PRÁCTICA 1: "ANÁLISIS DE PRESTACIONES"

Arquitectura e Ingeniería de Computadores
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Objetivos:

- Aplicar la ley de Amdahl.
- Evaluar y comparar las prestaciones de distintas arquitecturas.

Desarrollo:

Ley de Amdahl.

La ley de Amdahl nos indica que si una aplicación utiliza un cierto componente durante una fracción F de su tiempo de ejecución, la aceleración que se obtendría si se mejorara en un factor S dicho componente viene dada por la siguiente expresión:

$$S' = \frac{1}{1 - F + \frac{F}{S}}$$

Sin embargo, en muchas ocasiones, resulta difícil determinar la fracción F, bien por no disponer del código fuente de las aplicaciones, bien por estar distribuida la utilización del componente a lo largo de todo el tiempo de ejecución del programa. Pues bien, la aplicación "a la inversa" de la Ley de Amdahl permite fácilmente obtener dicha fracción.

Para ello, se necesita ejecutar la aplicación bajo estudio con dos tipos del componente en cuestión cuya mejora local S sea conocida. El cociente de los dos tiempos de ejecución obtenidos es la aceleración global S'. Conocidos S y S', no hay más que despejar F.

Para ilustrar todo esto con un ejemplo, supongamos que se pretende obtener la fracción F del tiempo de ejecución que una aplicación, que realiza operaciones con matrices (`matrix`), emplea en realizar productos escalares. El producto escalar de dos vectores $A = \{a_1, a_2, ..., a_n\}$ y $B = \{b_1, b_2, ..., b_n\}$ de tamaño n se define como,

$$A \cdot B = \{a_1, a_2, ..., a_n\} \cdot \{b_1, b_2, ..., b_n\} = a_1b_1 + a_2b_2 + ... + a_nb_n$$

y es una operación básica, por ejemplo, en la multiplicación de matrices. La idea será realizar dos implementaciones de esta operación (el "componente" bajo estudio), una haciendo uso de las instrucciones estándar del procesador y otra empleando instrucciones del juego SSE (*Streaming SIMD Extensions*).

```
float Scalar(float *s1,
             float *s2,
             int size)
{

    int i;
    float prod = 0.0;

    for(i=0; i<size; i++) {
      prod += s1[i] * s2[i];
    }

    return prod;

} // end Scalar()
```

Figura 1: Producto escalar implementado con instrucciones estandar.

```
float ScalarSSE(float *m1,
                float *m2,
                int size)
{
    float prod = 0.0;
    int i;
    __m128 X, Y, Z;

    Z = _mm_setzero_ps(); /* all to 0.0 */
    for(i=0; i<size; i+=4) {
      X = _mm_load_ps(&m1[i]);
      Y = _mm_load_ps(&m2[i]);
      X = _mm_mul_ps(X, Y);
      Z = _mm_add_ps(X, Z);
    }

    for(i=0; i<4; i++) {
      prod += Z[i];
    }

    return prod;

} // end ScalarSSE()
```

Figura 2: Producto escalar implementado con instrucciones SSE.

Cálculo de la aceleración local *S*

Necesitamos, por tanto, disponer de las implementaciones del componente en cuestión. Las funciones ***Scalar()*** y ***ScalarSSE()*** implementan el producto escalar de dos vectores tal y como se ha comentado en el apartado anterior. La implementación con instrucciones SSE debe ser más rápida ya que dichas instrucciones permiten operar simultáneamente con varios datos, en este caso con hasta 4 valores de coma flotantes. En las figuras 1 y 2 pueden verse las dos implementaciones.

La relación entre la velocidad obtenida por ambas implementaciones será S en la expresión de la Ley de Amdahl. Para obtener esta relación utilizaremos un sencillo programa en lenguaje C que realiza productos escalares dentro de un bucle. La figura 3 muestra el código del programa, denominado `scalar.c`.

Viendo el código vemos que hay dos líneas con el comentario,

```
// A MODIFICAR
```

con estas líneas trabajaremos para obtener diferentes tiempos de ejecución y así estimar el valor de S. Será necesario obtener el tiempo de ejecución para ambas implementaciones (instrucciones estándar t_{std}, instrucciones SSE t_{sse}) así como el tiempo de ejecución debido a la sobrecarga del bucle de medida y la inicialización de los vectores (t_{load}). Restando este valor a los anteriores podremos obtener el tiempo real empleado por el componente que nos interesa.

```
int main(int argc, char * argv[]) {

    int     i;
    int     rep=10;
    int     msize=MSIZE;
    float   fvalue;

    if (argc == 2) {
      rep = atoi(argv[1]);
    } else if (argc == 3) {
      rep = atoi(argv[1]);
      msize = atoi(argv[2]);
    } // end if/else
    fprintf(stderr, "Rep = %d / size = %d\n", rep, msize);

    for(i=0; i<rep; i++) {
      init_vector(vector_in, msize);
      init_vector(vector_in2, msize);
      fvalue = Scalar(vector_in2, vector_in, msize);       // A MODIFICAR
      //fvalue = ScalarSSE(vector_in2, vector_in, msize);  // A MODIFICAR
    } // end for

    exit(0);

} // end main()
```

Figura 3: Programa `scalar.c` empleado para evaluar la relación de las dos implementaciones del producto escalar.

