
        ret
        .cfi_endproc
.LFE22:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

Optimización: -O2

En este código se produce una optimización similar a la que se produce con -O1, pero se optimiza aún más el código, produciéndose una reestructuración de la posición de algunas de las instrucciones. Esto va a producir una mejora en el rendimiento.

```
        .file   "3.c"
        .intel_syntax noprefix
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Resultado = %d\n"
        .section        .text.startup,"ax",@progbits
        .p2align 4,,15
        .globl  main
        .type   main, @function
main:
.LFB22:
        .cfi_startproc
        push    ebp
        .cfi def cfa offset 8
        .cfi offset 5, -8
        mov     ebp, esp
        .cfi def cfa_register 5
        push    esi
        mov     esi, 1
        .cfi offset 6, -12
        push    ebx
        xor     ebx, ebx
        .cfi offset 3, -16
        and     esp, -16
        sub     esp, 16
        .p2align 4,,7
        .p2align 3
.L2:
        add     esi, ebx
        add     ebx, 1
        add     esi, esi
        mov     DWORD PTR [esp+8], esi
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        cmp     ebx, 10
        jne     .L2
        lea     esp, [ebp-8]
        pop     ebx
        .cfi_restore 3
        pop     esi
        .cfi_restore 6
        pop     ebp
        .cfi def cfa 4, 4
        .cfi_restore 5
        ret
        .cfi_endproc
.LFE22:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

Optimización: -O3

Para el último código ensamblador generado se produce una optimización, que presenta una gran variación con respecto a -O2, en la que se puede ver el código con el bucle desenrollado, y no hace las mismas llamadas de saltos condicionales. Esto va a significar que se reduzca el tiempo de ejecución.

```
        .file   "3.c"
        .intel syntax noprefix
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
```

```
        .string "Resultado = %d\n"
        .section        .text.startup,"ax",@progbits
        .p2align 4,,15
        .globl  main
        .type   main, @function
main:
.LFB22:
        .cfi_startproc
        push    ebp
        .cfi def cfa offset 8
        .cfi offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp+8], 2
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        mov     DWORD PTR [esp+8], 6
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call      printf chk
        mov     DWORD PTR [esp+8], 16
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call      printf chk
        mov     DWORD PTR [esp+8], 38
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        mov     DWORD PTR [esp+8], 84
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        mov     DWORD PTR [esp+8], 178
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call      printf chk
        mov     DWORD PTR [esp+8], 368
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        mov     DWORD PTR [esp+8], 750
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        mov     DWORD PTR [esp+8], 1516
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        mov     DWORD PTR [esp+8], 3050
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], 1
        call    __printf_chk
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE22:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

4.Diferencias entre -O1 y -funroll-loops.

En el último ejercicio vamos a compilar el código propuesto con la opción -O1 y luego con -O1 -funroll-loops (desenrollamiento de bucles), analizaremos los tiempos obtenidos de su ejecución y compraremos el código ensamblador de ambos.

Código en C

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#define N 100

double res[N];

main()
{
int i;
double x;
struct timeval inicio, final;
double tiempo;

gettimeofday(&inicio,NULL);
for(i=0;i<N;i++)
      res[i]=0.0005*i;
for(i=0;i<N;i++){
      x=res[i];
      if(x<10.0e6) x=x*x+0.0005;
      else x=x-1000;
      res[i]+=x;
}
gettimeofday(&final,NULL);
printf("resultado=%e\n",res[N-1]);
printf("%d\n",i);
printf("%f\n",x);


tiempo=(final.tv_sec-inicio.tv_sec+(final.tv_usec-
inicio.tv_usec)/1.e6);

printf("\nEl tiempo es: %f\n",tiempo);
}
```

Para empezar vamos a explicar las diferencias de ambos códigos ensamblador, con un fragmento de cada código, perteneciente a la etiqueta .L2, en ambos fragmentos, no se incluye el gettimeofday utilizado posteriormente para medición de tiempos:

Optimización: -O1

Con la optimización -O1, se consigue un código ensamblador corto, pero se pierde rendimiento al tener instrucciones mayor número de instrucciones dedicadas a la gestión del bucle frente a las que realizan tareas para el cálculo de resultado, esta es la función que va a realizar el primer bucle al volver a ejecutarse en cada interacción.

