-Nombre Oculto-
-Email Oculto-
28 de diciembre de 2016

Práctica 2: Generación Y Optimización De Código

Reemplazo de multiplicaciones y divisiones enteras por operaciones de desplazamiento (Optimización 7)

En este documento se explicará las utilidad de esta técnica para obtener mejor rendimiento a la hora de optimizar operaciones de multiplicación y división. En los siguientes apartados veremos en que se basa esta técnica para lograr mejores tiempos de ejecución para luego hacer una revisión de los resultados que hemos obtenido al aplicarla a nuestro pequeño programa ejemplo. Finalmente comentaremos las conclusiones obtenidas durante nuestro proceso de análisis y daremos recomendaciones sobre cuando se deberían tener en cuenta este tipo de técnicas.

Tabla de contenido:

1.- Descripción de la técnica utilizada

La técnica que estamos utilizando en lugar de realizar las operaciones de multiplicación y división tradicionales es conocida como "Bit-Shifting" o Desplazamiento de Bits. Este tipo de operación tiene sentido una vez comprendemos como los procesadores utilizan una lógica binaria para representar números en formato decimal (y en cualquier otro formato). Como estamos trabajando en un formato binario necesitaremos una serie de bits que representarán un número relevante para el usuario (en formato decimal en este caso), dicho número estará representado en un array de ceros y unos en el que la posición n empezando a contar por 0 y por la derecha representa el valor de 2^n.

Una vez establecido esto es fácil comprender que si movemos un 1 a la posición que está a su izquierda estaremos sumando 1 al valor de n anterior al que se elevaba 2, obteniendo de forma efectiva 2^{n+1}. De la misma forma, si lo movemos a la derecha estaremos haciendo lo contrario y obtendremos el valor de 2^{n-1}. Si generalizamos esto y lo que hacemos es mover todos los ceros y unos presentes en el array lo que estaremos haciendo es sumar o restar un número de posiciones determinadas a n, haciendo que al calcular el número en formato decimal estemos multiplicando o dividiendo por 2 un número determinado de veces todos los factores (el mismo numero de veces que posiciones hemos movido hacia la izquierda, multiplicando o derecha, dividiendo). Al multiplicar o dividir todos los factores lo que estamos haciendo es multiplicar o dividir por 2 todo el número una cierta cantidad de veces.

Como esto se basa en multiplicar o dividir por 2 las operaciones que se pueden llevar a cabo con un solo operador de desplazamiento de bits son limitadas a multiplicaciones y divisiones por múltiplos de 2. Sin embargo se puede considerar una forma generalizada de utilizar el desplazamiento si combinamos desplazamiento y suma/resta de forma que al combinar diferentes resultados de desplazamientos obtengamos una multiplicación o división por un factor que no es múltiplo de dos como en el siguiente ejemplo:

```
x * 14 == x * 16 - x * 2 == (x << 4) - (x << 1)
```

Sin embargo dependiendo de la complejidad añadida es posible que se supere la complejidad original de realizar la multiplicación directamente.

Comentar las ventajas o desventajas de esta técnica depende de si estamos considerando puramente las multiplicaciones y divisiones por múltiplos de dos o no. En este caso, como en el código aportado solo se usan múltiplos de dos consideraremos que solo se permiten este tipo de operaciones.

Dentro de las ventajas que nos aporta el método solo podemos comentar la pura velocidad en comparación con realizar la multiplicación original pues para el procesador es mucho más sencillo mover ciertos bits un número concreto de posiciones hacia un lado que realizar una compleja multiplicación en formato binario. En la tabla de costes de operaciones referencia da en el último apartado hemos comprobado que entre varios modelos diferentes de Intel y AMD las multiplicaciones de enteros convencionales suelen tener una latencia de entre 4 y 6 ciclos cuando el desplazamiento de bits tanto a la derecha como a la izquierda suele tener una latencia de 1 ciclo solamente. Es decir, en teoría, siempre que podamos elegir sin consecuencias entre las dos formas de multiplicar o dividir deberíamos optar por el desplazamiento de bits.

El intentar buscar desventajas a esta técnica no es una tarea fácil. Es cierto que es posible que se den casos de overflow o underflow en los que perdemos bits relevantes al desplazar a la izquierda y a la derecha respectivamente pero esto también puede ocurrir con multiplicaciones y divisiones convencionales. Una de las desventajas que sí se podría ver claramente es que hacemos que nuestro código sea más difícil de leer pues es más fácil entender una operación tan común como la multiplicación que ver un operador poco frecuente que sirve para desplazar bits y tener que pensar en lo que eso significa para el procesador en formato binario además de tener que pensar el valor de 2^n y si estamos dividiendo o multiplicando.

