Work assignment in AA

Matrix-matrix multiplication

Context

The main goal of this assignment is to develop in students transversal skills applied to a specific topic in the Curricular Unit *Advanced Architectures*: the methodology on the characterization of the performance bottlenecks on a computing platform and/or on the code profiling and its performance analysis on that platform.
The development of these skills will be achieved through training in literature search, reading & interpreting scientific papers, planning experimental work, synthesizing relevant information, writing a short essay on a given theme and a short (15 min) oral communication and discussion of the results (dates do be defined later).

You should start by reading the documentation on the performance model Roofline (already supplied, see the class summaries), the Amdahl law applied to heterogeneous multicore platforms and the respective Gustafson extension to this law.
The next step is to get acquainted with one of the most popular portable interfaces to hardware performance counters on current processor architectures (in the form of a library), the *Performance API* (PAPI), by reading the paper that addresses this approach.

Once you carefully read these papers related to performance analysis and measurement of programs and computer systems, you are required to perform some specific tasks and to prepare a short presentation interpreting the obtained results.
This work should be performed by a 2-student team (the same as in PCP), who will deliver a single report and a single oral presentation.

Task 1
Performance of different matrix multiplication algorithms & implementations

1.1 Identify all PAPI performance counters that are available for the system CPU in a 662 node (in the *mei* queue). From these, **select** (and justify) the most relevant ones to analyse an application execution time and to identify potential bottlenecks.

1.2 Write a program that calls a single-threaded C function that computes the dot product of 2 square matrices with size `NxN`, **C = `A * B`**, in single precision, and with no block optimization.

The main program should build a square matrix A with randomly generated values and a matrix B where all elements are "1" and call the dot product function. The function receives as arguments the pointers for the 3 matrices and their dimension `N`. The algorithm for this product contains 3 nested loops for the indexes `i`, `j` and `k`, with this order, where `i` and `j` represent the rows and columns of these matrices.

1.3 Develop different single threaded implementations of the dot-product function, with a triple nested loop, exploring two alternative combinations of the **index order**: **(1)** `i-k-j` and **(2)** `j-k-i`.

For each alternative implementation, the access to the elements of either A or B (or both) will be row by row, or column by column, which may impact performance; to eventually reduce this negative impact (when accesses are by column), **modify** each of your original versions that may have this negative impact, by transposing at the beginning the matrix(ces) that is(are) accessed by column, so that the memory accesses are performed row by row during the dot product computation.

1.4 To analyse the code execution in a single core of the **multicore devices** of one 662 node (in the *mei* queue), **select** (and justify) the sizes for 4 different data structure(s):

(i) That will completely fit in L1 cache,
(ii) That only requires accesses to L2 cache,
(iii) That only requires accesses to L3 cache and
(iv) That requires accesses to external RAM (memory requirements at least 4 times larger than L3 cache).

Validate your code by testing the product A*B (all resulting columns should have the same values) and the product B*A (all resulting rows should have the same values).

1.5 Build a table with your execution time measurements of your implementations of the dot-product function, following the *K*-best scheme, with *K*=3 with 5% tolerance and at most 8 execution times.

1.6 For the best execution time of each dot-product implementation, and using PAPI data from the hardware counters:

(i) Estimate the number of RAM accesses per instruction and the number of bytes transferred to/from the RAM, with and without transposed matrices;

(ii) Confirm those values with PAPI readings (note: some values may have to be inferred from other counters).

1.7 For each data set size:

(i) Estimate the number of FP op's executed in each dot-product implementation &
(ii) Plot the achieved performance of your implementations in the roofline graph.

1.8 For each data set size and both with and without transposed matrices, **build** a table where each line shows the miss rate (%) on memory reads in cache levels 1, 2 and 3 (only for the algorithm implementation that benefitted from transposed matrices).

1.9 Interpret the obtained results with your implementations, starting with bound characterization (CPU bound or memory bandwidth bound), performance bottlenecks and the impact of the matrix transpose approach to structure data in memory.

1.10 The **block optimization** technique is a key technique that may drastically improve performance of the matrix multiplication:

(i) Justify this statement and

(ii) Apply this technique to the single-threaded code that requires access to an external RAM.

1.11 This function contains the right ingredients for **vectorization**.

Compile your code with the adequate compiler switch and **confirm** it vectorized the code; if not, **modify** the code to force the compiler to vectorize.

Repeat 1.5 only for the two smaller data sets.

1.12 Modify your vectorized dot-product function to be efficiently executed in all cores of the **multicore devices** of one 662 node (without HT; use OpenMP).

Repeat 1.5 only for the larger data set (adapt the data structures to follow the guidelines in 1.4).

1.13 Modify your dot-product function to be executed in all SMX of a **GPU Kepler** of one 662 node (select only one implementation, **(1)** or **(2)**), and in all cores of the **Intel Knights Landing** many-core server.

Complement the table built in 1.12, including the data transfer times between the CPU-device and the accelerator and only for the larger data set (adapt the data structures to follow the guidelines in 1.4).

Task 2
Report writing and oral presentation

Write a short essay in English (no longer than 6 pages plus annexes with additional info).

Include in this essay:

- **(i)** Title
- **(ii)** List of authors
- **(iii)** Abstract
- **(iv)** Introduction
- **(v)** Relevant mid-sections, describing the multiplication algorithm behaviour (computations and memory accesses), the experimental setup and the relevant results, with associated discussion
- **(vi)** Conclusions
- **(vii)** References, by order of appearance in the essay, with all data pertinent to find the publication, e.g. author(s), title, place where it was published and who published, year.

Prepare a set of slides in English to give an **oral presentation & discussion** of your work, in no more than 10 min. Be prepared for a discussion with your colleagues.