COMPILADORES E INTÉRPRETES

GENERACIÓN Y OPTIMIZACIÓN DE CÓDIGO

Índice

TÉCNICA DE OPTIMIZACIÓN A TRATAR

En este caso, se trabajará sobre la técnica de optimización de lazos conocida como ***intercambio de lazos***. Como se ha dicho en clase, existe un principio que asegura que se utiliza el 90% del tiempo de ejecución de un programa en ejecutar el 10% del código de este, lo cual se debe precisamente a los bucles y la reutilización del código. Con esta técnica, por lo tanto, se pretende atacar precisamente a este cuello de botella que pueden presentar los bucles en el tiempo de ejecución de nuestros programas.

El intercambio de lazos consiste en intercambiar el orden de dos variables de iteración, es decir, cambiar el orden en el que se ejecutan dos bucles anidados. De esta forma, se pretende mejorar el rendimiento de la caché al acceder a los elementos de un array, que como bien sabemos se almacenan de forma consecutiva en C, lenguaje de programación que utilizaremos para implementar esta optimización.

Concretamente, en esta práctica trabajaremos sobre matrices, por lo que lo que nos interesará será recorrer la matriz en el mismo orden en el que esta se almacena en memoria, de forma que se aproveche la localidad espacial de las referencias que utilice nuestro programa. Así, reduciremos los fallos caché en favor del tiempo de ejecución, que se debería ver reducido al aplicar esta técnica correctamente como consecuencia.

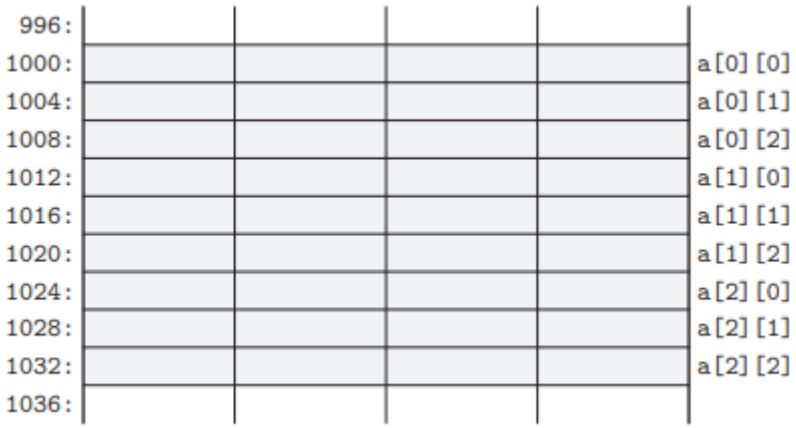


Figura 1

Como se puede ver en la *Figura 1*, cuando se declara un vector multidimensional en C el compilador reserva tantas posiciones contiguas como valores contenga el vector. En este caso, para una matriz 3x3 de enteros reservará nueve celdas de 4 bytes que como vemos se ordenan por **filas**, no por columnas. Teniendo esto en cuenta, en nuestro caso concreto nos interesará acceder a los elementos de la matriz por filas, ya que de esa forma podremos reducir los fallos de la memoria caché al acceder a cada elemento, algo que no ocurriría si accedemos por columnas, ya que lo más probable es que los elementos no se encuentren en la misma línea caché.

BENEFICIOS Y DESVENTAJAS DE LA TÉCNICA

Como hemos visto, el propósito de esta técnica reside en **explotar la localidad** de referencia, bien la espacial, que sería la tendencia que tiene un proceso para referenciar direcciones de memoria virtual cercanas a la última referencia, o bien la temporal, que se correspondería con la tendencia a referenciar en el futuro elementos que fueron utilizados en un pasado reciente. Como consecuencia de esto, tendríamos una **mejora del acceso a memoria**, ya que estaríamos utilizando más eficientemente las líneas caché.