Una nota muy importante que veremos claramente en el siguiente apartado es que este tipo de optimizaciones son fáciles de realizar por un compilador y siempre que ven un caso donde son muy simples de aplicar las aplican directamente por lo que aunque en nuestro código no hagamos desplazamiento de bits, si tenemos una multiplicación/división aislada por un múltiplo de dos es altamente probable que el compilador la transforme a desplazamiento de bits en el ensamblador que genera y por lo tanto también en el ejecutable.

2.- Resultados obtenidos

Al aplicar los scripts referenciados en el apartado de recursos utilizados hemos obtenido una serie de datos que hemos introducido en una hoja de cálculo en la plataforma online Drive. A la hora de obtener datos hemos seguido dos aproximaciones diferentes, primero hemos utilizado una serie de valores balanceados para el número de iteraciones que realizaremos del problema y para el tamaño del problema N. De esta forma, con un N pequeño hemos aumentado el número de iteraciones para hacer que el programa tardara un tiempo relevante en terminar eliminando así posibles errores de redondeo en las medidas. Sin embargo, con esta aproximación no tiene sentido comparar el tiempo entre distintos valores de N pues no es la única variable que cambia. Para solucionar esto hemos realizado otra serie de pruebas en las que siempre realizamos el mismo número de iteraciones y vamos variando el valor de N.

Tanto la gráfica de datos balanceados[1] como la tabla de datos con las mismas iteraciones[2] se encontrarán en un enlace al pié de esta página y en el apartado de recursos utilizados.

Para comprender los datos que vamos a ver es muy importante darse cuenta de que todos los compiladores actuales con los que se ha intentado aplicar esta técnica (GCC, Clang, …) han realizado optimizaciones automáticamente incluso cuando se les ha especificado explícitamente que no lo hicieran (por ejemplo con la opción -O0). Es decir, al ser esta técnica es tan útil y tener virtualmente cero desventajas al ser aplicada por el compilador es siempre considerada válida aunque se pida que no se realice optimización alguna.

Esto ha causado que pese a que hayamos intentado compilar el código no optimizado de muchas maneras diferentes, el ensamblador y ejecutable generado sea siempre prácticamente igual que el optimizado, dándolos así, como es lógico, tiempos casi iguales en ambos casos. Cuando los compiladores que hemos utilizado detectan una operación de multiplicación o división aislada por un múltiplo de 2 automáticamente generan instrucciones de desplazamiento de bits, no introduciendo instrucciones de multiplicación y división convencionales como se pudiera esperar. Incluso al intentar especificar al compilador que no haga ninguna optimización con las diferentes opciones que ofrecen siguen realizando esta optimización concreta, haciendo imposible obtener datos relevantes sobre la diferencia en eficiencia de ambas implementaciones.

