Generación y optimización de código

Reemplazo de múltiples divisiones por una única y multiplicaciones

Compiladores e Intérpretes - GREI USC

Índice

Resumen

En este informe se lleva a cabo un análisis de la técnica de sustitución de divisiones mediante la aplicación y testeo sobre un caso práctico. Primeramente se presenta la técnica y sus características. A continuación se lleva a cabo un análisis del código ensamblador, que permite reafirmar algunas de las características previamente atribuidas. Se describen las pruebas a realizar y su entorno y se presentan los resultados y sus conclusiones, los cuales determinan una amplia mejora en el rendimiento para la versión optimizada que se incrementa hasta estabilizarse en un factor de mejora de 3.

Introducción

En este documento se llevará a cabo un análisis de la técnica de sustitución de divisiones por multiplicaciones para el código del Ejercicio 2 de la práctica. Para ello se realizarán una serie de pruebas, que tratarán de ser lo más fiables posibles y arrojar datos útiles sobre el rendimiento de la optimización aplicada.

Para dicho análisis se comenzará por un apartado de presentación de la técnica, en el que se describirá en qué consiste y sus posibles ventajas y desventajas. A continuación, se describirá el hardware del equipo, para el cual se generará código ensamblador, cuyo análisis formará el siguiente apartado. Seguido de dicho análisis se presentará el entorno y se describirán las pruebas a realizar. Los resultados de dichas pruebas se presentarán de forma gráfica y se analizarán en el siguiente apartado. Por último, se elaborarán conclusiones del trabajo realizado.

Técnica

En primer lugar se presentará la técnica utilizada para optimizar el código. En este caso se trata de una técnica basada en la sustitución de múltiples divisiones por una única división y posteriores multiplicaciones. Esto es posible ya que, basándose en las propiedades matemáticas de la división y la multiplicación, es posible sacar factor común en el numerador, de forma que pueda multiplicarse por este a continuación. Así mismo también es posible multiplicar por un número presente en el denominador, de forma que se anule su efecto sobre el cociente resultante.

De esta forma partimos del código de la Figura 1 donde se realizan 5 divisiones en punto flotante:

```
gettimeofday(&inicio,NULL);
for (j = 0; j < ITER; j++)
{
    for(i=0; i<N; i++){
        x = (float)i+1.1;
        y = (float)j+x;
        a = 1.0 / x;
        b = 1.0 / (x*y);
        c = 1.0 / y;
        d = v /(x*y);
        e = 2.0*v /(x*y);
    }
}
gettimeofday(&final,NULL);
```

Figura 1: código sin optimizar

Para optimizar el código y reducir el número de divisiones, se establece como fracción común 1/(x*y) y se reestructuran los cálculos de las demás variables en función de este, por ejemplo para el cálculo del valor de a bastará con multiplicar por y para eliminarlo del denominador, o en el caso de d se multiplicará por v para que este pase a formar parte del numerador. De esta forma se reduce tanto el número de divisiones (pasando a ser 1) como el de multiplicaciones en casos como el de e, que pasa de 2 multiplicaciones y una división a una única multiplicación.

```
gettimeofday(&inicio,NULL);
for (j = 0; j < ITER; j++)
{
    for(i=0; i<N; i++){
        x = (float)i+1.1;
        y = (float)j+x;
        b = 1.0/(x*y);
        a = y * b;
        c = x * b;
        d = v * b;
        e = 2.0*d;
    }
}
gettimeofday(&final,NULL);
```

Figura 2: código optimizado

Ventajas e inconvenientes

La aplicación de la técnica descrita anteriormente tiene su principal ventaja en la mayor eficiencia computacional que presentan las multiplicaciones en punto flotante en comparación con las divisiones.

Si bien son operaciones de complejidades relativamente similares (entre O(n) y O(n*log n), la paralelización en el hardware es menor ya que, a diferencia de la multiplicación, la división no es ni asociativa ni conmutativa, esto exigirá una ejecución mucho más secuencial. En cuanto a la multiplicación, sus propiedades permiten paralelizarla de forma sencilla, reduciéndola a multiplicaciones menores que podrán realizarse paralelamente y luego fusionarse con una simple suma.

Destacar además que gracias a esta optimización se ahorran algunas otras operaciones fruto de reutilizar los valores como ocurre en el caso del cálculo de e (pasando de 2 multiplicaciones y una división a una única multiplicación).

