Cross Correlation of all pairs of columns in a huge dataset

Lorenzo Catoni

Leonardo Giovannoni

Marco Lucio Mangiacapre

Problem

- Given a set of many metrics measured together (ex.: CPU usage, memory usage, …) on the same entity (ex.: a server), determine if they are correlated in any way.
- Results could be used to reduce the number of metrics measured in future samples.
- The given dataset MUST be entirely analysed, input reduction is not allowed in this context.

Input format

- Single CSV file
- One column per measured metrics
  - Header: name of the metric
  - Body: measured values
- Some columns represents metadata (i.e. timestamp et al.), they can be ignored while processing data (stripped in this analysis)
- Unknow number of rows and columns in input
  - Input may be too large to be hold in memory
- Metrics in the same row are extracted from the same system

Input CSV example for tests

```
"col1","col2","col3","col4","col5","col6","col7","col8"
"13562","10357","30754","26825","23508","30043","28938","10111"
"4808","13096","16702","21936","14933","6328","20732","10430"
"12335","25313","1066","15242","24547","15170","3110","13334"
"21891","10185","26028","24719","19499","23544","20108","13872"
"27943","31421","16439","8083","9641","20630","30652","17694"
"2940","20078","29753","26833","3845","4946","22617","24890"
"18711","32243","17611","13786","17079","25250","22190","25307"
"9993","32422","29676","29116","9166","24302","11250","32370"
```

Output format

- Single text file
- One row per couple of metrics in the input, three numbers per row
  - First two are the indexes of two data columns in the input
  - The third is the Pearson correlation coefficient of the two columns
- Example:
  - (0,1) -0.0207705
  - Means column 0 and 1 are not correlated

Pearson correlation coefficient

"is a measure of linear correlation between two sets of data"

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$

Problem: naïve implementation requires two passes

Pearson correlation coefficient (cont'd)

With some substitutions it's possible to obtain a formula that requires only one pass

$$\rho_{X,Y} = \frac{\mathbb{E}[XY] - \mathbb{E}[X]\,\mathbb{E}[Y]}{\sqrt{\mathbb{E}[X^2] - (\mathbb{E}[X])^2}\,\sqrt{\mathbb{E}[Y^2] - (\mathbb{E}[Y])^2}}.$$

Analysed solutions

- CPU based solution, variations:
  - Single thread vs. multi thread
  - Data processed by rows vs. data processed by columns
  - Float vs. double
- GPU accelerated solution
  - Single thread
  - Read a chunk and process it on the GPU, maintain partial results in GPU memory
  - Float vs. double

Processing by rows vs. by columns

- To calculate the PCC every value in a data series must be combined only with all the corresponding values in the other DSs.
- So, given many DSs, we can evaluate and discard every row one by one, or evaluate entirely a pair of columns (in a single chunk) before considering the next pair.
- GPU based solutions process data by columns only.

- Processing by rows has always been significantly slower than processing by columns, then results have not been reported here.

The algorithm

- Read the input csv N lines at the time
- Calculate a partial result of this N lines for all the couples of columns
- Sum new partial result with the previous store
- Iterate until the whole file is read
- In one final computation convert the partial result into the correlation coefficient
- Save or print the results

CPU multi-thread solution

- To implement multi-thread CPU-based solutions we followed the producer-consumer(s) pattern:
  - The main thread parse the input and put the parsed chunk in a queue.
  - Workers concurrently access the queue to extract a chunk and process it.
  - When all the input has been processed the producer set a flag to make idle workers terminate and wait for them to put their partial results in a queue.
  - Eventually, the main thread collect, merge and display the partial results.

