Escola Técnica Superior de Enxeñaría

Compiladores e Intérpretes

Generación y optimización de código

Paso de bucles dentro de una función

5 de junio de 2022

Índice

Resumen

En el presente informe se realizará un estudio sobre una hipotética optimización de código compuesta principalmente por un bucle que se encarga de realizar una suma de dos vectores. En primer lugar se realizará una breve introducción sobre el método de optimización. Posteriormente se mostrará el proceso de optimización, mostrando el código original y el código que se desea optimizar. Una vez conocido el código y su optimización, se realizará la experimentación con el código. Por último se realizará un análisis de resultados obtenidos en la experimentación.

1. Introducción

El paso de bucles dentro de una función es una técnica de optimización que consiste en la inserción de un bucle dentro de una función de C, para posteriormente realizar una llamada a la función definida. Mediante esta técnica se pretende obtener unos tiempos de ejecución notablemente menores en comparación al código sin optimizar.

2. Proceso de optimización

Para la comparación que se pretende realizar, se tendrá el siguiente punto de partida:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>


int main(int argc, char *argv[]){
    int i, j;
    int N = atoi(argv[1]);
    int ITER = atoi(argv[2]);
    float *x, *y, *z, tiempo_transcurrido = 0.0, tiempo_total = 0.0;
    struct timeval inicio, final;

    x = (float *) malloc(N * sizeof(float));
    y = (float *) malloc(N * sizeof(float));
    z = (float *) malloc(N * sizeof(float));

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

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

    for(j = 0; j < ITER; j++){
        gettimeofday(&inicio, NULL);
            for(i = 0; i < N; i++){
                suma(x[i], y[i], &z[i]);
            }
        gettimeofday(&final, NULL);
        tiempo_transcurrido = (final.tv_sec - inicio.tv_sec) + 1e-6*(final.
    tv_usec - inicio.tv_usec);
        tiempo_total += tiempo_transcurrido;
    }
    printf("%0.8f\n", tiempo_total / ITER);
```

```
    free(x);
    free(y);
    free(z);
    return 0;
}
```

Listing 1: Código no optimizado

En este código, se puede observar en la función `main` que existe un bucle `for` que se encarga de recorrer los bucles para pasarle el valor de cada una de las posiciones del bucle a la función de suma. La función de suma se encarga de realizar la operación y almacenarla en la posición del vector solución correspondiente. Por tanto, tal y como se puede apreciar en el código, será objeto de medida de tiempos tanto el bucle que se encarga de recorrer los bucles como la ejecución de la función de suma.

Para la posible optimización del código, se tendrá el siguiente punto de partida:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(int argc, char *argv[]){
    int i, j;
    int N = atoi(argv[1]);
    int ITER = atoi(argv[2]);
    float *x, *y, *z, tiempo_transcurrido = 0.0, tiempo_total = 0.0;
    struct timeval inicio, final;

    x = (float *) malloc(N * sizeof(float));
    y = (float *) malloc(N * sizeof(float));
    z = (float *) malloc(N * sizeof(float));

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

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

    for(j = 0; j < ITER; j++){
        gettimeofday(&inicio, NULL);
        suma2(x, y, z);
        gettimeofday(&final, NULL);
        tiempo_transcurrido = (final.tv_sec - inicio.tv_sec) + 1e-6*(final.
    tv_usec - inicio.tv_usec);
        tiempo_total += tiempo_transcurrido;
    }
    printf("%0.8f\n", tiempo_total / ITER);
    free(x);
    free(y);
    free(z);
    return 0;
}
```

Listing 2: Código optimizado

En este segundo código, se puede observar que la función suma (`suma2`) es la encargada de realizar todo el proceso del recorrido del bucle y la realización de la operación de suma

correspondiente. Esto es, a la función de suma se le pasarán como parámetros los vectores operandos junto con el vector solución, por lo que para una posición determinada, se obtendrá de los vectores operandos el valor correspondiente y se almacenará el cómputo de la suma en la posición actual del bucle. En ambos programas se realizará una reserva dinámica de memoria debido a que cuando se realizan reservas de memoria con **N** notablemente grande, se llenaría el *stack* de memoria y se produciría un fallo de segmento.

