COMPILADORES E INTERPRETES

GENERACIÓN Y OPTIMIZACIÓN DE CÓDIGO

PRACTICA 2

Introducción

La técnica a analizar en esta práctica es la fusión de bucles. La fusión de bucles es una de las tantas técnicas que existen en la optimización de bucles, estos métodos de optimización buscan mejorar la velocidad de ejecución y reducir la sobrecarga asociada a los bucles, para conseguir esta mejora de rendimiento estos procedimientos mejoran el uso de la cache aumentando la localidad del programa y explotan las capacidades de procesamiento en paralelo de los procesadores. Estas mejoras de rendimiento suelen ser especialmente interesantes en programas científicos donde la mayoría del tiempo de ejecución lo ocupan los bucles.

Volviendo a la técnica a desarrollar en esta práctica la fusión de bucles consiste en combinar varios bucles adyacentes en uno solo, para poder hacer esta fusión deben 2 condiciones, la primera es que los bucles tienen que tener el mismo tamaño y la segunda es que los bucles no pueden ser dependientes entre sí. Para emplear la fusión de bucles habrá que examinar cada caso por separado ya que a pesar de disminuir la sobrecarga del bucle (proceso en el que se testea y se itera la variable del bucle) el rendimiento puede disminuir si las variables están en distintas zonas de memoria al disminuir la localidad espacial.

Viendo el código a optimizar se puede intentar predecir el resultado que tendrá la aplicación de la técnica, lo primero es comprobar que los bucles cumplen con las dos condiciones previamente citadas por lo que hay vía libre para proceder con la fusión, entrando en el tema del rendimiento se puede ver que los arrays están declarados uno detrás de otro por lo que no estarán muy lejos en memoria, además la versión optimizada mejora la localidad espacial ya que se usan los datos x[i] y y[i] justo después de iniciarlos, si juntamos todo lo anterior con el hecho de que se reduce la sobrecarga del bucle todo apunta a que al realizar la optimización se obtendrá una mejora de rendimiento.

```
int i, j, k;
float x[N], y[N];
for(k=0; k<ITER; k++){
    for(i=0; i<N; i++)
        x[i] = sqrt((float)i);
    for(i=0; i<N; i++)
        y[i] = (float)i+2.0;
    for(i=0; i<N; i++)
        x[i] += sqrt(y[i]);
    }
```

Versión sin optimizar

```
for(k=0; k<ITER; k++)
    for(i=0; i<N; i++) {
        x[i] = sqrt((float)i);
        y[i] = (float)i+2.0;
        x[i]+=sqrt(y[i]);
        }
```

Versión optimizada

Observación

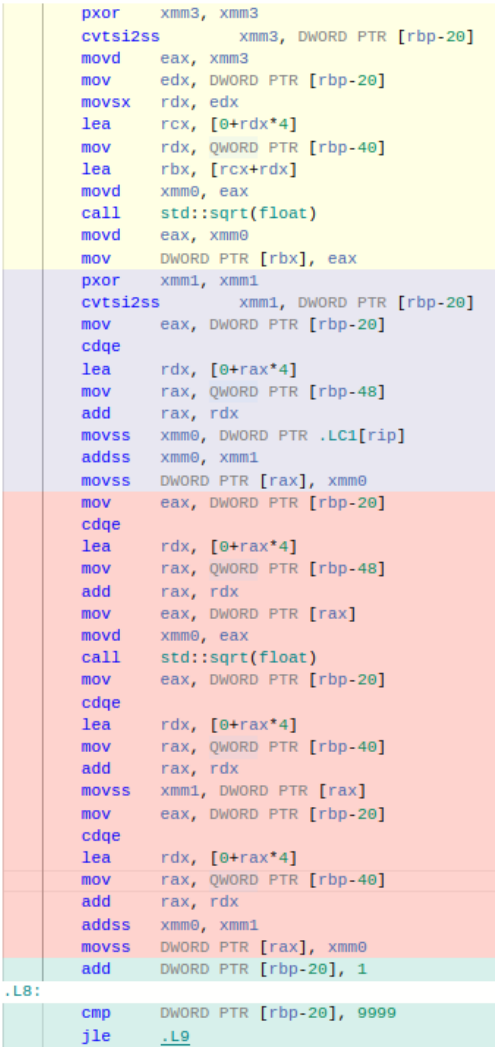
En este apartado se analizará el código ensamblador de los bucles para intentar encontrar las mejoras mencionadas en el apartado anterior, para para facilitar el análisis se utiliza la web https://godbolt.org/ con el compilador x86-64 gcc 11.2 y la opción de compilación -O0.