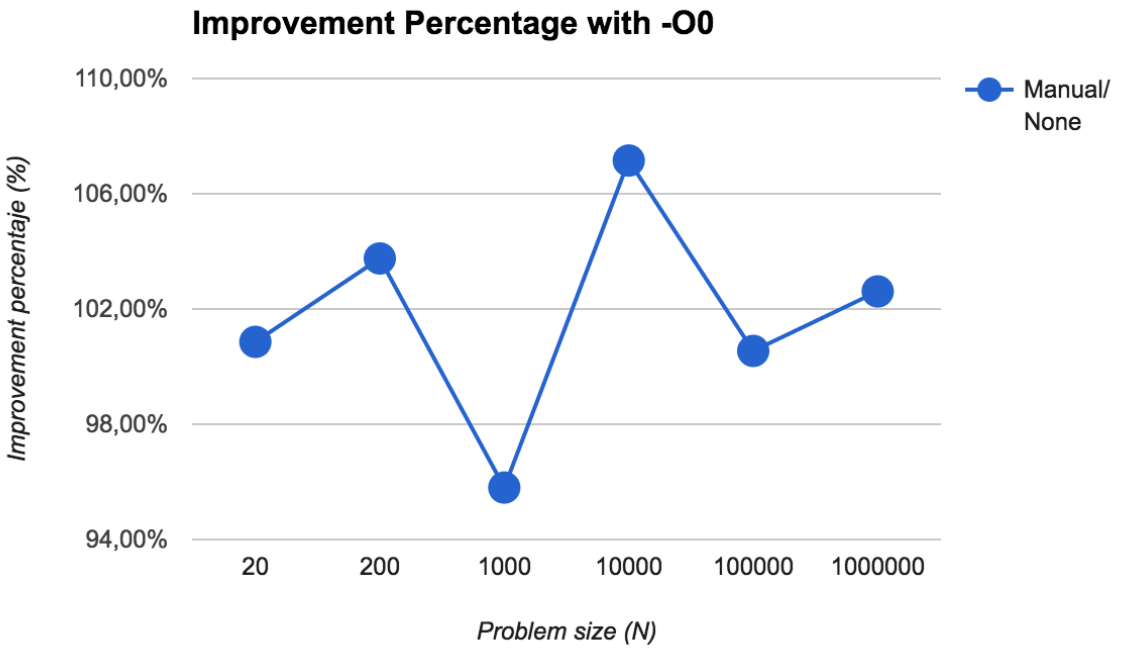
---

[1] Enlace a la tabla de datos balanceados: https://docs.google.com/spreadsheets/d/1j1Qio93CWFqvqkHXzEfdiIpg3-WtqrTAa1jf_yf5UL8/pubhtml

[2] Enlace a la tabla de datos con las mismas iteraciones: https://docs.google.com/spreadsheets/d/1IPpWcS73tzDOe4KNIfHtD2I9bg9xyD68W5T8z-zmbOQ/pubhtml

En los siguientes apartados analizaremos las diferencias que vemos según variemos el tamaño del problema y luego examinaremos las diferencias con los diferentes niveles de optimización que nos aporta el compilador.

2.1.- Variando el tamaño del problema



Como podemos ver en esta gráfica, el porcentaje de mejora es cercano al 100% en todos los casos, pudiendo asociar las variaciones que vemos a varianza en el tiempo de ejecución en este equipo. Prueba de que estamos ejecutando prácticamente el mismo código es que el ensamblador es prácticamente igual (comentaremos esto en el siguiente apartado) y que en algunos casos, la versión con el desplazamiento en código de alto nivel es peor que la versión sin mejoras, lo cual no tiene demasiado sentido.

Al comparar los ejecutables y el ensamblador para los diferentes tamaños vemos que las diferencias son prácticamente inexistentes aparte de los valores usados para controlar el número de iteraciones y el tamaño de N. En los datos de las pruebas balanceadas que intentan minimizar la varianza y error de los tiempos de ejecución vemos que la mejora es prácticamente siempre de un 100% independientemente del tamaño del problema, esto es porque estamos, como ya hemos dicho, ejecutando el mismo programa.

2.2.- Diferentes opciones de compilación

Al analizar los ensambladores y ejecutables generados con las opciones -O0, -O1, -O2 y -O3 hemos encontrado información interesante en lo que se refiere al contenido del programa.

Comenzando por la opción -O0: Este es el caso en el que los cidigos ensamblador de las dos versiones no son idénticos, cuando hacemos un diff de ambos archivos obtenemos que después de realizar las multiplicaciones y divisiones (ambas versiones con desplazamiento de bits) la versión no optimizada en código introduce tres instrucciones más que son las siguientes:

```
>     leal 31(%rax), %edx
>     testl     %eax, %eax
>     cmovs     %edx, %eax
```

Dichas instrucciones parecen ser fruto de falta de optimizaciones al trabajar con multiplicaciones y divisiones de enteros pues lo que se está haciendo es calcular una dirección e introducirla en un registro, luego se comprueba si %eax es cero para activar o no la flag que se usará en el salto de bucle ("Zero Flag") y luego se copia el contenido de %edx a %eax. Hemos comprobado que si eliminamos estas tres instrucciones del cidigo ensamblador y lo compilamos a ejecutable, el comportamiento es el mismo, de hecho obtener el mismo ejecutable que si compilamos la versión optimizada original. Con esta información podemos inferir que estas tres instrucciones son efectivamente causadas por una falta de optimización en operaciones de multiplicación y división sin usar desplazamiento por parte del compilador cuando se usa la opción -O0.

Estas tres instrucciones parecen ser las causantes de que al comparar las dos versiones obtengamos siempre una pequeña mejora en la optimizada (que no tiene estas instrucciones) por lo que no deberemos asociar dicha mejora a la técnica de optimización pues ambos la usan (ver comparativa de los datos balanceados).

Con -O1 si hacemos un diff de tanto el ensamblador como el ejecutable optimizado con el no optimizado no obtenemos absolutamente ninguna diferencia, ambos programas son exactamente el mismo, lo que se puede ver en los datos que comparan las mejoras que siempre son cercanos al 100%.