En cuanto a las posibles desventajas, no se observa ninguna especialmente salientable más allá del mayor esfuerzo de programación que puede suponer el buscar fracciones comunes a la hora de desarrollar código que cuente con estas optimizaciones.

Hardware

La máquina en la cual se ejecutarán las pruebas tendrá asignados dos de los 12 procesadores lógicos disponibles en el procesador Ryzen 5 5600G, los cuales estarán funcionando a una frecuencia de 3.9GHZ. En lo referente a la memoria de la máquina, esta tendrá asignados 4096Mb de memoria DDR4 a 3200Mhz, de los 16GB con los que cuenta el equipo. La máquina dispondrá además de 32MB de memoria de vídeo y un disco duro de 50GB reservados de forma dinámica.

La ejecución de las pruebas tendrá lugar en una máquina virtual Linux con el sistema operativo Ubuntu 20.04. Utilizando para su ejecución el software Virtual Box desde un sistema operativo Windows 10.

Análisis ensamblador

En este apartado se presentará el código ensamblador resultante, así como sus características. Para obtener dicho código ensamblador se utilizó la herramienta online Compiler Explorer. Destacar además que se compiló para una arquitectura x86 64 con la versión 9.4 de gcc.

En primer lugar se muestra en la Figura 3 el código ensamblador de la versión sin optimizar. En el lado izquierdo se indican con líneas verdes los bloques básicos, los cuales se pueden distinguir entre los relativos a las instrucciones de los bucles (los de menor tamaño) y el del cuerpo del bucle interno. Este último es el bloque básico de mayor tamaño y está constituido por 41 instrucciones. En él se pueden observar las operaciones tanto de división (divss) como de multiplicación (mulss).