Observando ambos códigos se puede observar que las mejoras se implementaron de la forma esperada, ambos códigos son bastante parecidos, siendo la única diferencia destacable la sobrecarga en los bucles, realizándose en la versión sin optimizar muchas más veces las instrucciones de incremento, salto y comparación propias de los bucles. Como consecuencia del aumento de instrucciones el código optimizado es un poco más corto que el no optimizado con 235 y 249 líneas respectivamente.

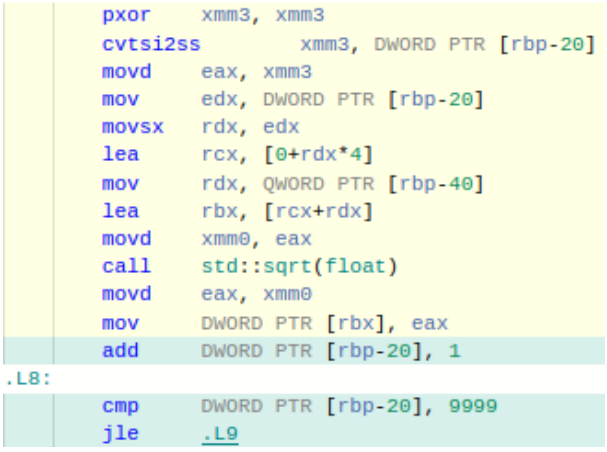
```
        add     DWORD PTR [rbp-20], 1
.L8:
        cmp     DWORD PTR [rbp-20], 9999
        jle     .L9
```

Se podría destacar también que como consecuencia de la diferencia de instrucciones de salto los bloques básicos de la versión sin optimizar son bastante más pequeños que el de la versión optimizada.

Viendo los códigos en ensamblador también se puede confirmar que la complejidad espacial mejora mucho en la versión optimizada ya que se usan los datos y[i] y x[i] justo después de inicializarlos, por lo que es muy probable que estos sigan en un nivel alto de la cache y no haya que traerlos desde la memoria principal.



Bloque básico en la versión optimizada



Bloque básico en la versión sin optimizar

Pruebas

En este apartado se discute el impacto real que tuvo la optimización con varios casos de prueba, para que las medidas de los experimentos sean fiables se ajustan los valores de ITER y N para que la ejecución final dure sobre una decena de segundos, además cada experimento se realizara 5 veces y se tomará la media como el valor final, como última medida para aumentar la precisión del experimento antes de cada ejecución del experimento se realizará un calentamiento la cache inicializando los punteros 'x' e 'y'.

Es interesante conocer también el entorno sobre el que se ejecutarán las pruebas, el ordenador es un portátil de 4 núcleos y 8 hilos a 2.3 Ghz, cuenta con 4 gigabytes de RAM y tiene una arquitectura de 64 bits.

```
//Calentamiento de cache
for (i = 0; i < N; i++)
{
        x[i] = rand() % 2;
        y[i] = rand() % 2;
}
```

Para medir los tiempos se utiliza la función de c gettimeofday() y se medirá únicamente el tiempo que tarda en ejecutarse el bucle ITER, sin embargo, como el tiempo que nos interesa es el de los bucles interiores se opera con el tiempo final dividido entre el número de iteraciones.