3. Experimentación

Ahora se realizarán las pruebas con respecto a la posible optimización que ofrece el segundo código con respecto al anterior. Para ello, se ejecutarán ambos códigos, obteniéndose resultados en función de tiempos en función de los tamaños de los vectores. Este tamaño estará determinado por **N**. Como hipótesis de partida se tiene que, cuanto mayor sea el tamaño de N, mayor será el tiempo de ejecución del código, pues se realizan un número de operaciones notablemente mayor. Para obtener los tiempos de ejecución, se ha calculado la media de tiempos de múltiples ejecuciones. Esto es, la constante **ITER** permite obtener múltiples tiempos para un mismo N. Teniendo varias mediciones para un mismo tamaño de vectores, se podrá realizar la media para obtener unos resultados más certeros. Cabe destacar que el número ITER variará en función del tamaño de N, pues tendrán una relación inversamente proporcional (cuanto mayor sea el tamaño de N, menor será ITER, y viceversa). Esto es debido a que cuando aumenta N, aumenta el tiempo de ejecución, por lo que el error que se pueda obtener será cada vez menos significativo. Por tanto, se obtienen las siguientes medidas medias:

N	Tiempo original (s)	Tiempo optimizado (s)
400 000 000	1.26	0.78
600 000 000	1.90	1.19
800 000 000	2.61	1.61
1 000 000 000	3.54	2.22
1 200 000 000	4.65	2.93

Tabla 1: Tabla comparativa entre los tiempos de ejecución de los programas en función de N

Representando estos valores de tiempos en función de N, se obtendrá la siguiente gráfica:

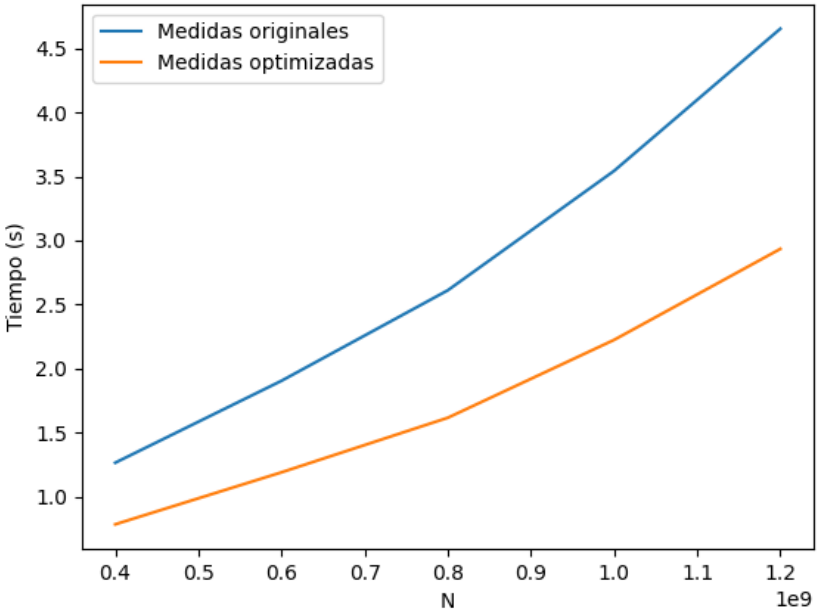

Figura 1: Comparativa de los tiempos de ejecución

Ahora se realizará un análisis de más bajo nivel para conocer las operaciones que se realizan a nivel ensamblador. Para ello, se realizará uso de la herramienta Compiler Explorer para poder visualizar el código en C a código ensamblador:

```
suma:                                   // 2-ETIQUETA SUMA
        push    rbp
        mov     rbp, rsp
        movss   DWORD PTR [rbp-4], xmm0 // Se obtienen los registros
        movss   DWORD PTR [rbp-8], xmm1 // Se obtienen los registros
        mov     QWORD PTR [rbp-16], rdi
        movss   xmm0, DWORD PTR [rbp-4]
        addss   xmm0, DWORD PTR [rbp-8] // Se realiza la suma
        mov     rax, QWORD PTR [rbp-16]
        movss   DWORD PTR [rax], xmm0   // Se almacena el resultado de la suma
        nop
        pop     rbp
        ret
.L7:                                    // 1-ETIQUETA DEL MAIN DEL PROGRAMA
        mov     eax, DWORD PTR [rbp-4]  // Carga del resultado &z[i] a suma2.0
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-48]
        add     rdx, rax
        mov     eax, DWORD PTR [rbp-4]  // Carga del operando y[i] a suma2.0
        cdqe
        lea     rcx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-40]
        add     rax, rcx
        movss   xmm0, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-4]  // Carga del resultado x[i] a suma2.0
        cdqe
        lea     rcx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-32]
        add     rax, rcx
```

```
        mov     eax, DWORD PTR [rax]
        mov     rdi, rdx
        movaps  xmm1, xmm0
        movd    xmm0, eax
        call    suma                    // Salto a suma
        add     DWORD PTR [rbp-4], 1    // Incremento de i
.L6:
        mov     eax, DWORD PTR [rbp-4]
        cmp     eax, DWORD PTR [rbp-16] // Si es menor que N
        jl      .L7                     // Se realiza un salto a L7
        lea     rax, [rbp-80]
        mov     esi, 0
        mov     rdi, rax
        call    gettimeofday
```

Listing 3: Código ensamblador original

Por otro lado, el código en ensamblador del código optimizado será:

```
suma2.0:                                // 2-ETIQUETA SUMA
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi  // Obtención de valores de registros
        mov     QWORD PTR [rbp-16], rsi
        mov     QWORD PTR [rbp-24], rdx
        mov     rax, r10
        mov     QWORD PTR [rbp-32], r10
        mov     edx, 0
        mov     DWORD PTR [rax+4], edx
        jmp     .L2                     // Salto a la etiqueta L2
.L3:                                    // 4-ETIQUETA L3
        mov     edx, DWORD PTR [rax+4]  // Obtención del primer sumando x[i]
        movsx   rdx, edx
        lea     rcx, [0+rdx*4]
        mov     rdx, QWORD PTR [rbp-8]
        add     rdx, rcx
        movss   xmm1, DWORD PTR [rdx]
        mov     edx, DWORD PTR [rax+4]  // Obtención del segundo sumando y[i]
        movsx   rdx, edx
        lea     rcx, [0+rdx*4]
        mov     rdx, QWORD PTR [rbp-16]
        add     rdx, rcx
        movss   xmm0, DWORD PTR [rdx]
        mov     edx, DWORD PTR [rax+4]  // Almacenamiento del resultado
        movsx   rdx, edx
        lea     rcx, [0+rdx*4]
        mov     rdx, QWORD PTR [rbp-24]
        add     rdx, rcx
        addss   xmm0, xmm1              // Suma de los operandos
        movss   DWORD PTR [rdx], xmm0   // Almacenamiento en Z[i]
        mov     edx, DWORD PTR [rax+4]
        add     edx, 1                  // Incremento de i
        mov     DWORD PTR [rax+4], edx
.L2:                                    // 3-ETIQUETA L2
        mov     ecx, DWORD PTR [rax+4]
        mov     edx, DWORD PTR [rax]
        cmp     ecx, edx                // Comparación de condición del FOR
        jl      .L3                     // Si es menor, salta a L3
        nop
        nop
        pop     rbp                     // Se extrae de la pila su tope
        ret
.L8:                                    // 1-ETIQUETA DEL MAIN DEL PROGRAMA
```

```
45          lea     rax, [rbp-80]
46          mov     esi, 0
47          mov     rdi, rax
48          call    gettimeofday            // Primera llamada a gettimeofday
49          mov     rdx, QWORD PTR [rbp-40] // Almacenar los arrays en registros
50          mov     rcx, QWORD PTR [rbp-32] // Almacenar los arrays en registros
51          mov     rax, QWORD PTR [rbp-24] // Almacenar los arrays en registros
52          lea     rsi, [rbp-64]           // Carga de dirección efectiva
53          mov     r10, rsi
54          mov     rsi, rcx
55          mov     rdi, rax
56          call    suma2.0                 // Salto a suma2.0
57          lea     rax, [rbp-96]
58          mov     esi, 0
59          mov     rdi, rax
60          call    gettimeofday            // Segunda llamada a gettimeofday
61
```

