Compiladores e Intérpretes

Práctica 2: Generación y optimización de código

Reemplazo de multiplicaciones y divisiones enteras por operaciones de desplazamiento

AUTOR ANÓNIMO

Universidade de Santiago de Compostela

Índice

Versión: 1 de mayo de 2025, 18:37

1. Introducción

El rendimiento de los programas informáticos depende en gran medida de cómo se implementen las operaciones aritméticas, especialmente en bucles intensivos en cálculos. En esta práctica se analiza una técnica clásica de optimización a bajo nivel: el reemplazo de multiplicaciones y divisiones enteras por potencias de dos mediante operaciones de desplazamiento binario. Esta transformación permite, en muchos casos, reducir el coste computacional asociado a estas operaciones, aprovechando que los desplazamientos ($\ll$, $\gg$) se ejecutan generalmente más rápido que las multiplicaciones o divisiones en la mayoría de arquitecturas modernas.

El objetivo principal del estudio es implementar dicha optimización sobre el código proporcionado, evaluar su impacto en el rendimiento y analizar su escalabilidad con respecto al tamaño del problema y al número de repeticiones (`ITER`). Para obtener medidas fiables, se realizan múltiples ejecuciones de cada versión del código (con y sin optimización), compiladas con el nivel de optimización `-O0` para evitar interferencias de otras transformaciones automáticas del compilador. Asimismo, se analiza el código ensamblador generado con el fin de confirmar que la sustitución ha tenido efecto y para entender mejor las diferencias observadas en los tiempos de ejecución.

La práctica se completa con una interpretación de los resultados experimentales y un estudio de cómo influyen factores como la memoria caché y el número de repeticiones sobre la estabilidad y precisión de las medidas de tiempo, aportando una visión más completa sobre los beneficios reales de aplicar esta optimización.

2. Descripción de la técnica

El desplazamiento de bits es una operación que consiste en mover los bits de un número binario hacia la izquierda o hacia la derecha una cantidad determinada de posiciones. Esta técnica es fundamental en programación de bajo nivel y se utiliza con frecuencia en ámbitos como sistemas embebidos, procesamiento gráfico y optimización de rendimiento, ya que permite realizar operaciones rápidas y con control detallado a nivel de bits, según se explica en [1].

Una operación de **desplazamiento a la izquierda** ($\ll$) mueve todos los bits de un número binario hacia la izquierda, rellenando con ceros los bits vacíos del lado derecho. Esta operación equivale a multiplicar el número original por 2^n, donde n es el número de posiciones desplazadas. Por ejemplo, desplazar un número una posición a la izquierda es equivalente a multiplicarlo por 2.

De forma análoga, una operación de **desplazamiento a la derecha** ($\gg$) traslada los bits hacia la derecha. Si el desplazamiento es lógico, los bits vacíos del lado izquierdo se rellenan con ceros; si es aritmético, se conserva el bit de signo en números con representación en complemento a dos. Esta operación es equivalente, en muchos casos, a una división entera por potencias de dos.

El uso de desplazamientos puede reemplazar multiplicaciones o divisiones enteras por potencias de dos, lo que permite reducir el coste computacional de estas operaciones en ciertas arquitecturas. Esto puede dar lugar a algoritmos más rápidos y eficientes, especialmente en aplicaciones críticas para el rendimiento, como el procesamiento gráfico, la criptografía o la indexación de grandes volúmenes de datos.

El código con multiplicaciones y divisiones inicialmente propuesto es:

```
int i, j, m3 = 8, m5 = 32, a = 0, b = 0;
for (j = 0; j < ITER; j++) {
        for (i = 0; i < N; i++) {
                a = i * m3;
                b += a / m5;
        }
}
```

Es claro que `a` representa el valor del índice (sobre el tamaño del problema) multiplicado por $2^3 = 8$, para luego sumar de forma acumulada `b` con `a` dividido entre $2^5 = 32$.