Para obtener el tiempo t_{std} debemos comprobar en el programa `scalar.c` que sólo la línea de la función ***Scalar()*** está sin comentar,

```
  ...

fvalue = Scalar(vector_in2, vector_in, msize);       // A MODIFICAR
//fvalue = ScalarSSE(vector_in2, vector_in, msize);  // A MODIFICAR
  ...
```

A continuación compilaremos el programa desde una consola de linux,

```
gcc -O0 -o scalar-std scalar.c
```

lo ejecutaremos y mediremos el tiempo usando la orden `time`.

ESTAS INSTRUCCIONES **NO** SON NECESARIAS SI SE REALIZA LA PRÁCTICA DESDE UN ENTORNO VIRTUAL (Polilabs, VirtualBox, VMware, ...)

No obstante antes de ejecutar el programa tenemos que fijar la velocidad del procesador para que las medidas sean uniformes. Por defecto los cores del procesador están en modo *ondemand* y la velocidad de los mismos varía según la carga. Esto no es adecuado para tomar medidas. Para cambiar el modo de los cores empleamos la orden `cpufreq-set`, pudiendo ver el modo en que está con la orden `cpufreq-info`. Fijaremos el modo de cada core a *performance*. El parametro `-c #` identifica el core que vamos a fijar:

```
cpufreq-set -c 0 -g performance
cpufreq-set -c 1 -g performance
cpufreq-set -c 2 -g performance
cpufreq-set -c 3 -g performance
```

lo verificamos con,

```
cpufreq-info
```

o,

```
cat /proc/cpuinfo
```

y ya podemos proceder con las ejecuciones.

IMPORTANTE: os recordamos que en linux el directorio actual no está en el PATH, obligándonos a colocar *./* antes de cada programa que queramos ejecutar desde la consola. Para evitar esto hay que ejecutar esta línea en la consola en la que estéis trabajando, o añadirla al final del fichero $HOME/.bashrc,

```
export PATH=./:$PATH
```

Tal y como hemos dicho ejecutamos el programa `scalar-std`,

```
time scalar-std 100000 1024
```

los parámetros simplemente indican que queremos hacer 100000 repeticiones del bucle y los vectores tendrán un tamaño de 1024. Anotaremos el tiempo empleado por la tarea (*user+system*) como t_{std}.

Ahora editaremos de nuevo el program `scalar.c` para obtener el tiempo t_{sse}. Lo haremos comentando la línea de la función ***Scalar()*** y descomentando la línea de la función ***ScalarSSE()***,

```
  ...

//fvalue = Scalar(vector_in2, vector_in, msize);   // A MODIFICAR
fvalue = ScalarSSE(vector_in2, vector_in, msize);  // A MODIFICAR
  ...
```

compilaremos y ejecutaremos de nuevo el programa,

```
gcc -O0 -msse -o scalar-sse scalar.c
time scalar-sse 100000 1024
```

y anotaremos el tiempo empleado como t_{sse}. Observar que al compilar hemos añadido la directiva ***-msse*** para que el compilador genere código utilizando las instrucciones SSE.

Finalmente editaremos de nuevo el program `scalar.c` y comentaremos las líneas que hacen la llamada al producto escalar. De este modo calcularemos la sobrecarga debida al bucle e inicializaciones,

```
  ...

//fvalue = Scalar(vector_in2, vector_in, msize);   // A MODIFICAR
//fvalue = ScalarSSE(vector_in2, vector_in, msize);  // A MODIFICAR
  ...
```

compilaremos y ejecutaremos una vez más el programa,

```
gcc -O0 -msse -o scalar-load scalar.c
time scalar-load 100000 1024
```

anotando el tiempo como t_{load}. Con esta información ya podemos calcular la estimación de S como,

$$S = \frac{t_{std} - t_{load}}{t_{sse} - t_{load}}$$

Una vez obtenida la relación entre las dos implementaciones (S), mediremos el tiempo de ejecución de la aplicación `matrix` bajo estudio con cada una de las implementaciones. La relación entre los tiempos de ejecución será la aceleración global que la aplicación alcanza al mejorar el producto escalar. Esta relación es S' en la Ley de Amdahl. Por lo tanto, sustituyendo S y S' en la ley de Amdahl, es posible despejar F para obtener la fracción del tiempo que la aplicación `matrix` hace uso del producto escalar.

Cálculo de la fracción de tiempo local *F*

Vamos a aplicar este proceso para obtener la fracción de tiempo que la aplicación `matrix` emplea en realizar productos escalares.

