Lab. 1: "PERFORMANCE ANALYSIS"

Computer Architecture and Engineering (3^{rd} course)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Objectives:

- Using of the Amdahl's law.
- Evaluating and comparing the performance of different architectures.

Development:

Amdahl's law.

Amdahl's law quantifies the maximum expected overall system speed-up (S') when a part of that system, used during a fraction (F) of its execution time, is enhanced with a (local) speed-up of S. This is the expression relating S', F and S :

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

However, it is sometimes hard to compute F, either because applications' source code is not available or because the use of the considered component is distributed all along the execution time of applications. In such cases, F can be obtained using the Amdahl's law in a "reverse way" .

Doing this encompasses with the execution of the application under study on two variants of the considered component, whose speed-up (S) must be known. The quotient of both execution times is the global speed-up S'. Knowing S and S', it is only a matter of finding the value of F.

Let us introduce an example to illustrate these ideas. Assume that the goal is to obtain the fraction of time F employed by some application. The application makes computations with matrices (`matrix`), in order to compute scalar products. The scalar product of two vector $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_n\}$ of size n can be defined as,

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

this is a basic operation in matrix multiplication. The idea here is to test two implementations of this operation, one relying on scalar instructions and another using SSE (*Streaming SIMD Extensions*) instructions.

```
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()
```

Figure 1: Scalar product implemented with standard instructions.

```
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()
```

Figure 2: Scalar product implemented with SSE instructions.

Computing the local speed-up *S*

The functions ***Scalar()*** and ***ScalarSSE()*** implement the two versions of the scalar product. The implementation with SSE instructions should be faster because these instructions can operate simultaneously with different data. In our case, it handles 4 floating point numbers at the same time. Both scalar- and SSE-based implementations can be checked in Figure 1 and 2 respectively.

The quotient between the execution time of these functions is the S appearing in the Amdahl's law formula. In order to exercise the code of considered functions, a simple program in C is used. Figure 3 shows the code of this program, also available in file `scalar.c`.

Looking at the code we can see that two lines have the following comment at the end,

```
\\ MODIFY
```

we will work with these lines in order to obtain de execution times. We need to obtain the execution time with both implementations (t_{std} for the implementation with standard instructions, t_{sse} for the implementation with SSE instructions), and the overload due to the loop and initializations (t_{load}). Then, the execution time of each implementation of the scalar product can be easily deduced by computing $t_{std} - t_{load}$ on one hand, and to $t_{sse} - t_{load}$ on the other hand.

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

    int     i;
    time_t  start, stop;
    double  avg_time;
    double  cur_time;
    int     rep=10;
    int     msize=MSIZE;
    float   fvalue;
    char   *cmd;

    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);       // MODIFY
      //fvalue = ScalarSSE(vector_in2, vector_in, msize);  // MODIFY
    } // end for

    exit(0);

} // end main()
```

Figure 3: Program `scalar.c` employed to obtain the execution time of both implementations.

To obtain t_{std} we have to check that the file `scalar.c` have the line that call the function *textbfScalar()* uncommented and the line invoking the fuction *textbfScalarSSE()* commented.

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

Then, the program can be compiled using the following command:

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

and then executing it and reading the time given by the `time` command. Nevertheless we have to set a fixed processor speed before executing it in order to obtain meaningful measures. By default processor cores speed is configured as in mode *ondemand*. Then the cores speed will vary depending on the processor load. To set a fixed speed we employ the `cpufreq-set` command, and we can inspect the current mode by mean of the `cpufreq-info` command. We will set the processor speed to 3.3Ghz by means of. The parameter `-c #` allows selecting the core we will fix,

```
cpufreq-set -c 0 -f 3.3GHz
cpufreq-set -c 1 -f 3.3GHz
cpufreq-set -c 2 -f 3.3GHz
cpufreq-set -c 3 -f 3.3GHz
```

and we will verify with,

