Compiladores e Intérpretes

Práctica 1: Generación y optimización de código
Optimización del compilador en un producto de matrices

IGNACIO GARBAYO FERNÁNDEZ

Universidade de Santiago de Compostela

Índice

1. Introducción **2**

2. Descripción de la técnica **2**

2.1. Configuración del experimento 2

3. Opciones de compilación **3**

4. Programación y código en ensamblador **3**

4.1. Arquitectura de compilación y ejecución 3

4.2. Bucles 3

5. Resultados obtenidos **3**

5.1. Tamaño de códigos objeto 4

5.2. Tiempos de ejecución 4

5.3. Aceleración 4

6. Conclusiones **4**

1. Introducción

2. Descripción de la técnica

Para la práctica, se propone estudiar en primera instancia el rendimiento de la siguiente porción de código.

```
1
```

2.1. Configuración del experimento

El objetivo del experimento es analizar el comportamiento de la versión original frente a la versión con *peeling* de 2 iteraciones. Para esto, se crea un *array* de valores de `N` con un número de datos suficiente (39 valores diferentes) y evitando potencias enteras de 2. El *array* se muestra en (1).

$$\mathtt{N} = \begin{pmatrix} 3000, & 4500, & 7000, & 9500, & 12000, \\ 17000, & 25000, & 34000, & 40000, & 56000, \\ 78000, & 95000, & 110000, & 160000, & 250000, \\ 320000, & 390000, & 570000, & 820000, & 1000000, \\ 1190000, & 1500000, & 1750000, & 1990000, & 2550000, \\ 3190000, & 4200000, & 5100000, & 5900000, & 7800000, \\ 9500000, & 11800000, & 15900000, & 23900000, & 53000000, \\ 95000000, & 199000000, & 383000000, & 959000000. & \end{pmatrix}. \tag{1}$$

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 (2), que garantiza esta condición.

$$\forall i \in \{1, \ldots, 39\},\ \mathtt{ITER}_i = \left\lfloor \frac{7.000.000.000}{\mathtt{N}_i} \right\rfloor. \tag{2}$$

Para el vector (1), los valores corrrespondientes se reflejan en (3).

$$\mathtt{ITER} = \begin{pmatrix} 2133333, & 1422222, & 914285, & 673684, & 533333, \\ 376470, & 256000, & 188235, & 160000, & 114285, \\ 82051, & 67368, & 58181, & 40000, & 25600, \\ 20000, & 16410, & 11228, & 7804, & 6400, \\ 5378, & 4266, & 3657, & 3216, & 2509, \\ 2006, & 1523, & 1254, & 1084, & 820, \\ 673, & 542, & 402, & 267, & 120, \\ 67, & 32, & 16, & 6. & \end{pmatrix}. \tag{3}$$

Para cada valor de `N`, es fundamental ejecutar el algoritmo un número suficiente de veces, denotado por `REPS`, con el fin de obtener medidas de rendimiento representativas y estables. La variabilidad inhe-

rente 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, `REPS` se fija a 15.

3. Opciones de compilación

4. Programación y código en ensamblador

4.1. Arquitectura de compilación y ejecución

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 ASUS Zenbook UX425EA 1.0, equipada con el sistema operativo Ubuntu 22.04.3 LTS x86_64. Cuenta con un procesador Intel i7-1165G7 de 4 núcleos físicos de undécima generación, que funciona a una frecuencia de reloj de 2.80 GHz. Las direcciones de memoria ocupan 39 bits físicos y 48 bits virtuales.

Cada núcleo dispone de 2 hilos de procesamiento, resultando en un total de 8 hilos. Dispone de 16 GiB de memoria RAM totales y de 4 niveles de memoria caché: L1d (192 KiB, 4 instancias), L1i (128 KiB, 4 instancias), L2 (5 MiB, 4 instancias) y L3 (12 MiB, 1 instancia). El TDP (*Thermal Design Power*) es configurable entre 12 y 28 W. Se omite la inclusión de otros componentes como la tarjeta gráfica o de sonido porque no afectarán a los resultados de las pruebas. La versión del compilador utilizada es GCC 11.4.0.

Con el fin de ejecutar las dos versiones en las mismas condiciones, el envoltorio de declaraciones y variables de medición previos a las llamadas correspondientes ha sido el que se muestra a continuación.

```
1
```

4.2. Bucles

Vamos a centrarnos en la traducción de los bucles de `algoritmo`. La versión original presenta la estructura siguiente.

```
1
```

Por el contrario, la versión con *peeling*, sigue el esquema que aparece a continuación.

```
1
```

5. Resultados obtenidos

Los experimentos se han realizado teniendo en cuenta los parámetros `N`, `ITER` y `REPS` explicados en la sección 2. Las fases del experimento también se han introducido en la sección 4. En primer lugar, se realiza el calentamiento de la caché descrito previamente. A continuación, se obtiene el *overhead*, medición que tiene como objetivo estimar el tiempo que tarda en ejecutarse la propia medición del tiempo. Esto se aproxima calculando el tiempo que se tarda en usar `gettimeofday()` dos veces seguidas. Con todo, el *overhead* ha resultado ser nulo durante las pruebas.

5.1. Tamaño de códigos objeto

5.2. Tiempos de ejecución

5.3. Aceleración

Una mejor manera de comparar las dos versiones del código es obtener la ganancia de una versión frente a la otra, es decir, la aceleración, que vendrá dada por:

$$ac = \frac{t_{\text{original}}}{t_{\text{peeling}}}. \tag{4}$$

Para cada tamaño de `N`, se consideran los tiempos de ejecución correspondientes a dos versiones distintas de un programa. Sea $T_{\text{OPT}}^{(\texttt{N})}$ y $T_{\text{SIN}}^{(\texttt{N})}$ los conjuntos de tiempos para cada versión. Se definen los cuantiles como $Q_p(X)$, siendo $Q_{0.25}(X)$ el cuantil inferior (primer cuartil) y $Q_{0.75}(X)$ el cuantil superior (tercer cuartil) del conjunto X. A partir de estos cuantiles, se estiman límites robustos para el *speedup* como una franja entre dos cocientes: el inferior, que representa una ganancia conservadora, se calcula como:

$$\text{Speedup}_{\text{mín}} = \frac{Q_{0.25}\left(T_{\text{SIN}}^{(N)}\right)}{Q_{0.75}\left(T_{\text{OPT}}^{(N)}\right)}, \tag{5}$$

y el superior, que representa una ganancia potencial más optimista, como

$$\text{Speedup}_{\text{máx}} = \frac{Q_{0.75}\left(T_{\text{SIN}}^{(N)}\right)}{Q_{0.25}\left(T_{\text{OPT}}^{(N)}\right)}. \tag{6}$$

Esta franja proporciona una estimación del rendimiento relativa entre ambas versiones, mitigando la influencia de valores atípicos.

6. Conclusiones