Listing 4: Código ensamblador optimizado

4. Análisis de resultados

A la vista de los resultados obtenidos tras la ejecución del proceso de experimentación, se han obtenido unos resultados notablemente positivos con las optimizaciones de código aplicadas sobre la versión original del mismo.

En primer lugar, se puede comprobar que según las medidas obtenidas en la Tabla 2. Se realizará un cálculo porcentual acerca de la mejora en cuanto a los tiempos obtenidos tras la ejecución de cada uno de los códigos (la mejora del código optimizado con respecto al código original). Para ello, se empleará sistemáticamente la siguiente fórmula en el correspondiente apartado:

$$Porcentaje = \left(1 - \frac{T_{Optimizado}}{T_{Original}}\right) \cdot 100$$

Se obtiene:

N	% Optimización
400 000 000	38.09 %
600 000 000	37.37 %
800 000 000	38.31 %
1 000 000 000	37.39 %
1 200 000 000	36.99 %

Tabla 2: Mejora porcentual de la optimización con respecto a la versión original

Se puede comprobar que la mejora mantiene una proporción aproximada (con una desviación de un `1.32%` entre el porcentaje más pequeño y el más grande) con respecto al tamaño de N (a medida que el tamaño de N crece, se mantiene de un modo aproximado el porcentaje optimizado). Esta proporción se puede ver reflejada en la gráfica 1, donde se aprecia que el desfase entre los tiempos de ejecución entre los códigos no se mantiene constante, sino que crece en función de N.

La respuesta a la posible mejora se puede encontrar en el código ensamblador del código asociado. A través de este, se pueden observar todas las operaciones más relevantes que toma el compilador para obtener la suma de los dos vectores. Como principal característica relevante, se puede observar que en el primer código (original), en la etiqueta **suma**, se realizan dos operaciones que conciernen al *stack* (`push rbp` en la *línea 2* y `pop rbp` en la *línea 12*). Por tanto, en cada una de las **N** iteraciones se realizan un total de `2 * N` operaciones de entrada/salida con la pila. Por otro lado, en la versión optimizada del código, se realiza una única operación de *push* a la pila en la *línea 2*, realizándose el *pop* una vez finalizado el bucle.

Esta diferencia de las operaciones se puede justificar por el aprovechamiento de la localidad que se realiza cuando se pasa el bucle a la función suma: cuando se pasan únicamente los valores de los vectores para realizar las operaciones (en la versión original), se realizan estas `2 * N` operaciones en la pila, pues una vez que se termina de operar con los valores, estos se extraen de la pila. Sin embargo, cuando se pasan los vectores como parámetros de la función, se realiza un aprovechamiento de la localidad, por lo que en la pila se realiza un único *push* en cada iteración.

Otro punto clave y que resulta más definitivo es que en la versión original, se realiza una llamada a una función en cada una de las **N** iteraciones. En la versión optimizada, se realiza una única llamada a la función y el bucle se ejecuta dentro de la propia función. Por tanto, el hecho de que se realice una única llamada a la función de suma frente a las **N** llamadas que se requerirían en la versión original, hace que se optimicen notablemente los tiempos de ejecución del código optimizado frente al original.

Referencias

[1] *Compiladores. Principios, técnicas y herramientas* (2ª edición): Aho, Alfred; Sethi, Ravi; Ullman, Jeffrey D.; Lam, Monica S