Por otro lado, puede haber alguna pequeña desventaja, como dejar el bucle más corto en el interior del anidamiento, algo que podría resultar inadecuado en ciertos softwares. Obviamente con todo esto estamos suponiendo una correcta aplicación de esta técnica, pero debemos tener mucho cuidado y estar muy seguros al aplicarla, ya que si la aplicamos cuando no debemos o de manera inadecuada estaremos obteniendo el efecto contrario, aumentando los fallos caché de nuestro código y empeorando ampliamente los tiempos de ejecución de nuestros programas.

EVALUACIÓN DE LOS RESULTADOS

Una vez se ha presentado la técnica de optimización sobre la que vamos a trabajar ha llegado el momento de ponerla a prueba y comprobar si realmente aporta los beneficios que se esperan de ella. Para ello, primero comentaremos el código que se ha empleado para la realización de nuestras pruebas, en el cual se realizará la ejecución de una serie de bucles con los que se pretende ver el resultado de aplicar o no aplicar la optimización.

A continuación, analizaremos tanto el hardware como el software que utilizaremos para realizar las pruebas e intentaremos obtener una primera estimación puramente teórica de los resultados en base a ellos. Finalmente, realizaremos las pruebas y extraeremos una serie de datos que nos permitirán realizar una última comparación, esta vez basándonos ya en datos reales, sobre las bondades de aplicar el intercambio de lazos.

CÓDIGO DE LAS PRUEBAS

Si observamos el código C que se ha construido para la implementación de la técnica de optimización sobre la que trata este informe (Anexo), podemos ver que es muy simple, ya que, en él, únicamente se realizan **tres** acciones importantes. La primera de ellas es el **calentamiento de la memoria caché**, con el que se pretende evitar el efecto de las cargas iniciales de los índices de los bucles en la memoria caché, que haría que la ejecución de la primera versión de nuestro código de pruebas se viese penalizada frente a la segunda versión al no haberse cargado todavía estos valores en esta memoria. Para evitar esto, simplemente se ejecuta primero nuestro lazo sin optimizar dos veces, realizándose las mediciones en su segunda ejecución.

Las otras dos acciones que restan, como cabe esperar, se corresponden con las **ejecuciones** de nuestro código sin optimizar y nuestro código optimizado, respectivamente. En ellas, lo que se hace es realizar ***MEAN*** mediciones, que nos permitirán posteriormente extraer un valor medio en lugar de obtener nuestros resultados de una única ejecución de cada porción de código. Por lo demás, el único detalle destacable respecto a la implementación que se proporcionaba en el guion está en la declaración de las matrices, que en este caso se realiza en memoria dinámica en lugar de en memoria estática para evitar problemas cuando el tamaño de las matrices es excesivo, pero esto no afectará a nuestros resultados.

Por otro lado, en lo que respecta al código ensamblador generado a partir del código fuente que acabamos de mencionar, se ha obtenido con la herramienta https://godbolt.org/ el código correspondiente a los bucles sobre los que vamos a trabajar, de forma que se pueda ver más gráficamente a qué parte corresponde cada línea de código ensamblador. Este código se incluye también en el Anexo en dos imágenes, una primera imagen en la que se ven las líneas asociadas a nuestra versión sin optimizar, y una segunda imagen en la que podemos ver nuestra versión optimizada.

Si nos fijamos en ambas figuras, el código que muestran es prácticamente idéntico, ya que la única diferencia se encuentra en el intercambio de los dos bucles, que se muestra en la *Figura 2*. Esto se debe a que la optimización que estamos implementando no afecta al código generado en sí, sino al aprovechamiento de la localidad espacial en el acceso a la información, por lo que no se apreciarán diferencias significativas más allá de esta que acabamos de comentar.