La optimización que se implementará será la siguiente. Un simple análisis llega para ver que:

$$\mathtt{b} = \mathtt{b} + \frac{\mathtt{a}}{2^5} = \mathtt{b} + \frac{i \cdot 2^3}{2^5} = \mathtt{b} + \frac{i}{2^2}. \tag{1}$$

En la versión optimizada del código tenemos en cuenta (1) e implementamos la operación con un desplazamiento de 2 bits a la derecha, como se ha explicado.

```
for (j = 0; j < ITER; j++) {
        for (i = 0; i < N; i++) {
                b += i >> 2;
        }
        a = (N - 1) << 3;
}
```

De manera similar, para intentar ser fiel con el cálculo de `a` en cada uno de los índices sobre `ITER`, terminamos la iteración multiplicando como sigue. El término $\mathtt{a}_i$ se refiere al valor de `a` antes de comenzar la iteración $i+1$ sobre `ITER`.

$$\forall i,\ \mathtt{a}_i = i \cdot 2^3 \implies \mathtt{a}_{N-1} = (N-1) \cdot 2^3. \tag{2}$$

Además de su uso en operaciones aritméticas, el desplazamiento de bits permite manipular directamente bits individuales, lo cual resulta útil para configurar, borrar o alternar banderas dentro de registros binarios. Comprender estas operaciones es esencial para aquellos que deseen profundizar en el funcionamiento interno del *hardware* y en la eficiencia del software a bajo nivel.

Para ambas versiones, se explica a continuación la motivación de los 2 bucles anidados.

2.1. Configuración del experimento

El objetivo del experimento es analizar el comportamiento de la versión con multiplicaciones y divisiones frente a la versión con operaciones de desplazamiento para diversos tamaños del problema. Para esto, se crea un *array* de valores de `N` con un número de datos suficiente (40 valores diferentes) y evitando potencias enteras de 2. El *array* se muestra en (3).

$$\mathtt{N} = \begin{pmatrix} 3000, & 4500, & 7000, & 9500, & 12000, \\ 17000, & 25000, & 34000, & 40000, & 56000, \\ 78000, & 95000, & 110000, & 160000, & 250000, \\ 320000, & 390000, & 570000, & 820000, & 1000000, \\ 1190000, & 1500000, & 1750000, & 1990000, & 2550000, \\ 3190000, & 4200000, & 5100000, & 5900000, & 7800000, \\ 9500000, & 11800000, & 15900000, & 23900000, & 53000000, \\ 95000000, & 199000000, & 383000000, & 959000000, & 1910000000 \end{pmatrix}. \tag{3}$$

Además, resulta importante incluir un bucle externo de número fijo de iteraciones (`ITER`) cuando se estudia el rendimiento de un algoritmo en función del tamaño del problema, especialmente si este es variable, para garantizar condiciones comparables entre ejecuciones con distintos valores de N. Para valores pequeños de `N`, la ejecución del algoritmo es demasiado rápida. Esto puede provocar medidas ruidosas y poco precisas por debajo del umbral de resolución del reloj del sistema, aparte de un dominio del *overhead* de la llamada al sistema o del propio cronómetro (si este no valiese 0).

Sin embargo, para valores grandes de tamaño del problema, el algoritmo tarda más, pero con mucha menor variabilidad relativa. Esto genera datos incomparables y gráficas sesgadas. Se introduce así un bucle externo de `ITER` repeticiones, donde `ITER` se elige dinámicamente para que el producto $\mathtt{ITER} \cdot \mathtt{N}$ sea constante (o aproximadamente constante). Fijando que cada repetición del experimento deba tardar, al menos, una decena de segundos, generamos el vector de `ITER` según (4), que garantiza esta condición.

$$\forall i \in \{1, \ldots, 40\},\ \mathtt{ITER}_i = \left\lfloor \frac{6.400.000.000}{\mathtt{N}_i} \right\rfloor. \tag{4}$$

Esta expresión se traducirá en el código C en la función `generar_iter()`. Para el vector (3), los valores corrrespondientes se reflejan en (5).