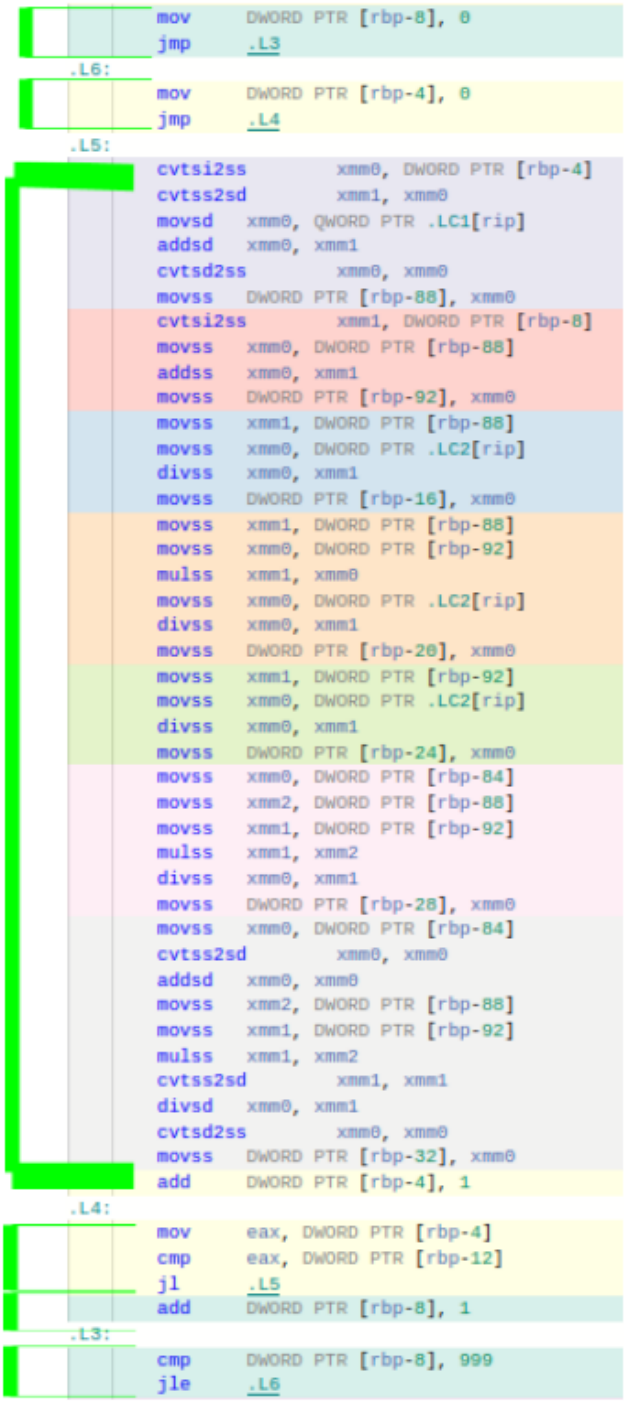

Figura 3: Código ensamblador sin optimizar

En este caso se muestra en la Figura 4 el código ensamblador resultante del programa optimizado. Al igual que en el anterior se pueden identificar bloques básicos de menor tamaño relativos a los bucles (son los mismos ya que no se han modificado los bucle). Las principales diferencias se hallan en el bloque básico de mayor tamaño (el cuerpo del bucle), en el cuál se puede observar como han desaparecido las instrucciones de división (divss) y han sido sustituídas por multiplicaciones. Además, la modificación del código no solo implica la sustitución de divisiones, si no que además se reduce el número de instrucciones necesarias en el bloque básico del bucle a 32.

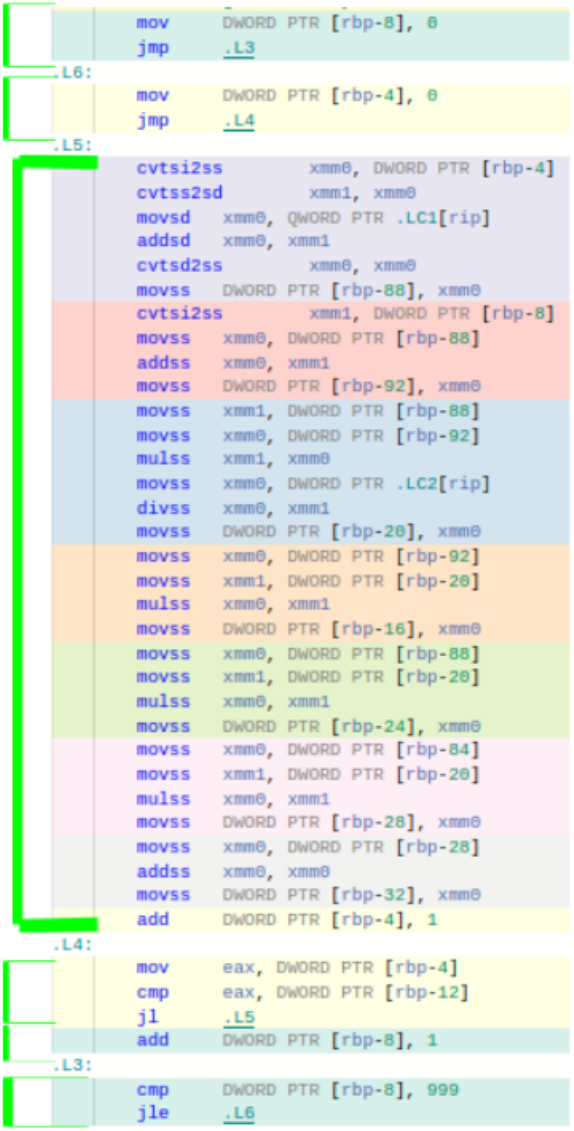

Figura 4: Código ensamblador optimizado

Dado el análisis del código ensamblador llevado a cabo, es posible afirmar que la optimización se ha llevado a cabo con éxito ya que, como se ha visto, las divisiones han sido sustituidas de forma efectiva por instrucciones de multiplicación en punto flotante. Esta no es la única diferencia observable, y es que la reducción en el número de instrucciones del bloque básico del cuerpo del bucle de 41 a 32 es un posible indicador de que esta técnica reducirá los tiempos de ejecución con alta probabilidad. Por último destacar que como no se han realizado modificaciones en las instrucciones que controlan el flujo del programa, no existen variaciones en el número de bloques básicos.

Entorno y pruebas

Para la ejecución de las pruebas se tratará de reducir el ruido en el proceso de toma de datos mediante la reducción al mínimo de los procesos activos en el dispositivo.

En lo referente al proceso de compilación, se utilizará la versión 9.4 de gcc con la opción -O0, lo cual permitirá reducir las optimizaciones realizadas por el compilador, con el objeto de que el código ensamblador resultante sea lo más fiel posible al código fuente sobre el que se aplica la optimización a objeto del análisis de este informe.