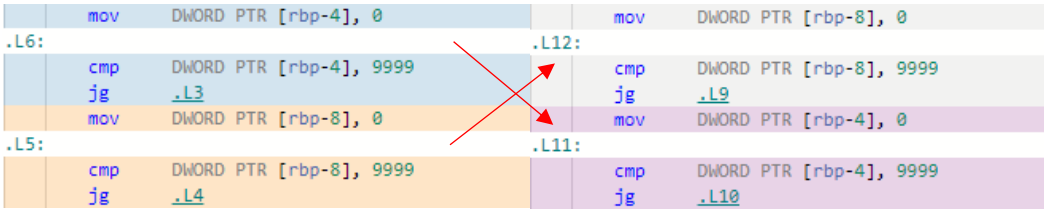


Figura 2

HARDWARE Y SOFTWARE DE LAS PRUEBAS

Antes de pasar a la realización de las pruebas, deberemos tener claros los elementos hardware y software de la máquina sobre la que se van a realizar, que se detallarán a continuación. En cuanto al hardware, disponemos de los siguientes elementos principales:

- Procesador ***Intel Core i7-7700k*** *(4x4,20GHz)*
  - Memoria L1 ***64KB*** por núcleo (*32KB* para datos y *32KB* para instrucciones)
  - Memoria caché L2 ***256KB*** por núcleo
  - Memoria caché L3 ***8MB***
- Memoria RAM de ***16GB***

De este hardware que acabamos de mencionar, lo que más nos interesa es la caché, cuya información se muestra en profundidad en la *Figura 3*. Así, además de los tamaños que se acaban de proporcionar arriba, podremos obtener información más detallada acerca de esta. Concretamente, estamos ante una caché **asociativa por conjuntos**, de forma que tenemos una memoria caché L1 de **8 vías**, una memoria caché L2 de **8 vías** y una memoria caché L3 de **16 vías**, todas ellas con un tamaño de línea de **64 bytes**.

Por otra parte, centrándonos ahora en el software estaremos trabajando sobre un ***Windows 10*** **de 64 bits** en el que se correrá una instancia de ***WSL*** en su versión *2* (*Windows Subsystem for Linux*), que será la que nos permita ejecutar un entorno Ubuntu, concretamente ***Ubuntu 20.04.3 LTS***. Para la compilación, utilizaremos la herramienta ***gcc*** en su versión ***9.3***. Por último, me gustaría destacar que, dada la utilización de *WSL* para emular Linux (no disponía de otra posibilidad actualmente), no

ha sido posible utilizar la herramienta ***PAPI*** para evaluar los fallos caché, algo que intenté hacer porque lo consideraba muy interesante y enriquecedor para el análisis de resultados, ya que WSL no facilita el acceso a los contadores hardware necesarios para que la herramienta funcione.

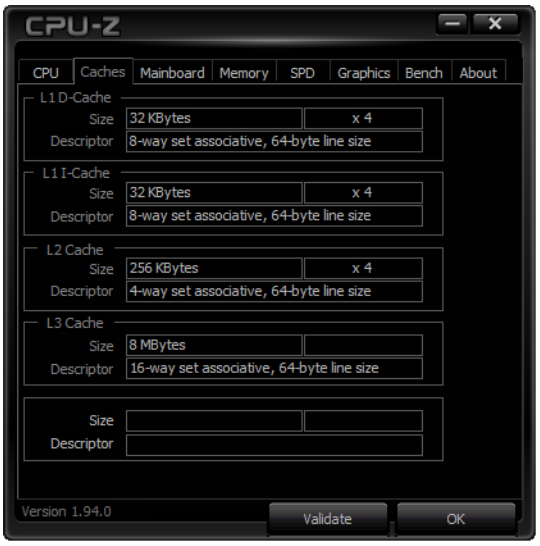


Figura 3

ESTIMACIÓN DE LOS RESULTADOS ESPERADOS

Una vez conocemos las capacidades hardware y software de la máquina sobre la que vamos a realizar la ejecución de nuestro código, podemos realizar una primera estimación teórica acerca de lo que va a suceder al aplicar el intercambio de bucles, de forma que podamos posteriormente comparar y verificar los resultados que obtengamos.

Lo que más nos importa en este caso es el tamaño de la línea caché, que será el que nos limite el número de elementos del array que podemos traernos a memoria con cada fallo. En este caso, disponemos de un tamaño de línea de **64 bytes**, por lo que, como estamos trabajando con *floats*, que ocupan **4 bytes**, podremos traernos en cada línea **16** elementos de nuestra matriz. Ahora se nos plantean dos escenarios:

- En el **código sin optimizar**, estaremos obteniendo **un fallo caché con cada lectura** de nuestras matrices, ya que con los tamaños de matriz sobre los que trabajaremos nunca traeremos en una lectura de memoria principal dos elementos pertenecientes a la misma columna.
- En el **código optimizado**, tendremos una tasa de fallos caché de **1/16**, ya que al estar accediendo a los elementos de la matriz de forma secuencial nos traeremos con cada fallo caché dieciséis elementos que se referenciarán a continuación y que no supondrán un fallo caché.

