Laboratory #1: “Performance Analysis”

Computer Architecture and Engineering (AIC)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Objectives:

- Computing the fraction of time of a program component using the Amdahl's law.
- Evaluating and comparing the performance of different architectures.

Development:

Amdahl's law.

Amdahl's law quantifies the maximum global speed-up (S') that can be expected from a system when a part of that system, used during a fraction (F) of its execution time, is accelerated with a (local) speed-up of S. The following expression relates S', F and S :

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

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

Doing this requires executing the target application for two variants of the considered component, whose local speed-up (S) must be known. The division of both execution times is the global speed-up S'. Once S and S' are known, it only remains 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. Our application `matrix` operates with matrices, computing certain scalar products. The scalar product of two vectors, $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_n\}$, both of size n, is defined as

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

This is a basic kernel for matrix multiplication. The idea is to implement various version of this operation (the “component” under study). One version will carry out the operation using standard scalar instructions, while the other will use the SIMD instructions provided by extensions SSE, AVX or AVX512 existing in the lab processor, an Intel i5-11600K 11th generation, with Sunny Cover microarchitecture. **It is important to note that all these extensions may not be supported by older processors**.

Computing the local speed-up S

Functions ***Scalar()***, ***ScalarSSE()***, ***ScalarAVX()*** and ***ScalarAVX512()*** implement the scalar product if two vectors. The implementation with SIMD instructions must be faster than the scalar implementation since SIMD instructions operate simultaneously with several data items. In particular, the SSE implementation simultaneously handles 4 floating point numbers, the AVX 8 and the AVX512 16. Figures 1, 2, 3 and 4 show implementations with the standard (scalar) instructions and the various SIMD extensions under consideration.

The relation between the execution time of the scalar version and each one of the SIMD functions is the factor S in the formula for Amdahl's law. In order to exercise the code of these functions, we use a simple program in C, given in Figure 5 and also available in the file `scalar.c`.

Inspecting the code we can see several macro directives controlling the version of function $Scalar()$ that is compile and used. Different execution times will be obtained from these various versions in order to estimate the value of S in each case. To do so, the execution time of the standar version (t_{std}) and the SIMD versions ($t_{sse}, t_{avx}, t_{avx512}$) required. It is also necessary to estimate the overhead imposed by the loop and vector initialization (t_{load}) by eliminating the scalar product from the code. Computing the difference between the different execution times and t_{load} one can obtain the execution time really devoted to compute the scalar product of vectors.

In order to obtain the afoementioned execution times, different executable versions of the program must be generated through the linux console. To do so, the compiler must be instructed with the macro to use (-Dxxx). The option -mxxx activates the use of the SIMD extension in the compiler. **By default, these extensions are not active in the compiler to avoid compatibility problems with old processors**.

```
gcc -O0 -o scalar-std scalar.c -DSCALAR
gcc -O0 -o scalar-sse scalar.c -DSSE -msse
gcc -O0 -o scalar-avx scalar.c -DAVX -mavx
gcc -O0 -o scalar-avx512 scalar.c -DAVX512 -mavx512f
gcc -O0 -o scalar-load scalar.c -DLOAD
```

these commands must be executed to measure execution times using the `time` command.**BEFORE PROCEEDING READ THE FOLLOWING PARAGRAPH CAREFULLY**:

THE NEXT INSTRUCTIONS **SHOULD BE SKIPPED** IF THIS LAB ASSIGNMENT IS CARRIED OUT IN A VIRTUAL ENVIRONMENT (Polilabs, VirtualBox, VMware, ...)

We should fix the processor speed before executing the test in order to obtain accurate timing results. The reason is that, by default, the speed of the processor cores is set to *ondemand* mode. With this configuration, the cores vary their frequency depending on the processor load. To set a different mode we can employ the command `cpufreq-set`; also, we can inspect the current mode by mean of the command `cpufreq-info`. In our case, we will set the processor mode to *performace*. The parameter `-c #` has to be used in order to select to which core this command applies. For example, we can run

```
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 with standard (scalar) instuctions