Cabe puntualizar que el uso de dos ejecutables diferentes hace innecesario el calentamiento de caché, el cúal sí sería preciso de ejecutarse ambos en el mismo código, ya que el primero de ellos sería mucho más penalizado por los fallos caché.

Las pruebas consistirán en la ejecución reiterada del ejecutable en C, el cúal devuelve el tiempo de ejecución medio. De esta forma el código será ejecutado un número de iteraciones igual al producto por el número de iteraciones establecido en el código (ITER), que en este caso será 1000, por el número de ejecuciones del ejecutable. El objetivo de realizar diversas ejecuciones a mayores de las ya establecidas en el código es tanto reducir las posibles discordancias como la extracción de otros datos útiles mediante la automatización del proceso con un script de bash. En este caso se extraerán máximos y mínimos.

De esta forma las pruebas se realizarán mediante dicho script, el cual recibe como parámetros los nombres de los ejecutable y el número de veces a ejecutar cada uno para cada valor de N (tamaño de los array). De esta forma el script recorrerá un conjunto de valores de N (1.000, 50.000, 100.000, 150.000, 250.000, 500.000, 750.000 y 1.000.000) ejecutando el código optimizado y sin optimizar por separado almacenando los resultados para el cálculo de media, máximo y mínimo. En este caso se optó por ejecutar un total de 10 veces cada ejecutable, por lo que la invocación del script si se desea repetir las pruebas sería:

```
./script.sh nombre_ejecutable1 nombre_ejecutable2 10
```

Una vez ejecutadas las pruebas se extraen los obtenidos para su procesamiento y posterior presentación en el apartado de resultados.

Resultados

En este apartado se presentarán los resultados obtenidos en un formato visual que facilite su interpretación. En primer lugar, se presentará una gráfica en la que se recogen los tiempos medios de cada código en función del valor de N. El eje vertical se corresponde con el tiempo en segundos, mientras que en el horizontal se representan los valores de N. La línea naranja se corresponde con los resultados de los tiempos medios para el código sin optimizar, mientras que la azul se corresponderá con los tiempos del código optimizado. En la Figura 5 se puede hallar dicha gráfica.

Figura 5: Tiempos medios.

Como se puede observar en la anterior gráfica, los tiempos obtenidos son considerablemente mejores para el código optimizado. Este hecho se acentúa a medida que aumenta el valor N, poniéndose de manifiesto el ahorro computacional fruto del menor número de instrucciones y de la sustitución de las divisiones por multiplicaciones. En el punto donde N es igual a 1.000.000 el tiempo necesario para ejecutar el código sin optimizar triplica al del tiempo optimizado (0,013s frente a los 0,0043s del optimizado).

En la siguiente gráfica se presentan unos datos similares, solo que en esta ocasión se añaden los valores máximos y mínimos obtenidos para las 10 ejecuciones efectuadas desde el script (descripción detallada en el apartado de Entorno y Pruebas). En la Figura 6 se muestra dicha gráfica.

Figura 6: Máximos y mínimos.

En este caso la gráfica obtenida, si bien no permite extraer nuevas conclusiones, sí permite refrendar la fiabilidad de las pruebas, ya que las discordancias entre máximos, mínimos y medias son prácticamente inexistentes, lo que hace poco probable que los resultados obtenidos estén de cierto modo sesgados o carezcan de validez.

Una vez presentadas estas gráficas, se puede evaluar también en qué medida el código optimizado mejora al original. Para ello se utilizará el speed-up, que se calculará como el cociente entre el tiempo empleado por el código original y el optimizado. De esta forma un valor mayor que uno supondría que el código optimizado mejora al original, igual a uno que mantiene el rendimiento y menor que 1, lo empeora. El resultado de dicho cálculo se muestra en la Figura 9.