Teniendo esto en cuenta tendría sentido afirmar que, si el coste computacional únicamente dependiera de las lecturas a memoria principal, podríamos obtener diferencias enormes entre la versión sin optimizar y la versión mejorada, que ofrecería un rendimiento incluso **16** veces más rápido. Obviamente esto no es así y el tiempo de ejecución no dependerá exclusivamente de esto, pero a medida que crezca el tamaño de nuestras matrices cada vez serán más importantes y evidentes estos fallos caché, lo cual esperamos corroborar a continuación, aunque nunca nos acercaremos a este límite teórico.

DISCUSIÓN DE LOS RESULTADOS OBTENIDOS

Una vez llegados a este punto, solamente resta ejecutar nuestro código y comprobar los resultados que nos ofrezca. Para ello, variaremos el valor de la constante ***N***, que será la que defina las dimensiones de nuestras matrices. Concretamente, esta variable adoptará los valores: *100*, *1000*, *5000*, *10000*, *20000*, *30000* y *40000*. La razón de esta decisión está en la viabilidad de las pruebas, ya que valores más altos presentaban problemas a la hora de reservar la memoria (tengamos en cuenta que las matrices tienen un tamaño *NxN*).

Respecto a las mediciones que se realizan en nuestro código, obtendremos el coste por iteración (*ITER*) de cada una de las dos versiones, a partir del cual calcularemos el factor de mejora que presenta la aplicación del intercambio de bucles. Todos estos resultados, para cada valor de *N*, se muestran a continuación en la *Tabla 1*. Además, se han representado gráficamente estos resultados en una gráfica que presenta, por un lado, las diferencias en cuanto a tiempo de ejecución respecta y, por otro lado, el crecimiento de la mejora que ofrece la aplicación de la técnica.

N	S/Optimización	C/Optimización	Mejora
100	0,000022	0,000021	x1,05
1000	0,002958	0,001541	x1,92
5000	0,076298	0,038155	x2,00
10000	0,291332	0,152185	x1,91
20000	1,355056	0,610867	x2,22
30000	4,271471	1,372733	x3,11
40000	12,902825	2,443936	x5,28