Testing environment

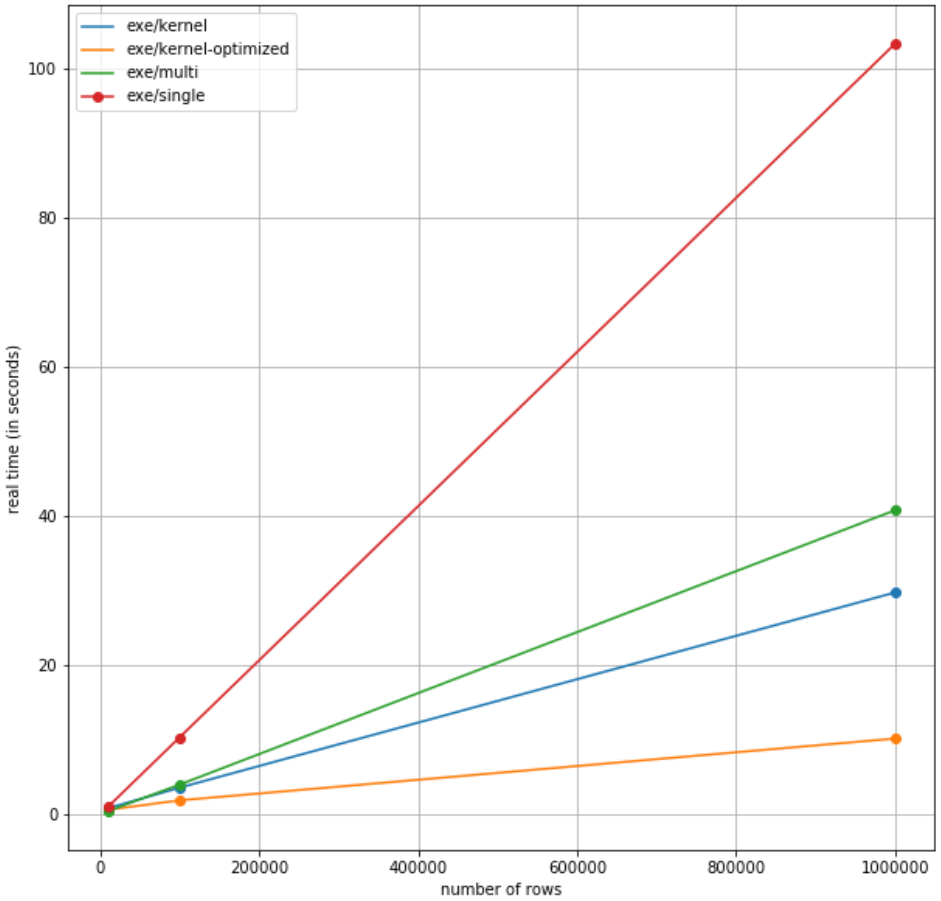
- All benchmarks and measures have been performed on a remote otherwise idle server with the following specs:
  - GPU: GeForce GTX 1080 Ti
  - CPU: Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz, 8 core
  - RAM: 32GB
  - HDD: ST2000DM001-1ER1 (2TB)
- Performance measurement tools:
  - time: to gather general measures about CPU time
  - nvprof: to get GPU-only results
  - perf: to collect CPU-only execution statistics and find bottleneck

I/O

- In the original problem, data are stored on an HDD whose read speed was about 0.2GB/s.
- To speed up tests and ignore disk access time, benchmarks have been performed relying on the content of the input files being present in the kernel caches in RAM. Thus I time was negligible: about 10GB/s.
- Input parsing must be performed sequentially.
- Read time has been measured using programs like wc and cat, both with disk caches enabled or disabled.
- Output time depends only on the number of columns, then it is negligible for large datasets.

Results

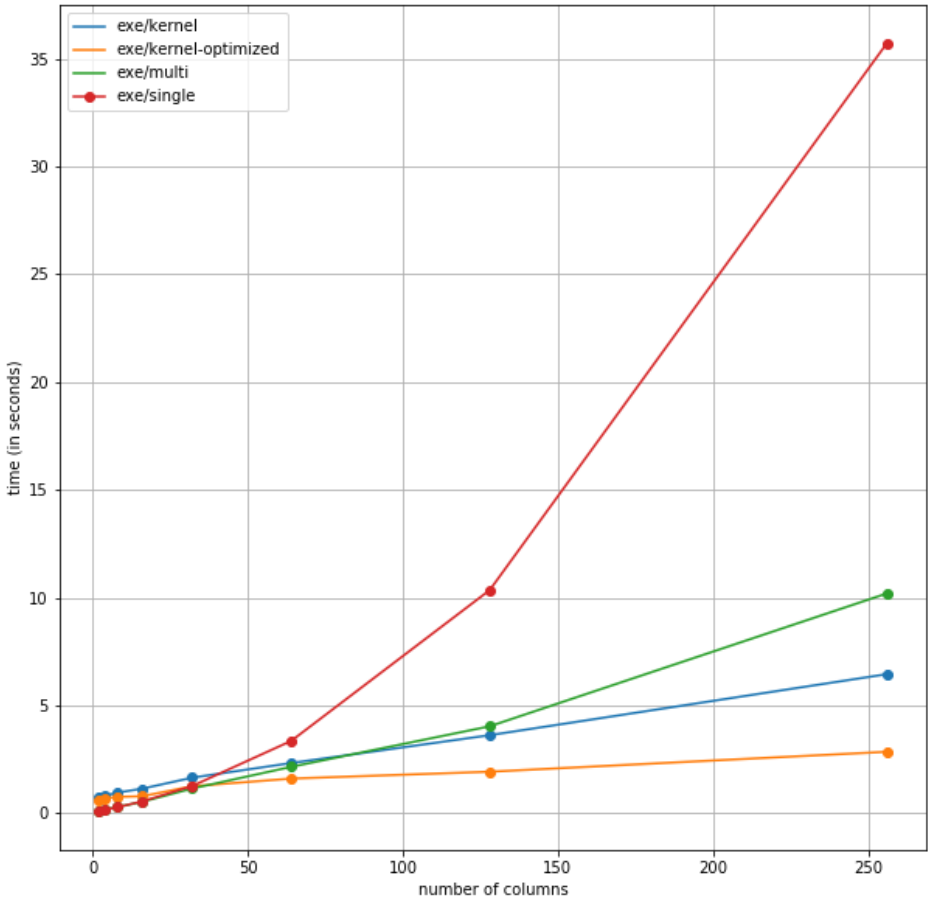
Execution time over number of rows with different configurations (number of columns fixed at 128)



