Task 3: Cache and Performance

NOTE: It is important to read the complete description of each exercise before starting with the implementation.

The goal of this task is to understand how caches can improve the performance of the processor. We will use the *valgrind* tool [1], specifically *cachegrind* [2] and *callgrind* [3], to configure different types of caches and execute a set of programs, to compare performances. The Linux operative system installed on laboratory computers includes all required tools.

NOTE: Although you can use your computer for implementing and testing the code / scripts, the numerical values must be those obtained using the ARQO cluster.

If you do not have the valgrind tool framework on your personal computer, you can install it by running the following command as root (assuming your Linux distribution is Ubuntu):

```
> (sudo) apt-get install valgrind
```

If you do not have GNUplot on your personal computer, you can install it by running the following command as root (assuming your Linux distribution is Ubuntu):

```
> (sudo) apt-get install gnuplot
```

If you do not have lstopo on your personal computer, you can install it by running the following command as root (assuming your Linux distribution is Ubuntu):

```
> (sudo) apt-get install hwloc
```

MATERIAL

A "`.zip`" file is provided it and contains all necessary files to complete this task:

- `arqo3.c` – functions to manage floating point matrixes.
- `arqo3.h` – headers of the functions to manage floating point matrixes.
- `slow.c` – program to calculate the addition of all elements in a matrix.
- `fast.c` – program to calculi the addition of all elements in a matrix (optimal implementation).
- `Makefile` – file to compile all programs.
- `slow_fast_time.sh` – fichero que contiene un script Bash de ejemplo para ejecutar los programas slow y fast.

IMPORTANT: the script `slow_fast_time.sh` does NOT generate the results and charts requested in this task. If you decide to use it, you must modify it according to what is requested in this task. After compiling the fast and slow programs, you can launch the script by entering the following command:

```
> source slow_fast_time.sh
```

IMPORTANT

A number P, necessary for all experiments, can be calculated as P = (team_number mod 7). Examples:

- Team number PT08: P = (8 mod 7) = 1
- Team number PM13: P = (13 mod 7) = 6

In this task is necessary to generate several charts. All charts must include a description, units, etc. per axis. In addition, a legend must be included when the chart shows more than one curve.

RECOMMENDATION:

Although it is not mandatory, students are strongly encouraged to use Bash and GNUplot scripts similar to those presented in class to launch the executions, get the results, and generate the charts. Using such scripts will save you time running tests, since a well-designed script can execute the tests in unattended manner, and in the sequence you want. Likewise, the generation of GNUplot scripts will allow you to quickly generate the charts very easily.
In case students submit the scripts, they have used (and they work properly) together with the memory and the results files, they will be valued positively in the practice.
If you are new to bash scripting and want to learn the basics, it is recommended that you do the simple tutorials referenced in [4] and [5]. The time you invest in them will save your time for this practice. Likewise, if you have no prior knowledge of GNUplot and want to acquire the basics, it is recommended that you do the simple tutorial referenced in [6].

Exercise 0: System cache (1 p)

Obtain the cache configuration of the Linux workstation using the following commands:

```
> cat /proc/cpuinfo
```

```
> dmidecode
```

NOTE: This command requires root permissions.

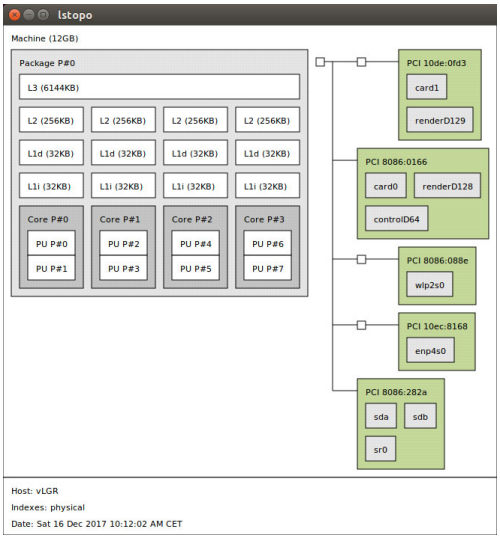
```
> getconf -a | grep -i cache
```

Explain the information you obtained for the system cache. It is important to identify the different cache levels, if there are different caches for data and instructions, the types of the caches, etc.

You can also get additional information by running:

```
> lstopo
```
(remember to install it: `apt-get install hwloc`)

It shows the complete architecture of the microprocessor:



Exercise 1: Cache and Performance (3 p)

Check experimentally how the data access pattern affects the performance. For that, we will use the programs "slow" and "fast" provided with this task.