Tabla 1

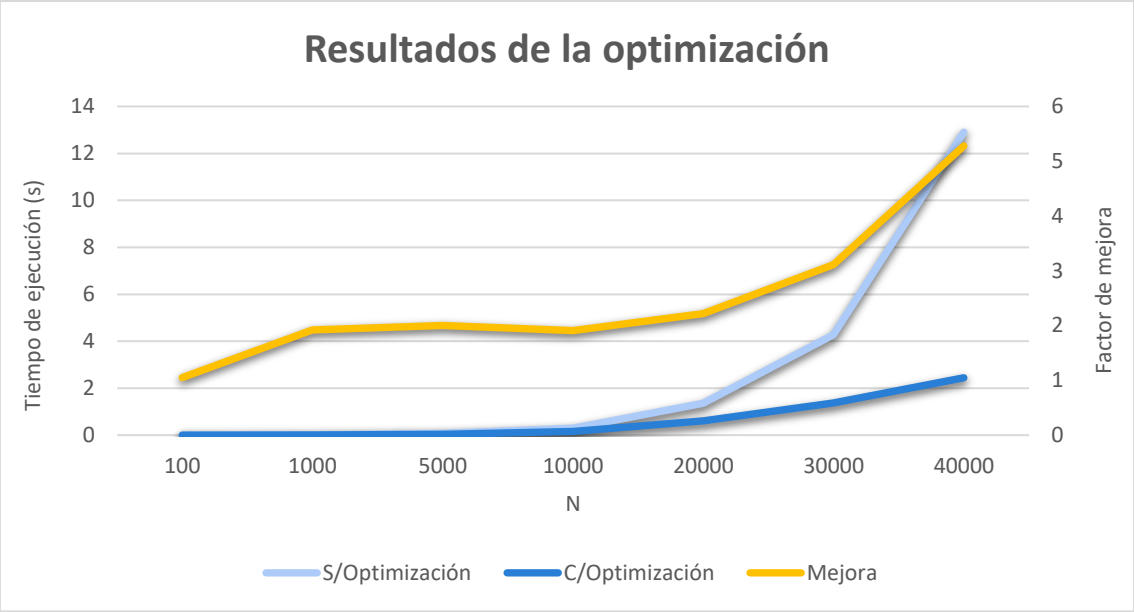


Figura 4

Una vez presentados los resultados, podemos ver cómo se puede verificar nuestra estimación, observándose un claro aumento en el factor de mejora entre ambas versiones a medida que el tamaño de las matrices aumenta. Es cierto que no estamos cerca de los valores que habíamos dicho, lo cual me parece totalmente comprensible, pero la mejora es notable y evidente, con una diferencia en lo que a tiempo de ejecución respecta de más de diez segundos cuando las matrices alcanzan una dimensión de *40.000 x 40.000*, diferencia más que significativa si tenemos en cuenta que únicamente estamos realizando una operación de multiplicación entre una matriz y un escalar cuyo resultado se almacena en otra matriz.

CONCLUSIONES

Antes de terminar este informe, me gustaría dar brevemente mi opinión al respecto. Tengo que decir que me han parecido muy sorprendentes las grandes diferencias que presenta la aplicación o la no aplicación del intercambio de lazos. Además, es algo en lo que nunca había caído durante todo este tiempo mientras realizaba mis programas, lo cual, en mi caso, no me ha afectado negativamente porque las matrices en C se almacenan por filas, pero esto es algo que no ocurre en todos los lenguajes de programación, por lo que es necesario tenerlo siempre presente. En Fortran, por ejemplo, las matrices se almacenan en memoria por **columnas**, al contrario que en C, por lo que si las recorriéramos utilizando la metodología que se nos ha enseñado siempre (por filas), que es la que suele aparecer en los pseudocódigos, estaríamos generando un código muy poco eficiente.

Para concluir, he de destacar nuevamente la utilidad de esta práctica de la asignatura para ser conscientes de estas pequeñas optimizaciones que dan lugar a un código de gran calidad y rendimiento, que es lo que nos interesa siempre que codificamos nuestros programas.