```
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: With SSE instructions

```
float ScalarAVX(float *m1,
                float *m2,
                int size)
{
    float prod = 0.0;
    int i;
    __m256 X, Y, Z;
    Z = _mm256_setzero_ps(); // all 0.0
    for(i=0; i<size; i+=8) {
      X = _mm256_load_ps(&m1[i]);
      Y = _mm256_load_ps(&m2[i]);
      X = _mm256_mul_ps(X, Y);
      Z = _mm256_add_ps(X, Z);
    }
    for(i=0; i<8; i++) {
      prod += Z[i];
    }
    return prod;
} // end ScalarAVX()
```

Figure 3: With AVX instructions

```
float ScalarAVX512(float *m1,
                   float *m2,
                   int size)
{
    float prod = 0.0;
    int i;
    __m512 X, Y, Z;
    Z = _mm512_setzero_ps(); // all 0.0
    for(i=0; i<size; i+=16) {
      X = _mm512_load_ps(&m1[i]);
      Y = _mm512_load_ps(&m2[i]);
      X = _mm512_mul_ps(X, Y);
      Z = _mm512_add_ps(X, Z);
    }
    for(i=0; i<16; i++) {
      prod += Z[i];
    }
    return prod;
} // end ScalarAVX512()
```

Figure 4: With AVX512 instructions

```
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]);
      msize = (msize > MSIZE) ? MSIZE : msize;
    } // end if/else

   fprintf(stderr, "\n");

   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);
#if defined(LOAD)
      // DO NOTHING
#elif defined(SSE)
      fvalue = ScalarSSE(vector_in2, vector_in, msize);
#elif defined(AVX)
      fvalue = ScalarAVX(vector_in2, vector_in, msize);
#elif defined(AVX512)
      fvalue = ScalarAVX512(vector_in2, vector_in, msize);
#else
      fvalue = Scalar(vector_in2, vector_in, msize);
#endif
    } // end for

    exit(0);

} // end main()
```

Figure 5: Code `scalar.c` employed to obtain the execution time of the various implementations of the scalar product under study

```
for i in $(seq 0 11)
do
cpufreq-set -c $i -g performance
done
```

and then verify the change with:

```
cpufreq-info
```

or

```
cat /proc/cpuinfo
```

Now, we are ready to run the program.

IMPORTANT: in Linux, the PATH environment variable does not include the current directory. Therefore, we will have to add the "prefix" ./ to any local program in order to execute it. To avoid this, we can include the following line in the $HOME/.bashrc file:

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

Now we can run the different versions of the program `scalar-`xxx as follows:

```
time scalar-xxx 1000000 1024
```

command parameters specify that the loop has to iterate $1.000.000$ times and the program operates on vectors with 1024 components. Take note of the time taken by each task (**remember that only *user+system* times must be added**) as t_{xxx} and estimate S_{xxx} as previously explained,

$$S_{xxx} = \frac{t_{std} - t_{load}}{t_{xxx} - t_{load}}$$

Attending to results, complete the following table:

	t	S
$load$	3.783s	—
std	5.665s	—
sse	4.740s	1.966
avx	4.286s	3.742
$avx512$	4.070s	6.557

⇒ What do you observe? Are there any differences among the various versions of the program? Are results filling expectations? Which is the version reporting the best performance? Performance increases as we go down in the table. Different versions have different speedups. The best performance is provided by avx512.

Select the standard version (std) and the SIMD version reporting the best performance($best$) to continue with the experiments regarding the Amdahl's law. Let's consider the application `matrix`. This application computes a matrix product using the scalar product several times.

Measure the execution time of the standard and the selected SIMD version of the application. Relation between execution times will be the global speedup reached using an improved version of the scalar product. This global speedup is S' in the Amdahl's law. As a result, replacing S and S' in the Amdahl's law it is possible to deduce F and estimate the fraction of time devoted by application `matrix` to compute the standard scalar product.

Computation of the local fraction of time *F*

Let's apply the process previously explained to obtain the fraction of time that is devoted in application `matrix` to carry out scalar products in its standard version.