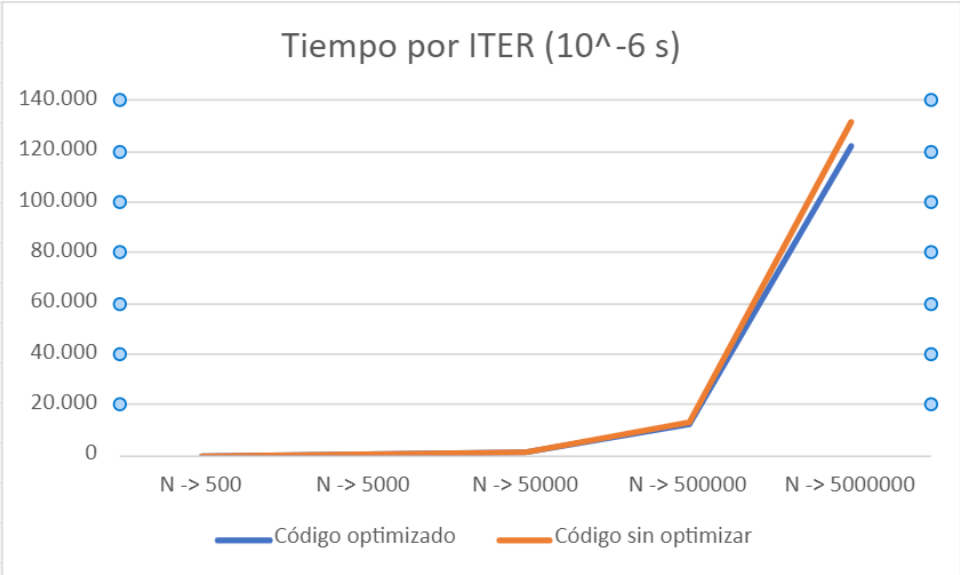
En total se realizan 50 pruebas para 5 valores de N distintos (5 pruebas para cada valor de N y una prueba para cada programa), los valores de N elegidos fueron:

- N -> 500
- N -> 5000
- N -> 50000
- N -> 500000
- N -> 5000000

Resultados: (Tiempo en ejecutar el bucle ITER dividido entre ITER. En microsegundos)

Valor de N	Valor 1	Valor 2	Valor 3	Valor 4	Valor 5	Media
Programa Optimizado						
N -> 500	12.060965	12.060965	11.879719	12.219258	12.033344	12.0139062
N -> 5000	119.81714	123.35575	121.37137	125.41111	124.14969	122.821012
N -> 50000	1219.7011	1204.6919	1196.6752	1215.4745	1193.6360	1206.03574
N -> 500000	12092.315	11857.584	12408.895	12006.561	11940.336	12061.1382
N -> 5000000	121435.53	123170.78	120453.38	120453.38	122192.58	121860.794
Programa no Optimizado						
N -> 500	13.154697	13.341427	13.316129	13.232731	13.421591	13.293315
N -> 5000	13262263	13147888	13603228	13271544	13252809	133.075464
N -> 50000	13331103	13384375	13423819	13685823	13775303	1352.00846
N -> 500000	13069376	13376739	13097065	13057266	13291732	13178.4356
N -> 5000000	13201894	13238286	13213760	13131600	13194744	131960.568

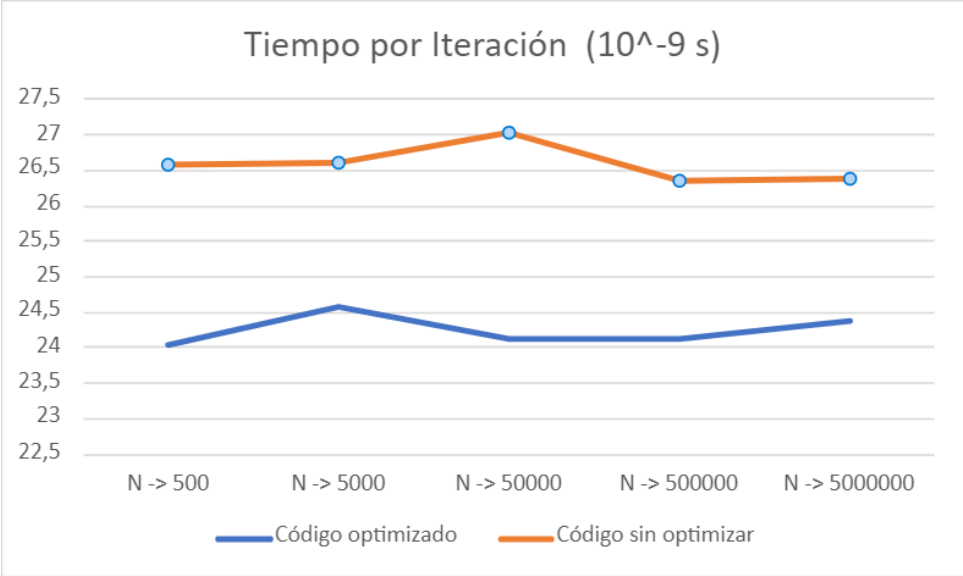
Para ver mejor los datos del experimento se representan las medias en un gráfico de líneas:



Viendo la gráfica es difícil sacar algo en claro ya que los valores de N son demasiado distintos, sin embargo, parece que el código optimizado tiene un mejor rendimiento en general. Para poder ver las cosas más claras se realiza una nueva representación gráfica en la que se muestra el tiempo medio que lleva ejecutar las instrucciones:

```
x[i] = sqrt((float)i);
y[i] = (float)i + 2.0;
x[i] += sqrt(y[i]);
```

Es decir, se dividirán los resultados anteriores entre N.



En esta gráfica se puede ver mucho mejor el aumento de rendimiento, para las pruebas realizadas el programa optimizado siempre tiene mejor rendimiento, lo cual era algo que se predijo desde la introducción por la disminución en la sobrecarga de los bucles (operaciones de incremento, comparación y salto que se realizan en cada iteración del bucle) y el aumento de la localidad temporal (acceso a los valores x[i] e y[i] justo después de inicializarlos).

Lo que también es muy interesante es que parece que el tamaño de N no parece influir demasiado en el rendimiento, cuando la teoría nos dice que al aumentar el tamaño de los vectores x e y los datos a utilizar en cada iteración estarán más lejos unos de otros empeorando así la localidad espacial y reduciendo el rendimiento de ambos programas, podría parecer que el rendimiento del segundo programa no tendría por qué empeorar ya que los vectores se inicializan por separado en el primer y segundo bucle, pero en el tercer bucle se utilizan variables de los dos bucles al mismo tiempo por lo que sería de esperar que también se redujese el rendimiento en esta versión, sin embargo, parece que el rendimiento no se ve afectado por este problema.

Conclusiones

Después de las pruebas realizadas se puede concluir que esta estrategia de optimización es útil y puede llegar a mejorar el rendimiento, sin embargo, no siempre se puede utilizar ya que tiene que cumplir 2 condiciones, que los bucles sean independientes y que sean del mismo tamaño. Además, aunque se cumplan las condiciones para llevar a cabo la fusión de bucles puede que el rendimiento no mejore o incluso empeore si se llega a deteriorar mucho la localidad, especialmente hoy en día que los procesadores tienen varios núcleos y pueden dividir las tareas en subtareas que se realizan al mismo tiempo, este factor es tan relevante que da lugar a una técnica de optimización opuesta a la abordada en esta práctica, la fisión de bucles.

Teniendo todo lo anterior en cuenta se puede afirmar que la fusión de bucles SI es una técnica útil pero solo si se tienen los conocimientos adecuados para utilizarla, además habrá unos casos en los que sea más interesante que en otros, en programas comunes en los que las operaciones de E/S ocupan el mayor porcentaje del tiempo de ejecución no tiene sentido preocuparse por este tipo de optimizaciones, ya que, aunque se dé el caso más favorable la mejora será ínfima, mientras que en programas científicos o de simulación en los que los bucles ocupan prácticamente la totalidad del tiempo de ejecución y tardan mucho tiempo en ejecutarse estás mejoras son muy interesantes y pueden dar lugar a mejoras importantes de rendimiento.

En conclusión, la fusión de bucles es una técnica interesante si se utiliza con los bucles y programas adecuados, sin embargo, si no se hace un uso correcto puede incluso ser perjudicial usarla ya que es posible que reduzca el rendimiento del programa.

Referencias

https://en.wikipedia.org/wiki/Loop_fission_and_fusion

https://www.techopedia.com/definition/8130/loop-fusion

https://www.sciencedirect.com/topics/computer-science/loop-fusion