Necesitaremos el código fuente de dicha aplicación para poder obtener los tiempos que nos hacen falta,

$$S' = \frac{t_{mat-std}}{t_{mat-sse}}$$

Para obtener $t_{mat-std}$ editaremos el archivo `matrix.c` y buscaremos al principio del mismo la definición de la macro **__SCALAR_PROD()**,

```
  ...
#define __SCALAR_PROD(v1, v2, s)  Scalar(v1, v2, s);
//#define __SCALAR_PROD(v1, v2, s)  ScalarSSE(v1, v2, s);
//#define __SCALAR_PROD(v1, v2, s)
  ...
```

debemos dejar sin comentar la versión estandar tal y como se muestra en el fragmento de código anterior. De este modo la aplicación `matrix` hará uso de esta implementación del producto escalar.

compilamos y ejecutamos,

```
gcc -O0 -msse -o matrix-std matrix.c -lm
time matrix-std 1 1024
```

en este caso sólo realizaremos una repetición y usaremos matrices también de 1024x1024, para que el tamaño del vector sea el mismo que en las mediciones anteriores. Anotaremos el tiempo como $t_{mat-std}$.

A continuación editaremos de nuevo el archivo `matrix.c`, pero seleccionaremos la operación de producto escalar que emplea las instruccciones SSE,

```
  ...
//#define __SCALAR_PROD(v1, v2, s)  Scalar(v1, v2, s);
#define __SCALAR_PROD(v1, v2, s)  ScalarSSE(v1, v2, s);
//#define __SCALAR_PROD(v1, v2, s)
  ...
```

compilaremos y ejecutaremos de nuevo,

```
gcc -O0 -msse -o matrix-sse matrix.c -lm
time matrix-sse 1 1024
```

anotando el tiempo como $t_{mat-sse}$.

$\Rightarrow$ Con los datos obtenidos, calcula el porcentaje de tiempo F que esta aplicación emplea en realizar productos escalares.

Cálculo experimental de la fracción de tiempo local (F_{exp})

Aunque en general esto no es posible, dada la naturaleza del código de `matrix` y del componente que estamos evaluando, es posible obtener experimentalmente la fracción del tiempo que esta aplicación emplea en realizar productos escalares. Para ello simplemente tenemos que definir la macro del producto escalar como vacía,

```
  ...
//#define __SCALAR_PROD(v1, v2, s)  Scalar(v1, v2, s);
//#define __SCALAR_PROD(v1, v2, s)  ScalarSSE(v1, v2, s);
#define __SCALAR_PROD(v1, v2, s)
  ...
```

para poder estimar cuanto tiempo dedica el programa al resto de tareas. A este tiempo lo denominaremos $t_{mat-res}$ y nos permitirá calcular la fracción de tiempo experimentalmente (F_{exp}) mediante la siguiente fórmula,

$$F_{exp} = \frac{t_{mat-std} - t_{mat-res}}{t_{mat-std}}$$

compilando y ejecutando la nueva versión del programa,

```
gcc -O0 -msse -o matrix-res matrix.c -lm
time matrix-res 1 1024
```

obtenemos $t_{mat-res}$.

$\Rightarrow$ Calcula experimentalmente la F_{exp} empleada por el producto escalar en `matrix`. Compárala con la obtenida mediante la ley de Amdahl.

Análisis de las prestaciones de las arquitecturas.

En este apartado se van a medir las prestaciones de los computadores del laboratorio empleando los siguientes programas:

- dos *benchmarks* sintéticos (**dhrystone**, para aritmética entera, y **whetstone**, para aritmética en coma flotante)
- dos aplicaciones reales: el compilador del lenguaje de programación C **gcc**, que sólo utiliza aritmética entera, y la aplicación **xv** de procesamiento de imagen.

Para ello, ejecutaremos los programas, midiendo los tiempos de ejecución.

- **dhrystone** (10.000.000 iteraciones):

  ```
  time dhrystone
  ```

  Indicarle que debe de realizar 10.000.000 de iteraciones cuando el programa lo pregunte.

  Anotaremos el tiempo de ejecución $T_{dhrystone}$ del programa.

- **whetstone** (10.000 iteraciones):

  En este caso teclear:

  ```
  time whetstone 10000
  ```

  Anotaremos el tiempo de ejecución T_{whet-h}.

- Compilador del lenguaje C, compilando una aplicación:

  Compilaremos la aplicación **xv**. Para ello, suponiendo que los fuentes están ubicados en la carpeta `xv-310a` de nuestro directorio, teclearemos:

  ```
  cd xv-310a/
  make clean
  time make
  ```

  Anotar el tiempo de ejecución T_{gcc}.

- Aplicación *xv*:

  Ejecutar la aplicación **xv** que acabamos de compilar, abriendo un fichero de imagen:

  ```
  cd ..
  time xv-310a/xv -wait 5 mundo.jpg
  ```

  Anotar el tiempo de ejecución T_{xv}.

La tabla siguiente muestra los resultados obtenidos (tiempos de ejecución en s) tras ejecutar dichas aplicaciones en tres máquinas distintas. La máquina C se corresponde con la que estamos trabajando en el laboratorio:

Programa/Máquina	A	B	C
dhrystone	5	18	
whetstone	2.5	10	
gcc	40	130	
xv	4.5	15	

$\Rightarrow$ Comparar las prestaciones de los tres computadores empleando tres formas distintas. La primera considerará cada uno de los programas de un modo aislado, la segunda empleará la media aritmética de los tiempos de ejecución, y la tercera la media geométrica de los tiempos de ejecución normalizados a la máquina B.