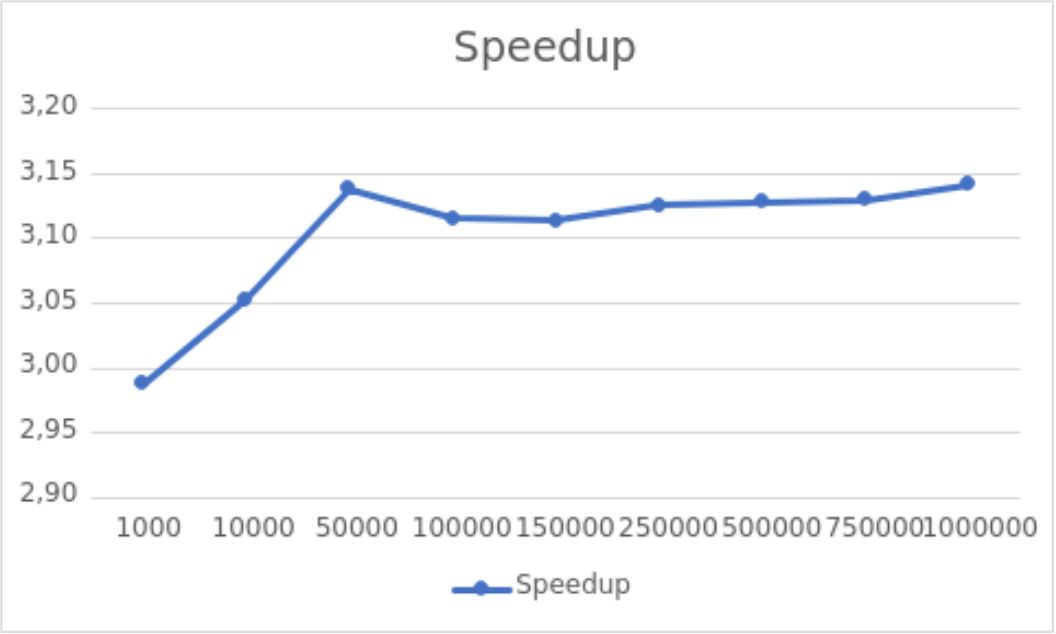

Figura 9: Speed-up.

La aplicación del speed-up permite observar que la mejora con respecto a los tiempos del código original se incrementa especialmente entre los primeros valores de N y que, una vez alcanzado el 50.000, esta se estanca entorno a un factor de mejora de entre 3,10 y 3,15, los cuales asemejan ser los límites de mejora alcanzables con dicha optimización. Dichos valores son coherentes con lo obtenido en la primera gráfica, ya que como se puntualizaba previamente el código sin optimizar triplicaba el tiempo del optimizado. Destacar además que, pese a que el factor de mejora se estabilice, la diferencia entre el tiempo de ambos códigos seguiría creciendo al depender esta del tamaño del problema. Se podría utilizar la siguiente fórmula para aproximar el tiempo de ejecución del código optimizado:

$$Toptimizado = \frac{1}{3} \cdot Tsinoptimizar$$

Conclusión

El análisis llevado a cabo hasta el momento comenzó con una presentación de la técnica de sustitución de divisiones aplicada y sus posibles ventajas y desventajas. Algunas de sus ventajas son el ahorro de instrucciones y la ganancia en la capacidad de paralelización del código gracias a sustituir las divisiones por multiplicaciones. Por otra parte, no se encontraron desventajas más allá de posibles esfuerzos a mayores de aplicar la técnica.

El análisis continuaba con la inspección del código ensamblador resultante, el cúal permitía comprobar como efectivamente se habían eliminado las instrucciones de división, así como la reducción de instrucciones en el bloque básico del cuerpo del bucle.

Por último, se llevaron a cabo las pruebas en un entorno controlado y descrito que pretendió reducir el ruido en los resultados lo máximo posible. Estas mostraron resultados ampliamente favorables para la optimización realizada, la cual alcanzaba a triplicar el rendimiento del código original. El hecho de lograr un código 3 veces más rápido para valores grandes pone en valor el uso de esta técnica de optimización, ya que constituye una mejora más que considerable y que, para tamaños de problema muy grandes, será clave el poder reducir en un tercio el tiempo de ejecución.

En definitiva, la sustitución de divisiones constituye una optimización altamente efectiva que debe ser tomada en cuenta a la hora de desarrollar código eficiente, ya que sin excesivo esfuerzo constituye una mejora en el rendimiento más que significativa.

Bibliografía

[1] Marcel Noe, Diploma in Computer Science & Biomedical Engineering, Karlsruhe Institute of Technology, Why does division of floating point numbers take a lot more time than multiplication of floats in computers?
https://www.quora.com/Is-floating-point-multiplication-faster-than-division

[2] Programación en ensamblador (x86-64), Miguel Albert Orenga y Gerard Enrique Manonellas PID_00178132
https://www.exabyteinformatica.com/uoc/Informatica/Estructura_de_computadores/Estructura_de_computadores_(Modulo_6).pdf