An example of program execution is:

```
> ./slow 1000
Word size: 64 bits
Execution time: 0.007926
Total: 5000066.116831
```

These programs print the word size, necessary to calculate the size of the memory used by the matrixes, the execution time of the addition of all elements in matrix (the size of the matrix is the parameter of the program, 1000 in this example, so 1000x1000 elements of 8 bytes = 8.000.000 bytes), and finally, the result of the addition.

The matrixes are generated using random numbers. Using the same seed and the same size, the result should be the same for all executions. However, each program could obtain different results because floating-point operations are not commutative (normalization and rounding issues).

Also note that if you run the program multiple times, its execution time will vary. It is for this reason that performance tests must be carried out multiple times and interleaved as indicated in the following table. There are statistical techniques to determine the necessary number of repetitions of the experiments, but, since this type of study is not the objective of this subject, the student have to to determine experimentally (by carrying out multiple tests) what number of repetitions is adequate. The results are adequate when the resulting charts do not present "large" peaks in the measurements.

Interleaved executions (**Correct**)	Non Interleaved executions (**Incorrect**)
`./slow N1`	`./slow N1`
`./slow N2`	`./slow N1`
`./fast N1`	`./slow N2`
`./fast N2`	`./slow N2`
`./slow N1`	`./fast N1`
`./slow N2`	`./fast N1`
`./fast N1`	`./fast N2`
`./fast N2`	`./fast N2`

Tasks to do:

1) Obtain the execution time for both programs ("fast" and "slow") using NxN matrix, when N is a value from 1024 y 16384 in increments of 1024. You will have to take results for all runs multiple times (at least 10) and interleaved, as indicated above, and calculate the average of them.

2) Explain the reason why performance measurements must be taken multiple times for each program and matrix size.

3) Save the results in a file called `time_slow_fast.dat` using the following format:

```
<N> <tiempo "slow"> <tiempo "fast">
```

4) Draw a chart (using GNUPlot) in PNG format to compare the results. Call the file `time_slow_fast.png` and include the chart in the final document.

5) The final document should contain, in addition to the chart, an explanation of results and the methodology used to obtain the results. Why is the execution time for small matrix similar, but different when the size is increased? How is the matrix store in memory, per rows (elements of the same row are stored consecutive in memory) or per column (elements of the same column are stored consecutive in memory)?

NOTE: To answer these questions is important to understand the source code of the programs.

Exercise 2: Cache size and Performance (3 p)

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. Valgrind includes the tools *cachegrind* or *callgrind* (an extension of the first one). Particularly, cachegrind simulates how your program interacts with a machine's cache hierarchy and (optionally) branch predictor. It simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). This exactly matches the configuration of many modern machines.

To use cachegrind or callgrind just execute the following:

```
>valgrind --tool=<tool name> [tool options] <programs> [program
args]
```

Both tools provide a set of parameters (options) to configure the two levels of cache:

```
--I1=<size>,<associativity>,<line size>
Specify the size, associativity and line size of the level 1
instruction cache.

--D1=<size>,<associativity>,<line size>
Specify the size, associativity and line size of the level 1
data cache.

--LL=<size>,<associativity>,<line size>
Specify the size, associativity and line size of the last-level
cache.
```

Note that size value is in bytes, and the *associativity* option is the number of ways of the cache.

The output of these tools, in addition to the program and tool messages, is file with the following name: *cachegrind.out.<pid>* or *callgrind.out.<pid>*. dependiendo de la herramienta utilizada. Se puede modificar el nombre del fichero que contenga el volcado de la salida mediante los argumentos siguientes:

```
--callgrind-out-file=<file>    Output file name [callgrind.out.%p]

--cachegrind-out-file=<file>   Output file name [cachegrind.out.%p]
```

These files include the simulation results, in particular, counters and statistics of cache use. A human-readable output of these files can be generated using the tool *cg_annotate* (for *cachegrind*) and *callgrind_annotate* (for *callgrind*).

For example, the information of the cache when executing the command 'ls' is obtained executing the following commands:

```
> valgrind --tool=cachegrind --cachegrind-out-file=ls_out.dat ls
      ...
> cg_annotate ls_out.dat | head -n 30

--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
```

```
LL cache:          8388608 B, 64 B, 16-way associative
Command:           ls
Data file:         ls_out.dat
Events recorded:   Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:      Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order:  D1mr
Thresholds:        0.1
Include dirs:
User annotated:
Auto-annotation:   off

--------------------------------------------------------------------------------
     Ir  I1mr  ILmr      Dr  D1mr  DLmr     Dw  D1mw  DLmw
--------------------------------------------------------------------------------
548,211 1,696 1,569 142,123 4,342 2,819 59,621 1,152 1,042  PROGRAM TOTALS
```

The output shows the default cache configuration (do not belong to a laboratory computer):

- A first level data and instructions caches of 32KB, block size of 64B and 8 ways.
- A second level cache of 8MB, block size of 64B and 16 ways.