Finalmente, como nota interesante, hemos comprobado que el compilador con las opciones -O2 y -O3 simplemente realiza los bucles en tiempo de compilación introduciendo el resultado en un registro y no recalculándolo en tiempo de ejecución por lo que los tiempos

son prácticamente de 0 segundos. Al observar los códigos ensamblador de estas dos opciones vemos como efectivamente no encontramos las instrucciones asociadas a los bucles que sí veíamos previamente y podemos ver claramente que se está introduciendo el resultado ya obtenido en un registro que luego se imprime por lo tanto, desde el punto de vista de esta técnica de optimización, estas dos versiones no tienen demasiado valor para nosotros.

3.- Conclusiones obtenidas y recomendaciones de uso

Pese a que no hemos sido capaces de ver realmente el porcentaje de mejora que nos hubiera dado esta técnica si hemos descubierto conceptos muy interesantes en lo que respecta a la optimización y a este tipo de técnicas en concreto.

Lo primero fue darnos cuenta de que si una optimización no tiene desventaja alguna, el compilador la aplicará por nosotros. Además, es sorprendente que automáticamente el compilador busque de forma activa por este tipo de operaciones claramente optimizables y lo haga por nosotros en todos los casos. Esto debería servir de ejemplo de como han mejorado los compiladores actuales ya que intentando buscar información por la red sobre esta mejora hemos descubierto que versiones bastante antiguas de GCC no realizaban esta optimización automáticamente.

Otra cosa que sacamos de esto es que debemos tener cuidado con realizar optimizaciones demasiado temprano pues en este caso nos hubiéramos preocupado de comprobar que las operaciones de desplazamiento están bien hechas y que hacen lo que queremos para luego no obtener beneficio alguno y quedarnos con un código un poco más ilegible que antes. Por lo tanto debemos ser conscientes de la potencia de los compiladores actuales y confiar en ellos a la hora de realizar optimizaciones claras como esta.

Un detalle interesante es lo que hemos visto en las opciones -O2.y -O3 que directamente precalculaban el resultado. Aquí vemos otro ejemplo más de lo avanzados que están los compiladores actuales pues son capaces de detectar los bucles que se pueden precalcular en tiempo de compilación y hacerlo ahorrándonos una cantidad inmensa de ciclos en tiempo de ejecución.

En lo que respecta a recomendaciones de uso efectivamente es una técnica extremadamente util por si misma pero hemos comprobado que el compilador sabe cuando se debe aplicar y cuando no por lo que no recomendaría preocuparse mucho con esta mejora en concreto. Es cierto que para factores que no son múltiplos de dos se puede hacer desplazamiento+suma/resta pero los ciclos necesarios pueden superar a los que hubiera usado la operación original. En general, considero esta técnica extremadamente util e importante pero se debería que sea el compilador el que la aplique cuando sea necesario, liberándonos a nosotros del trabajo y dándonos códigos fuente más legibles.

4.- Recursos Utilizados

- Código fuente adjunto a este documento:
    - En el directorio /scripts encontraremos los scripts para compilar (compile*) y ejecutar (run*) tanto la versión que varía las iteraciones como la que no (en cada uno de los directorios correspondientes).
    - En el directorio /src veremos el ejecutable usado para las experimentaciones: optimization_7.c.
    - El apartado /out tendrá un subdirectorio por opción de compilador que contendrá los ejecutables y los códigos ensamblador asociados además de otro subdirectorio /logs con los datos obtenidos de las ejecuciones.
- Enlace a la tabla de datos balanceados: https://docs.google.com/spreadsheets/d/1j1Qjo93CWFqvqkHXzEfdiIpg3-WtqrTAa1jf_yf5UL8/pubhtml
- Enlace a la tabla de datos con las mismas iteraciones (pulsar en la pestaña "Graphs" para ver las gráficas asociadas a los datos obtenidos): https://docs.google.com/spreadsheets/d/1IPpWcS73tzDOe4KNIfHtD2I9bg9xyD68W5T8z-zmbOQ/pubhtml
- Post sobre optimizaciones en C y C++: https://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Code_optimization/Faster_operations
- Post sobre "shift and add": https://en.wikipedia.org/wiki/Multiplication_algorithm#Shift_and_add
- Tabla con costes de instrucciones: http://www.agner.org/optimize/instruction_tables.pdf
- Web con información sobre instrucciones en ensamblador: https://en.wikibooks.org/wiki/X86_Assembly