Compiladores e Intérpretes Práctica 2

4 - Paso de bucles dentro de una función

Introducción

Esta técnica se basa en que si tenemos un conjunto de bucles anidados que invocan una función para cada uno de sus elementos (lo que se suele considerar una operación map sobre un conjunto), como es el siguiente código:

```
void suma(float x, float y, float *z){
      *z = x + y;
}

int main(){
        int i, j;
        static float x[N], y[N], z[N];

        for(j=0; j<ITER; j++)
            for(i=0; i<N; i++){
                  suma(x[i], y[i], &z[i]);
            }

        return 0;
}
```

Entonces podemos conseguir mejoras en el rendimiento de nuestro código, si hacemos que los bucles pasen a formar parte de la función y en el cuerpo de nuestro programa solo invocamos la función. A continuación se muestra la optimización aplicada sobre el código anterior:

```
void suma2(float *x, float *y, float *z){
      register int i, j;
      for(j=0; j<ITER; j++)
            for(i=0; i<N; i++)
                  z[i] = x[i] + y[i];
}

int main(){
  static float x[N], y[N], z[N];

  suma2(x, y, z);

  return 0;
}
```

Propósito de la técnica

Al aplicar esta técnica, se persiguen las siguientes mejoras:

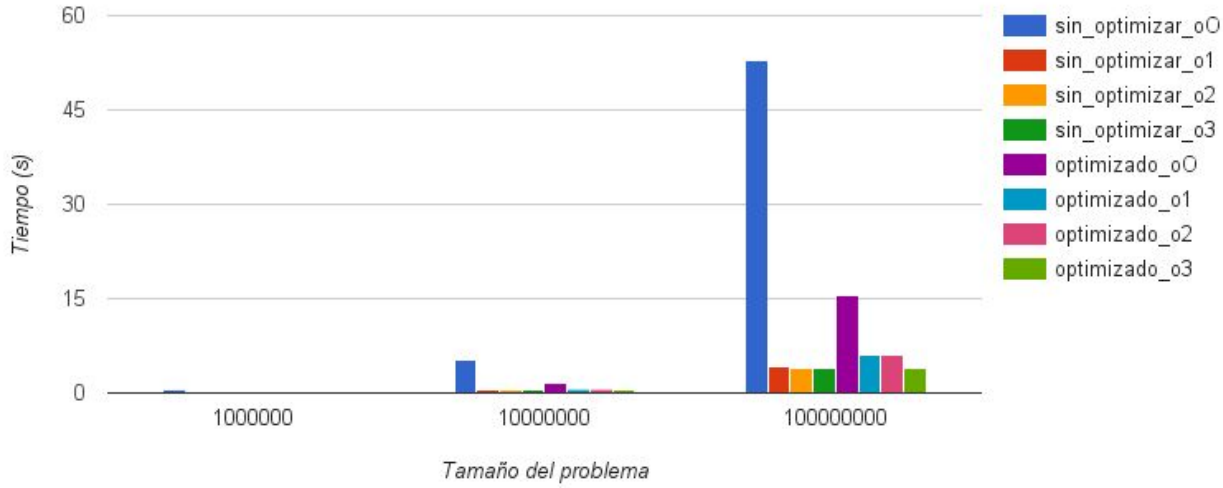
- **Reducir el número de llamadas a función.** Al utilizar la primera versión del código, en cada iteración de este estamos guardando el estado del bloque actual de nuestro programa, preparando los argumentos que espera la función, desplazando nuestro contador de instrucción actual, saltando a una nueva posición en la pila del sistema para gestionar la memoria utilizada por la función o subrutina, almacenando la dirección de retorno a la que tenemos que volver una vez que terminemos la función actual y restaurando el estado anterior de la pila una vez que abandonamos la función o subrutina.

  Si en vez de realizar **i · j** llamadas a función, simplemente realizamos una única llamada a función en la que realizamos todas las iteraciones necesarias sobre el conjunto de datos con el que estamos trabajando, las ventajas en cuanto a rendimiento son bastante claras.

Resultados obtenidos

A continuación, se muestran los resultados obtenidos de compilar el código anterior en su versión optimizada y sin optimizar, utilizando en ambos casos las diferentes opciones de optimización utilizadas por el compilador.

Para realizar la medición, en cuanto a tiempo de ejecución del código, se está midiendo todo el bloque de código mostrado anteriormente y se han añadido un par de instrucciones printf fuera del fragmento a medir para que el compilador no trate el código como código muerto.