$$\mathtt{ITER} = \Big(\mathit{valores} \Big). \tag{5}$$

Para cada valor de `N`, es fundamental ejecutar el algoritmo un número suficiente de veces, denotado como `REPETICIONES`, con el fin de obtener medidas de rendimiento representativas y estables. La variabilidad inherente al sistema —debido a factores como la planificación del sistema operativo, el estado de la caché, o procesos en segundo plano— puede provocar fluctuaciones significativas en los tiempos de ejecución individuales. Al repetir las mediciones y calcular posteriormente estadísticas como la **media**, el **máximo** y el **mínimo**, se obtiene una caracterización más completa del comportamiento del algoritmo. Estas métricas permiten representar en las gráficas no solo la tendencia central, sino también la dispersión y los extremos del rendimiento, ofreciendo una visión más realista y robusta para comparar implementaciones o analizar cuellos de botella. Para este particular, `REPETICIONES` se fija a 20.

3. Beneficios y desventajas esperados

Las técnicas de **reducción por fuerza**, como se ve en 2 , permiten reemplazar operaciones costosas por otras equivalentes pero más eficientes en la máquina objetivo. Estas transformaciones están motiva-

das por el hecho de que ciertas instrucciones del conjunto de la arquitectura son mucho más económicas que otras. Por ejemplo, elevar un número al cuadrado, x^2, puede reemplazarse por la multiplicación $x \times x$, que resulta menos costosa que una llamada a una rutina de exponenciación general.

Un caso particularmente interesante es el uso de **desplazamientos binarios** ($\ll$, $\gg$) como sustitutos de multiplicaciones o divisiones por potencias de dos. Estas operaciones, en principio, deberían ser más rápidas que una multiplicación o división convencional, ya que el *hardware* puede implementarlas mediante simples circuitos de cableado sin necesidad de utilizar una unidad aritmética compleja.

Sin embargo, en la práctica, esta ventaja no siempre se manifiesta de forma significativa. En arquitecturas modernas, las unidades aritméticas están tan optimizadas que las diferencias de coste entre una multiplicación por una constante y un desplazamiento pueden ser mínimas o incluso inexistentes, especialmente cuando el compilador ya realiza estas optimizaciones automáticamente, *aún compilando sin optimizaciones*. Además, el uso de desplazamientos puede introducir ambigüedad o dificultar la legibilidad del código fuente si no se utiliza con criterio.

De acuerdo a 3 , la compilación en GCC con `-O0` reduce el tiempo de compilación y permite que la depuración produzca los resultados esperados. Este es el nivel predeterminado. Con la opción `-O0`, GCC desactiva completamente la mayoría de las pasadas de optimización; estas no se ejecutan ni siquiera si se habilitan explícitamente en la línea de comandos o si aparecen listadas como activadas por defecto.

Por lo tanto, aunque los desplazamientos siguen siendo una herramienta útil en la optimización de bajo nivel —especialmente en sistemas embebidos o compiladores de alto rendimiento—, es importante evaluar su beneficio real en el contexto específico de la arquitectura y el compilador utilizados.

4. Programación y código en ensamblador

La compilación con la opción `-S` de GCC produce códigos fuente del ensamblador de la máquina mediante los que podemos analizar la traducción de las 2 estrategias. Se han ejecutado los códigos en una máquina con las siguientes características:

- Sistema operativo:
- Procesador y número de núcleos:
- Versión de GCC:
- Memoria caché:
- Memoria RAM:

Con el fin de ejecutar las dos versiones en las mismas condiciones, se ha vuelto en cada caso el algoritmo en una estructura de bucles bajo `REPETICIONES` y variables de la forma que sigue:

```
1 int main() {
2         int ITER[TAM_N]; // Array ITER del mismo tamaño que N
3         generar_ITER(ITER); // Generar valores
4
```

```
        // Abrimos en modo escritura
        FILE *file = fopen(nombre_archivo, "w");
        if (!file) { ... }

        // Escribimos la cabecera en el archivo
        fprintf(file, "N\t ... \n");

        // No hay que calentar la caché porque no se usa ningún vector
        // solo se utilizan 3 variables que estarán en la pila

        // Medir overhead, que ha resultado 0, luego se obvia
        gettimeofday(&overhead_start, NULL);
        gettimeofday(&overhead_end, NULL);
        overhead = (overhead_end.tv_sec - overhead_start.tv_sec) +
        (overhead_end.tv_usec - overhead_start.tv_usec) / 1e6;

        // Variables locales
        int i, j, a = 0, b = 0; // y m3, m5

        for (int index = INDEX_INICIO; index < INDEX_FIN; index++) {
                double tiempos[REPETICIONES];
                int N_local = N[index], ITER_local = ITER[index];

                for (int k = 0; k < REPETICIONES; k++) {
                        b=0;
                        gettimeofday(&start_time, NULL);
                        // algoritmo
                        gettimeofday(&end_time, NULL);
                        tiempos[k] = ((end_time.tv_sec - start_time.tv_sec +
                        (end_time.tv_usec - start_time.tv_usec)/1.e6))/
                        ITER_local;
                        fprintf(file, "%d\t%.6f\n", N_local, tiempos[k]);
                }
        }

        fclose(file);
        return 0;
}
```

Los bloques denotados por el comentario `algoritmo` son los que se han descrito en la sección 2. En las dos versiones, el bucle externo de `ITER` se traduce en la directiva `.L10` y el bucle interno sobre `N` en `.L11`.

4.1. Bucles

En el caso de la versión original, los dos bucles anteriores se traducen en lo siguiente.

```
.L11:
```

```
        movl    -384(%rbp), %eax
        imull    -364(%rbp), %eax
        movl    %eax, -356(%rbp)
        movl    -356(%rbp), %eax
        cltd
        idivl    -360(%rbp)
        addl    %eax, -376(%rbp)
        addl    $1, -384(%rbp)
.L10:
        movl    -384(%rbp), %eax
        cmpl    -352(%rbp), %eax
        jl      .L11
        addl    $1, -380(%rbp)
```

Las instrucciones `imull` e `idivl`, que se corresponden con `a = i * m3` y `b += a / m5`, respectivamente, representan multipliciaciones y divisiones explícitas en cada iteración sobre un valor de `N`. Estas 2, teóricamente, son operaciones con un coste alto, aunque se podría ver reducido en computadores modernos.

Para la versión con desplazamientos, los dos bucles anteriores se traducen en lo siguiente.

```
.L11:
        movl    -376(%rbp), %eax
        sarl    $2, %eax
        addl    %eax, -368(%rbp)
        addl    $1, -376(%rbp)
.L10:
        movl    -376(%rbp), %eax
        cmpl    -352(%rbp), %eax
        jl      .L11
        movl    -352(%rbp), %eax
        subl    $1, %eax
        sall    $3, %eax
        movl    %eax, -356(%rbp)
        addl    $1, -372(%rbp)
```

Las instrucciones `sarl` y `sall`, que se corresponden con `i` $\gg$ `2` y `(N-1)` $\ll$ `3`, respectivamente, son operaciones de desplazamiento de bits. Por lo tanto, estas suelen tardar 1 ciclo en cada llamada, lo que las haría en principio más rápidas que las aritméticas de multiplicación y división. A pesar de todo, con `-O0` el compilador no elimina accesos redundantes a memoria, por lo que se podría anular esta ventaja.

4.2. Accesos a memoria

Seguir completando con PDF

4.3. Resumen comparado

5. Resultados obtenidos

5.1. Tiempos de ejecución

5.2. Aceleración

5.3. Tamaños del problema pequeños

6. Conclusiones

Referencias

1. https://wraycastle.com/es/blogs/knowledge-base/bit-shift-calculator#:~:text=El%20desplazamiento%20de%20bits
2. Aho (libro asignatura)
3. -O0 en el manual de GCC: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-O0