The default configuration is equal to the system cache configuration where *valgrind* is executed. The output shows the number of fetched instructions, 548.211 (column *Ir*), with 1.696 misses in first level (column *I1r*) and 1.569 misses in second level (column *ILmr*). Same results are available for read and write operations for data cache (columns *Dr?* y *Dw?* respectively).

Tasks to do:

1) Obtain the number of data cache misses for read and write operations for both programs (*fast* y *slow*) using NxN matrixes, where N is a value from 1024+1024*P to 1024+1024*(P+1) in increments of 256. The first level cache sizes (data and instructions) to test are 1024, 2048, 4096 y 8192 Bytes, and the second level cache size is 8 Mbytes. All caches are direct mapped (1 way) and the block size is 64 bytes. Valgrind is an emulator, so it is only necessary to **run the program once** for each configuration.

2) Save the results in a file with the name `<tamCache>.dat` (one file per cache size) using the following format:

   `<N> <D1mr "slow"> <D1mw "slow"> <D1mr "fast"> <D1mw "fast">`

3) Draw two charts, one for read misses and the other for write misses, in PNG format and save them as `cache_lectura.png` and `cache_escritura.png`. In each case, the size of the cache will be indicated on the abscissa axis, while the number of failures will be indicated on the ordinate axis. You will need to paint a series of data for each cache size.

4) Explain the results. Are there any trend changes observed by varying the cache sizes for the slow program? And for the fast program? Does the trend change when you look at a specific cache size and compare the slow and fast program? What is the cause of each of the observed effects?

Exercise 3: Cache and Matrix multiplication (3 p)

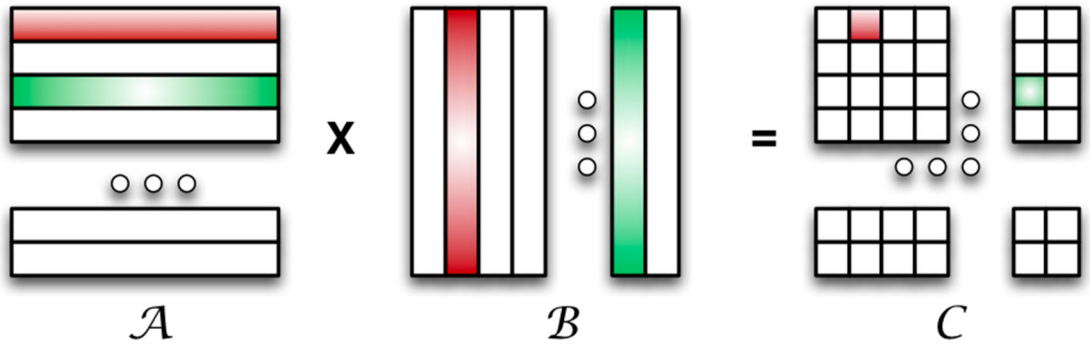
Implement a program to multiply two square matrixes of the same size. This program only receives a parameter, the size of the matrixes, N.

To simplify the implementation, use the functions in file `arqo3.c`: function `generateMatrix` creates one input matrix, and function `generateEmptyMatrix` creates the output matrix. The program must print the execution time after computing the multiplication. Do not consider the time to generate and destroy the matrixes.

IMPORTANT: Printing data to the screen has a high computing cost, so a program that measures computational performance in terms of computation time should never print data to the screen during time measurement.