	1000000	10000000	100000000
sin_optimizar_oO	0.5146	5.3144	52.8798
sin_optimizar_o1	0.0342	0.4093	4.0821
sin_optimizar_o2	0.0345	0.4161	4.0741
sin_optimizar_o3	0.0162	0.3956	3.8861
optimizado_oO	0.1515	1.5680	15.6067
optimizado_o1	0.0580	0.6214	6.1341
optimizado_o2	0.0578	0.6154	6.0888
optimizado_o3	0.0182	0.3966	3.9878

En los resultados obtenidos podemos observar como al aplicar la técnica que estamos estudiando sobre la solución sin optimizar, obtenemos mejoras de rendimiento de entorno al 70% para cualquier tamaño del problema estudiado, Si observamos el código ensamblador que se genera para la solución sin optimizar, podemos ver cómo se cumple la explicación dada en el apartado anterior.

/*Inicio del cuerpo del bucle for anidado*/

`.L5:`	
`cmpl $99999999, -64(%rbp)`	
`jg .L4`	
`movl -64(%rbp), %eax`	
`cltq`	
`salq $2, %rax`	

`leaq _ZZ4mainE1z(%rax), %rdx`	
`movl -64(%rbp), %eax`	
`cltq`	
`movss _ZZ4mainE1y(,%rax,4), %xmm0`	
`movl -64(%rbp), %eax`	
`cltq`	
`movl _ZZ4mainE1x(,%rax,4), %eax`	
`movq %rdx, %rdi`	Según gcc, este usa los registro %rdx e %rdi para preparar los argumentos que recibe una función[1], por lo que podemos intuir que aquí se está preparando los argumentos de la función suma
`movaps %xmm0, %xmm1`	
`movl %eax, -68(%rbp)`	
`movss -68(%rbp), %xmm0`	
`call _Z4sumaffPf`	salto a la función suma
`addl $1, -64(%rbp)`	
`jmp .L5`	

/* función suma*/

`_Z4sumaffPf:`	
`.LFB7:`	
`.cfi_startproc`	
`pushq %rbp`	preparación de la pila para almacenar los valores de la rutina actual
`.cfi_def_cfa_offset 16`	
`.cfi_offset 6, -16`	
`movq %rsp, %rbp`	se establece el valor de la base de la pila con el contenido de rsp

[1] https://www.cs.uaf.edu/2007/fall/cs301/support/x86_64/index.html
http://stackoverflow.com/questions/23367624/intel-64-rsi-and-rdi-registers

`.cfi_def_cfa_register 6`	
`movss %xmm0, -4(%rbp)`	lectura de los argumentos de la función
`movss %xmm1, -8(%rbp)`	lectura de los argumentos de la función
`movq %rdi, -16(%rbp)`	lectura de los argumentos de la función
`movss -4(%rbp), %xmm0`	
`addss -8(%rbp), %xmm0`	
`movq -16(%rbp), %rax`	
`movss %xmm0, (%rax)`	
`nop`	
`popq %rbp`	descartamos el punteroo a la base de la pila utilizada para los valores de esta función
`.cfi_def_cfa 7, 8`	
`ret`	sitúa en los registros necesarios, los valores para permitir volver al fragmento de código que provocó la ejecución de esta función
`.cfi_endproc`	termina la función actual

En el código ensamblador que se acaba de mostrar, se puede observar como cada vez que invocamos una función estamos ejecutando un número considerable de instrucciones en ensamblador, que podemos evitar al utilizar la técnica que se está estudiando.

Por otro lado, es muy curioso observar como las optimizaciones realizadas por el compilador con el flag -O1 y -O2 dan mejores resultados si las aplicamos sobre la versión sin optimizar, en lugar de aplicarlas sobre la versión optimizada. Aunque finalmente, si aplicamos el flag -O3, ambas versiones alcanzan prácticamente el mismo resultado generando código en ensamblador un tanto diferente, obteniendo una leve mejora la versión sin optimizar.

Esta pequeña ventaja que ofrecen las optimizaciones realizadas sobre la versión sin optimizar es posible que se deban a que el compilador le resulta más sencillo aplicar técnicas de optimización sobre un fragmento de código con menos ramificaciones en lugar de intentar optimizar directamente la función suma2 que únicamente contiene bucles for y la operación que aplica el bucle interno. En cualquiera caso, esta optimización es un caso bastante claro de que no todas las mejoras que apliquemos nosotros, o el compilador, al código tienen que producir mejores resultados.

Finalmente comentar que no se ha apreciado en ningún momento la influencia del tamaño del problema sobre el código generado.