```
.L2:
        mov     DWORD PTR [esp+28], eax
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+eax*8]
        add     eax, 1
        cmp     eax, 10000
        jne     .L2
        fstp    st(0)
        mov     ax, 0
        fld     DWORD PTR .LC1
```

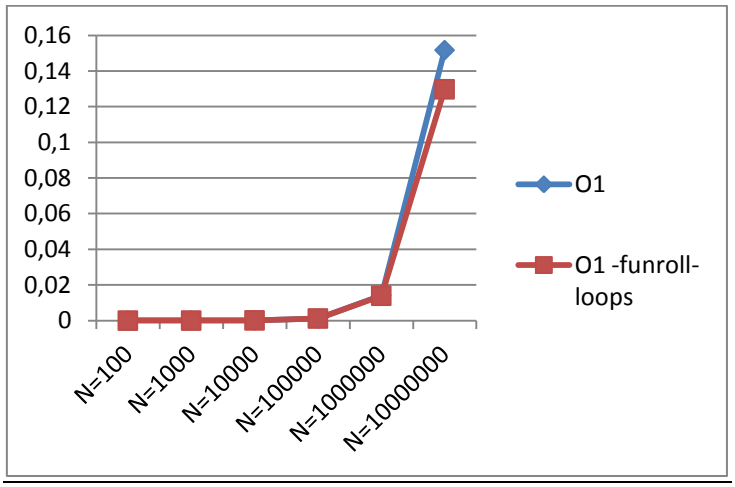
Optimización: -O1 -funroll-loops

Con la optimización -O1 -funroll-loops se obtiene un código ensamblador más largo, ya que el bucle se desenrolla. A cambio, se aumenta el número de instrucciones.

```
.L2:
        mov     DWORD PTR [esp+28], eax
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+eax*8]
        add     eax, 1
        mov     DWORD PTR [esp+28], eax
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+eax*8]
        lea     ecx, [eax+1]
        mov     DWORD PTR [esp+28], ecx
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+ecx*8]
        lea     edx, [eax+2]
        mov     DWORD PTR [esp+28], edx
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+edx*8]
        lea     ecx, [eax+3]
        mov     DWORD PTR [esp+28], ecx
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+ecx*8]
        lea     edx, [eax+4]
        mov     DWORD PTR [esp+28], edx
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+edx*8]
        lea     ecx, [eax+5]
        mov     DWORD PTR [esp+28], ecx
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+ecx*8]
        lea     edx, [eax+6]
        mov     DWORD PTR [esp+28], edx
        fild    DWORD PTR [esp+28]
        fmul    st, st(1)
        fstp    QWORD PTR res[0+edx*8]
        add     eax, 7
        cmp     eax, 10000
        jne     .L2
        fstp    st(0)
        mov     ax, 0
        fld     DWORD PTR .LC1
```

Una vez explicados brevemente los fragmentos de los códigos ensamblador para las diferentes opciones -O1 y -O1 -funroll-loops, procederemos a mostrar y analizar los resultados obtenidos de los tiempos de ejecución:

	-O1	-O1 -funroll-loops
N=100	0,000018	0,000016
N=1000	0,00005	0,000027
N=10000	0,00016	0,000135
N=100000	0,00123	0,001196
N=1000000	0,014094	0,013973
N=10000000	0,151714	0,129771



Como se puede ver en la tabla y gráfica de los tiempos de ejecución, al principio cuando N el número de iteraciones es muy pequeño, los tiempos son muy parejos y a medida que incrementamos el valor de N la diferencia entre ambas opciones de ejecución va incrementando. Si siguiésemos aumentando el valor de N, la diferencia seguiría creciendo, demostrando que la opción de desenrollamiento de bucle es la que mejor rendimiento obtiene en tiempos de ejecución. Sin embargo, con esta opción, el tamaño del código ensamblador se incrementará.

Referencias

- Material docente de la asignatura
- https://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/Optimize-Options.html
- http://www.davidam.com/docu/gccintro.es.html