Remainder: The method to multiply two matrixes, A and B, to obtain a matrix C is:

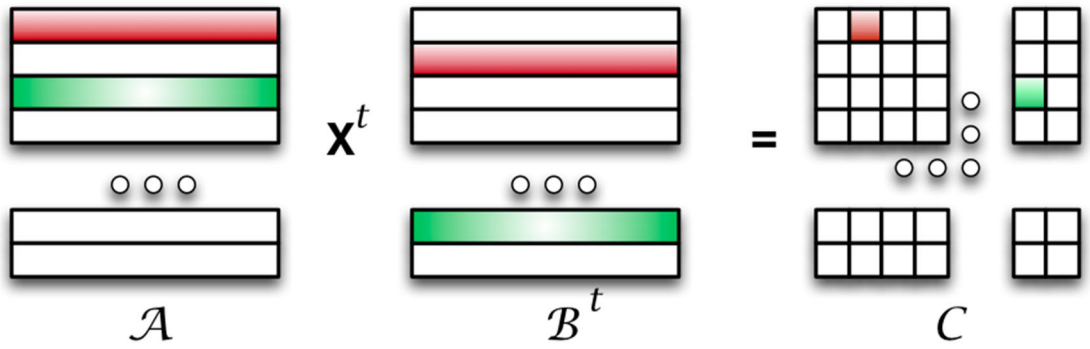
$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$$



- Note that matrix A is read row by row, while matrix B is read column by column.

Once the program that performs matrix multiplication has been developed, you have to create a new program from the previous one that performs "transposed multiplication". Note that in this case the result must be equivalent to the previous program, but the storage of matrix B changes. The elements of the transposed multiplication are calculated according to the following expression:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b^{t}_{jk}$$



As can be seen, the pattern of accesses to the data of the second matrix changes.

IMPORTANT: The version of the program that performs the "transposed multiplication" must consider the calculation time required to carry out the transposition of matrix B, but not the time to allocate and free the matrix, so the matrix must be allocated before starting the measurement and it must be free after the end of the measurement.

Once both programs are implemented, check that the result is the same, and then do the following:

1) Obtain the execution time for both implementations using NxN matrixes where N is a value from 256+256*P y 256+256*(P+1) in increments of 32. You will need to take results for all runs multiple times and interleaved (as indicated in exercise 1) and calculate the average of them.

2) Obtain the number of data cache misses for read and write operations for both programs using NxN matrixes where N is a value from 256+256*P y 256+256*(P+1) in increments of 32. Valgrind is an emulator, so it is only necessary to **run the program once** for each configuration.

3) Save the results in a file called `mult.dat` using the following format:

```
<N> <tiempo "normal"> <D1mr "normal"> <D1mw "normal">
<tiempo "trasp"> <D1mr "trasp"> <D1mw "trasp">
```

   Note: It is possible to execute sections 1) and 2) in the same script, which will allow unifying the output of both programs in a simpler way than if they are executed in separate scripts. But remember to execute valgrind once.

4) Draw two charts in PNG format, one for cache misses and the other for the execution time, to compare both programs. Save the charts as `mult_cache.png` and `mult_time.png`.

5) Explain the results. Are there any changes in the trend when varying the sizes of the matrices? What are the observed effects?

Exercise 4 (Optional): Caché parameters in matrix multiplication (2 p)

This exercise is not mandatory to pass, but it is required to obtain the maximum score for the practice. To consolidate the knowledge acquired in this practice, the students are asked to carry out a small study on how the different configurations of the caches affect the execution of the matrix multiplication and matrix transpose multiplication programs.

The student can "play" with the different configuration parameters of the caches studied throughout the practice (which may include, if they wish, the associativity and the cache line size) as well as the size of the multiplied matrices. The results that you report may include reading and writing errors, as well as execution times, and any other significant data that occurs to you.

All the experiments that you decide to carry out, as well as the results that you obtain, must be explained to justify the reason why you have decided to carry out the experiment and the conclusions that you have obtained from them. It will also be valued in this exercise to provide the Bash and GNUplot scripts or the Linux commands used to obtain the results.

MATERIAL TO SUBMIT

- Final document with all answers.
- In different folders, all files with results, and the scripts to generate the results and charts (if exists).
- Source code of programs for exercise 3 and 4.

Submit the zip file through Moodle before the day the next practice will be explain (until 11:59 pm).

EVALUATION

This practice total 10 points for the compulsory part plus (up to) 2 points for the optional exercise.

REFENCE MATERIAL

[1] **Valgrind.**
http://valgrind.org/

[2] **Cachegrind: a cache and branch-prediction profiler.**
http://valgrind.org/docs/manual/cg-manual.html

[3] **Callgrind: a call-graph generating cache and branch prediction profiler.**
http://valgrind.org/docs/manual/cl-manual.html

[4] **Bash Scripting tutorial.**
https://linuxconfig.org/bash-scripting-tutorial

[5] **Introduction to bash shell redirections.**
https://linuxconfig.org/introduction-to-bash-shell-redirections

[6] **GNUPLOT 4.2 - A Brief Manual and Tutorial.**
http://people.duke.edu/~hpgavin/gnuplot.html