We must recompile the source code of the application to obtain the required times,

$$S' = \frac{t_{Mstd}}{t_{Mbest}}$$

in this case we must use the following compilation commands in the console to generate the required executables,

```
gcc -O0 -o matrix-std matrix.c -lm -DSCALAR
```

```
gcc -O0 -o matrix-best matrix.c -lm -DYYY -mxxx
```

and obtain t_{Mstd} y t_{Mbest} using the command `time`,

```
time matrix-std 1 1024
```

```
time matrix-best 1 1024
```

in this case only one repetition is required and we will use matrixes of 1024x1024 elements. In this way, the size of the vector will be the same than in previous measurements. Take not of the times, the value of S′ (global speedup in this section) and S (local speedup in previous section),

$matrix$	t	S′	S
std	2.374s	—	—
$best$	0.690s	3.441	6.557

⇒ With the resulting data, compute the percentage of time F in which the application is computing scalar products. 83.7% (F = 0.837)

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

Although it is not possible in general, considering the code in `matrix` and the component under evaluation, it is possible to experimentally estimate the fraction of time devoted by the application to scalar product computation. The code must be recompiled with the value of macro $LOAD$ to eliminate all scalar products and estimate the time devoted to the rest of tasks. We will refer to this time as t_{Mres}. It will be used in our experimental estimation of the fraction of time F_{exp}) as follows,

$$F_{exp} = \frac{t_{Mstd} - t_{Mres}}{t_{Mstd}}$$

compiling and executing the new version of the program,

```
gcc -O0 -o matrix-load matrix.c -lm -DLOAD
```

we obtain t_{Mres} as previously carried out.

⇒ Compute F_{exp}. Compare the experimental value and the value obtained by means of the Amdahl's law.

F = 0.832

Performance analysis: obtention of required measurements

This section measures the performance of the computers in the laboratory 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**, which only involves integer arithmetic, and the application **xv**, for image processing.

To accomplish the aforementioned goal, the programs need to be executed and their execution times will be measured using the `time` command:

- **dhrystone** (100.000.000 iterations). Execute

  ```
  time dhrystone
  ```

  Annotate the execution time of the program $T_{dhrystone}$. 2.922s

- **whetstone** (10.000 iterations):

  Now we type:

  ```
  time whetstone 10000
  ```

  Annotate the execution time T_{whet-h}. 0.108s

- C language compiler, compiling the *xv* program

  Let us now compile the application `xv`. Assuming the source code is located in folder `xv-310a/` inside the current directory, type the following commands:

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

  Annotate the execution time T_{gcc}. 7.218s

- *xv* application

  Finally, in order to execute the application **xv** , we just have use the command:

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

  Annotate the execution time T_{xv}. 0.556s

Performance analysis: comparison of results

Once all required measurements available, it is time to compare results to establish which machine provides better performance.

Table 1 shows the (execution times in seconds obtained from the execution of the 4 considered applications in three different machines. It is assumed that results for machines A and B are already available. Column C references to the machine where the lab being carried out. So, this column must be completed with the measurements previously obtained from the machine.

Program/Machine	A	B	C
dhrystone	5	18	2.922
whetstone	2.5	10	0.108
gcc	40	130	7.218
xv	4.5	15	0.556
Arithmetic mean	13	43.25	2.701

Table 1: Execution times of applications and arithmetic mean

Considering the execution times for each application on each of the 3 considered machines, a performance comparison is possible. This comparison can rely on two classical metrics, the arithmetic mean of the execution times, the geometric mean of individual applications wrt machine B (the geometric mean of the speedups of individual applications wrt machine B). The arithmetic mean will be included in the last row of table 1 and the geometric mean in table 2. Due to the big volume of data typically used in performance analysis, it is common to rely on the use of data sheets or specific scripts. It is adviced to use a data sheet and the function PROMEDIO (AVERAGE) in order to obtain the average mean , and MEDIA.GEOM (GEOMEAN) for computing the geometric mean.

Program/Machine	A	B	C
dhrystone	3.6	1	6.16
whetstone	4	1	92.593
gcc	3.25	1	18.011
xv	3.333	1	26.98
Geometric mean of speed-ups wrt B	3.534	1	22.94

Table 2: Geometric mean of execution times normalized to machine B. Put into the cells the speedup wrt B.

Additional questions:

- Which is the speedup of machine A wrt machine B considering the arithmetic mean?
  ....
- Which is the speedup of machine A wrt B considering the geometric mean of speedups?
  ....