```
cpufreq-info
```

or,

```
cat /proc/cpuinfo
```

now we can run the program.

NOTE: in linux the current directory is NOT in the PATH by default, then we have to add ./ previous to every local program in order to execute it. To avoid this we can run or add to the $HOME/.basrc file the following line:

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

Now we can run the program `scalar-std`,

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

The parameters in the command simply mean that we want the loop to iterate 100000 times working on vectors of 1024 components. Write down the time employed by the program (*user+system*) as t_{std}.

Now we edit again the file `scalar.c` and we comment the line that call the function ***Scalar()*** and we uncomment the line containing the function ***ScalarSSE()***,

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

we compile and run the program again to obtain t_{sse},

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

write down the new time. Compiler flag ***-msse*** is required to allow the use of SSE instructions.

Finally we edit again the file `scalar.c` and we comment both lines containing scalar product calls. This way we can estimate the overload due to the loop and initializations.

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

we compile and run the new program,

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

writing down the new time as t_{load}. With the obtained data we can calculate S as,

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

Once obtained S, the local speed-up, we will calculate the execution time of the application `matrix` with both implementations. The relation between resulting execution times will give us the global speed-up (S'). With S and S' we can use the Amdahl law to find the local fraction of time (F) that the scalar product is used originally in the application `matrix`.

Computation of the local fraction of time *F*

We will apply the previous described process to obtain the fraction of time that the scalar product is executed in application `matrix`.

We need the source code of the application to obtain the execution time with both implementations and to obtain S',

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

To get $t_{mat-std}$ we have to edit the file `matrix.c` and lool for the macro **__SCALAR_PROD()** at the very beginning of the file.

```
  ...
#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)
  ...
```

in this experiment only the standard version have to be uncomment as seen in the previous code. This way the application `matrix` will use this implementation of the scalar product throughout the code.

compile and execute,

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

only one iteration is performed and matrices are of size of $1024x1024$ in order the size of the vectors to be the same as in the previous experiments. Write down the time as $t_{mat-std}$.

Next edit again the file `matrix.c` and let only uncommented the line that defines the macro that use the scalar product implementation using SSE as seen in the following code,

```
  ...
//#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)
  ...
```

Now, compile and execute,

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

write down the time as $t_{mat-sse}$.

$\Rightarrow$ With the data obtained, compute the local fraction of time that the application `matrix` use the scalar product.

Experimental local fraction of time computation (F_{exp})

Although this is not always possible, due to the kind of code of the application `matrix` and the component under study, we can compute experimentally the fraction of time. To do that we have only to define the macro that define the scalar product as empty.

```
  ...
//#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)
  ...
```

this way we can obtain the time that the application employs in the other tasks. We will refer to this time as $t_{mat-res}$ and will help us to compute experimentally the fraction of time (F_{exp}) by means of the formula,

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

compile and execute this new version,

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

and write down the value $t_{mat-res}$.

$\Rightarrow$ Compute F_{exp}. Compare the experimental value and the value obtained by means of the Amdahl law.

Performance analysis of architectures.

This section focuses on the measure of the performance of lab computers using the following programs:

- two synthetic *benchmarks* (**dhrystone**, for integer arithmetic and **whetstone**, for floating point arithmetic)
- two real applications: the C language compiler **gcc**, that only uses integer arithmetic and the **xv** applications, that processes images.

To cope with the aforementioned goals, programs will be executed and their execution times will be measured using the `time` order:

- **dhrystone** (10.000.000 iterations):

  ```
  time dhrystone
  ```

  Indicate the 10.000.000 iterations that must be performed.

  Note the execution time of the program $T_{dhrystone}$.

- **whetstone** (10.000 iterations):

  Now we type:

  ```
  time whet-h 10000
  ```

  Note the execution time T_{whet-h} and the other data provided by the program.

- C language compiler, compiling the *xv* program

  Let's compile the application xv. Assume source code is located in our directory at folder `xv-310a/` . Type the following orders:

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

  Note the execution time T_{gcc}.

- *xv* application

  Now we execute the **xv** application that have just compile:

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

  Note the execution time T_{xv}.

The following table shows the data (execution times in seconds) obtained from the execution of the application in three different machines. Machine C is the one where we are working on in the lab:

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

$\Rightarrow$ Compare the performance of the three computers attending to three different measures. The first one will consider each program in an isolated way. The second one will use the arithmetic mean of executions times, and the third one will apply the geometric mean of execution times normalized to machine B.