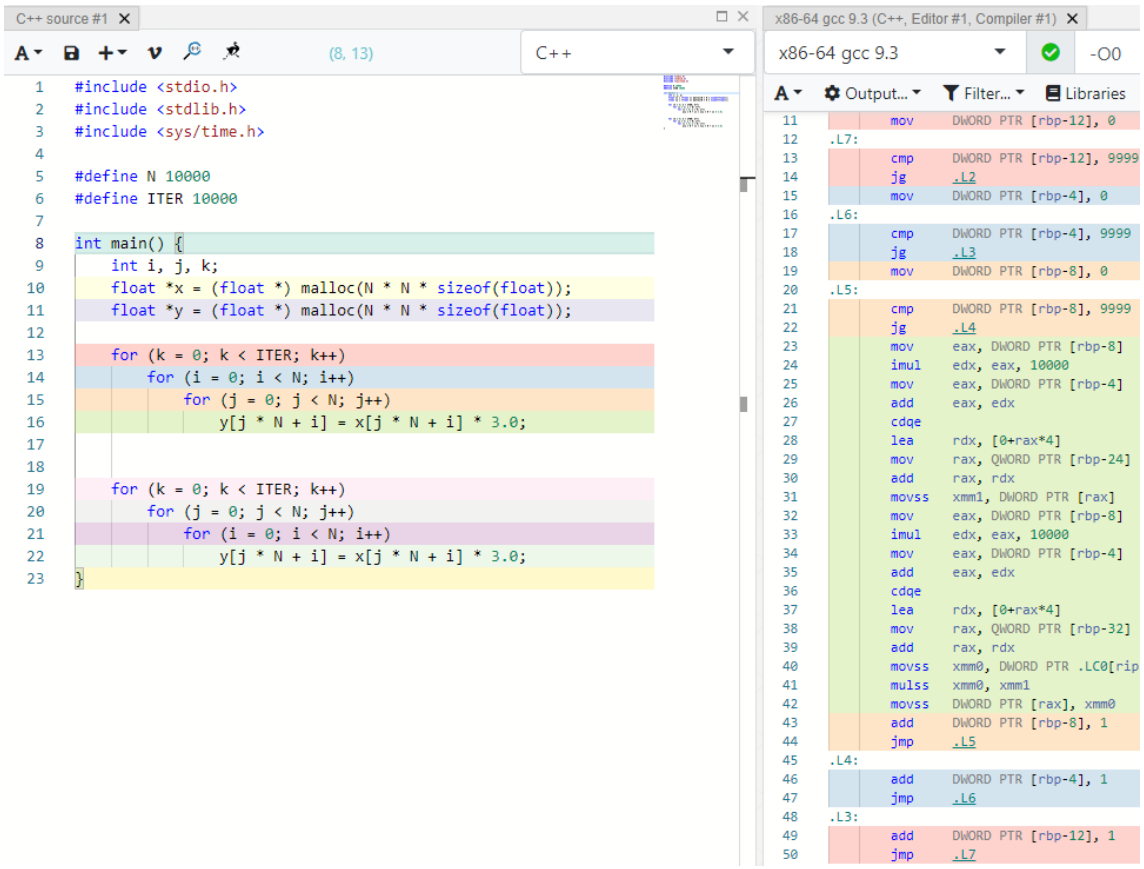
ANEXO

CÓDIGO FUENTE DE LAS PRUEBAS

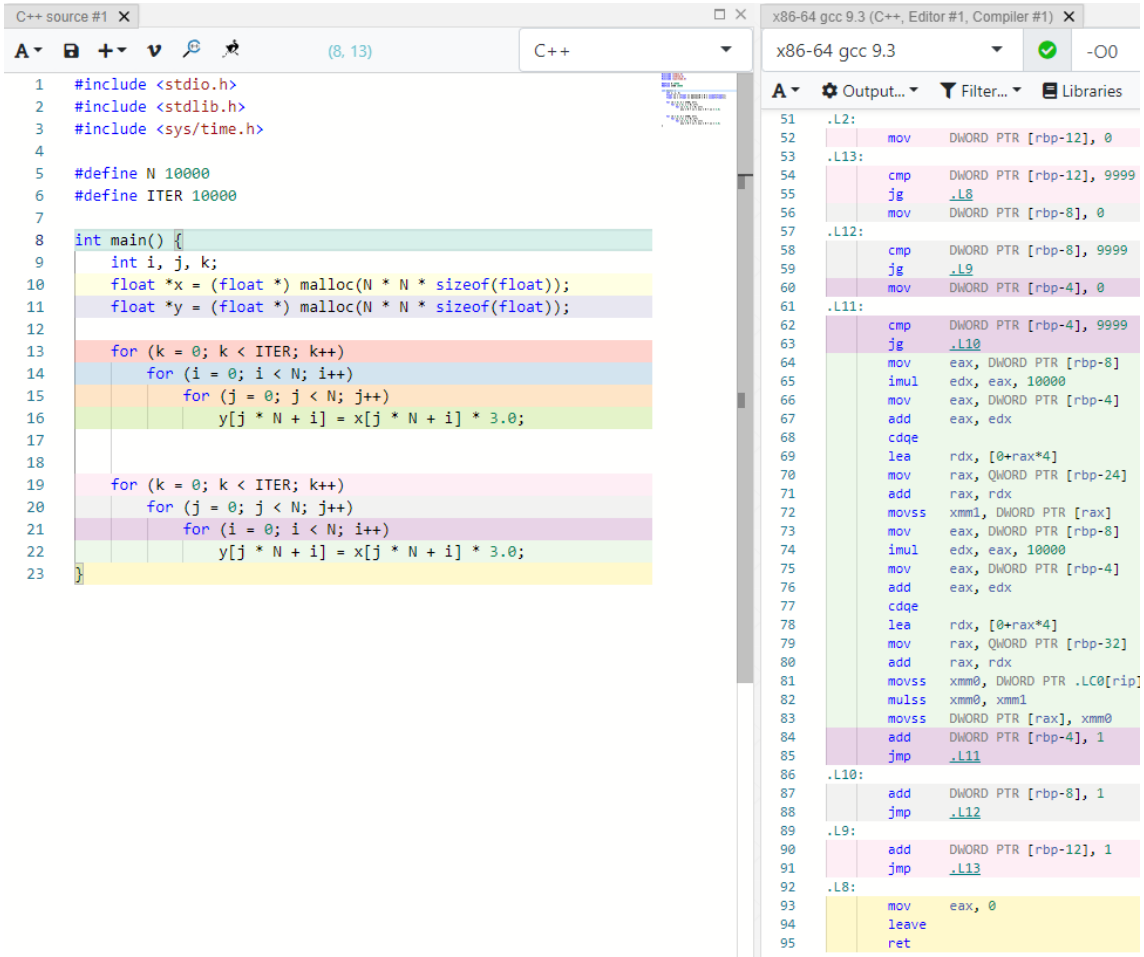
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define N 10000
#define ITER 100
// Valor auxiliar que nos permitirá obtener una media de los resultados obtenidos
#define MEAN 10