Configuration	Speedup
exe/single	1.0
exe/multi	2.5315
exe/kernel	3.4680
exe/kernel-optimized	10.1147

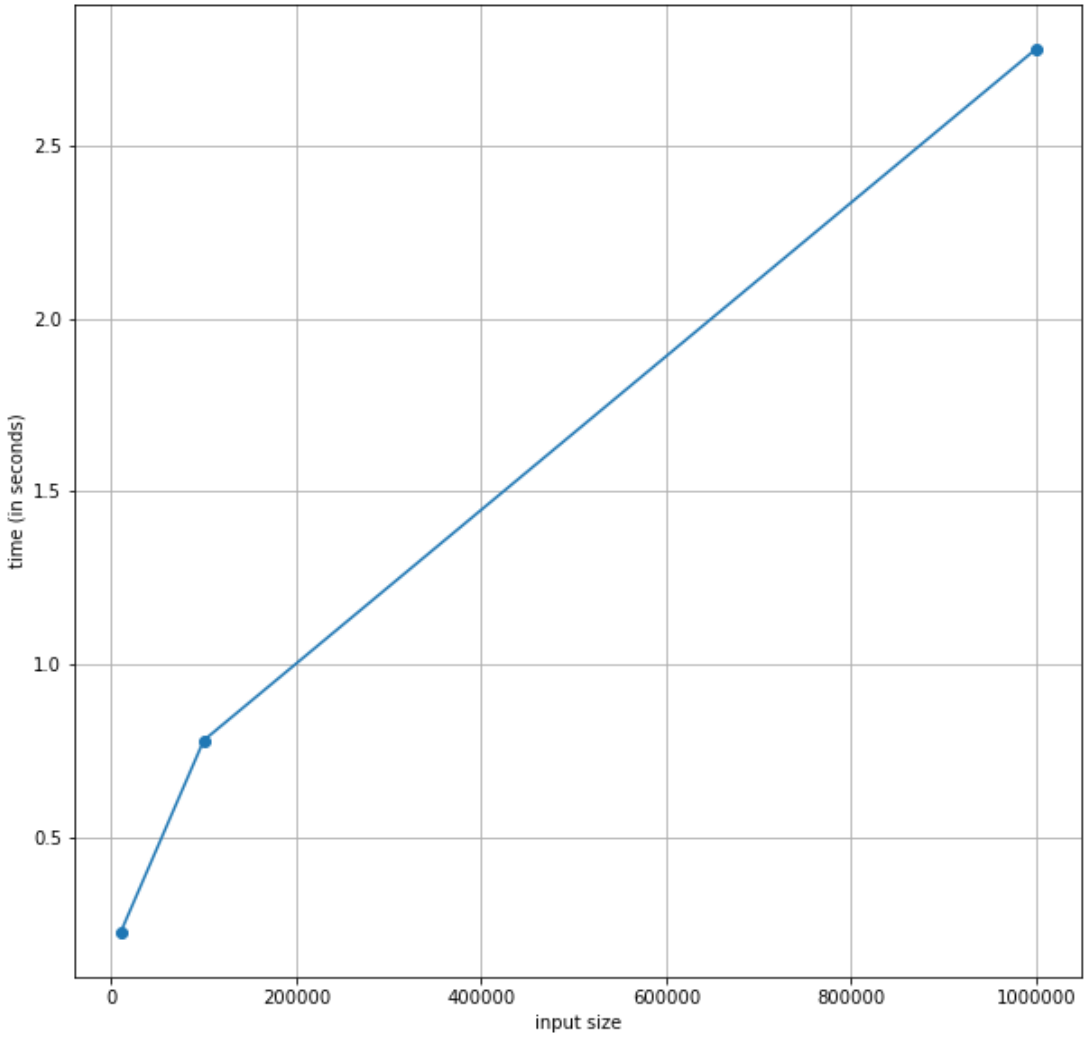
Results (cont'd)

Execution time over number of columns with different configurations (number of rows fixed at 100k)



Configuration	Speedup
exe/single	1.0
exe/multi	3.500
exe/kernel	5.530
exe/kernel-optimized	12.50

CUDA only results



- nvproofGPU only execution time over a varying input size, gathered with “nvprof”.
- CUDA kernel start-up require a varying amount of time, causing the initial bent in the graph.
- The same GPU-time is achieved both with or without compiler optimizations, that affect only the sequential part of the task.