int main() {
    // Declaraciones de las variables necesarias para la ejecución de nuestro código
    struct timeval inicio, final;
    double tiempo, tiempo_opt;
    long i, j, k, n;
    float *x = (float *) malloc(N * N * sizeof(float));
    float *y = (float *) malloc(N * N * sizeof(float));

    // Inicializamos las variables que acumularán los tiempos obtenidos para realizar una media
    tiempo = 0;
    tiempo_opt = 0;

    // Calentamos la caché con los valores de iteración
    for (k = 0; k < ITER; k++)
            for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                    y[j * N + i] = x[j * N + i] * 3.0;

    // Realizamos n iteraciones para obtener una media, en este caso diez iteraciones
    for (n = 0; n < MEAN; n++) {
        // Tomamos la medida inicial de gettimeofday
        gettimeofday(&inicio, NULL);
        // Operamos las matrices
        for (k = 0; k < ITER; k++)
            for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                    y[j * N + i] = x[j * N + i] * 3.0;
        // Tomamos la medida final de gettimeofday
        gettimeofday(&final, NULL);
        // Añadimos el tiempo de ejecución consumido por iteración de nuestro código sin optimizar
        tiempo += (final.tv_sec - inicio.tv_sec + (final.tv_usec - inicio.tv_usec)/1.e6)/ITER;
    }

    // Realizamos n iteraciones para obtener una media, en este caso diez iteraciones
    for (n = 0; n < MEAN; n++) {
        // Tomamos la medida inicial de gettimeofday
        gettimeofday(&inicio, NULL);
        // Operamos las matrices
        for (k = 0; k < ITER; k++)
            for (j = 0; j < N; j++)
                for (i = 0; i < N; i++)
```

```
                    y[j * N + i] = x[j * N + i] * 3.0;
        // Tomamos la medida final de gettimeofday
        gettimeofday(&final, NULL);
        // Añadimos el tiempo de ejecución consumido por iteración de nuestro código optimizado
        tiempo_opt += (final.tv_sec - inicio.tv_sec + (final.tv_usec - inicio.tv_usec)/1.e6)/ITER;
    }

    // Salida de los resultados
    printf("\n> Tiempo por iteración del código sin optimizar: %lf\n", tiempo / MEAN);
    printf("> Tiempo por iteración del código optimizado: %lf\n", tiempo_opt / MEAN);
}
```

CÓDIGO ENSAMBLADOR



Código ensamblador correspondiente con la versión sin optimizar de nuestro código



Código ensamblador correspondiente con la versión optimizada de nuestro código