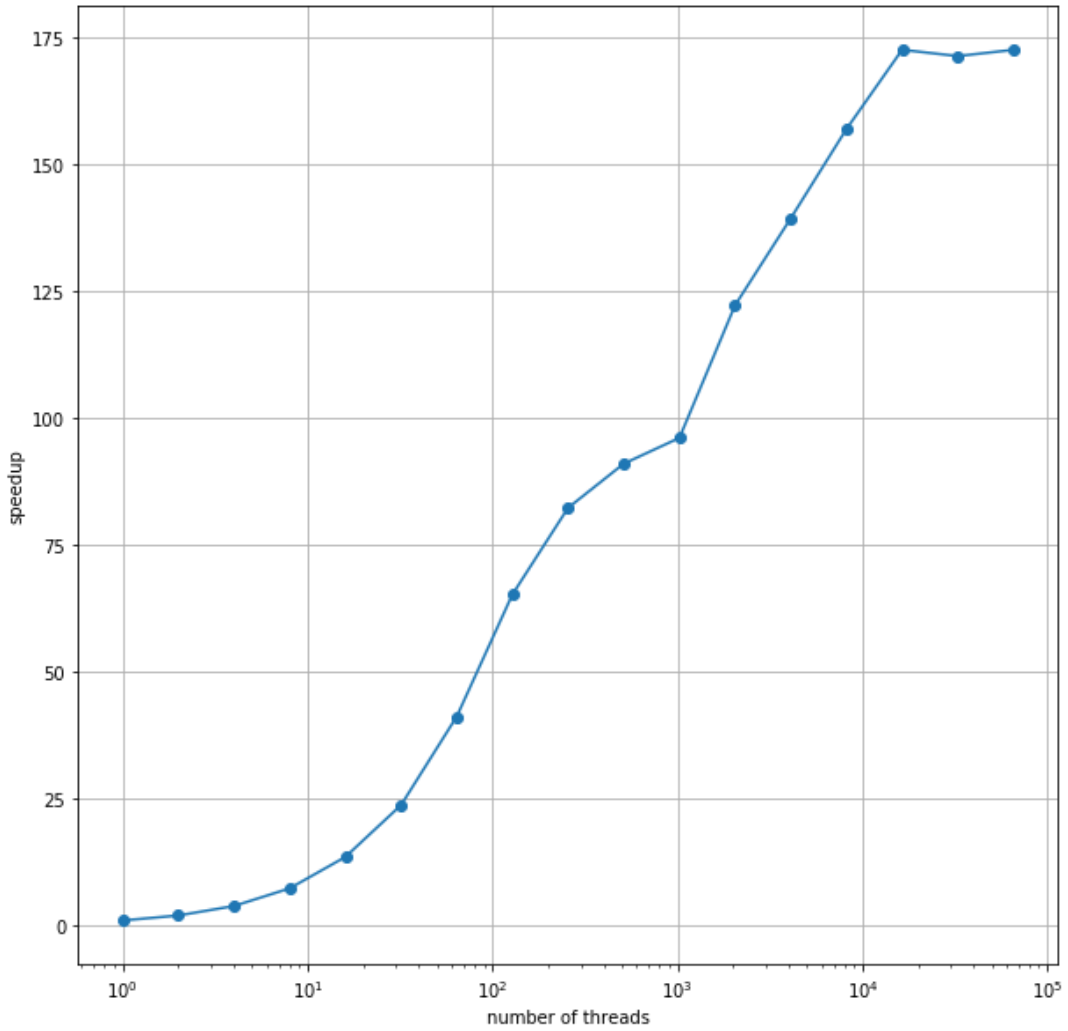
Execution bottleneck

- Benchmarks and performance measurement tools showed the main bottleneck in execution performances was due to CSV parsing:
  - The parsing without disk access (csv reading+conversion to floating point) speed was about 0.11GB/s.
  - This results has been collected running a modified version of the executables not performing the computations on the dataset.
- “perf” showed the most time consuming task was specifically parsing the csv lines and parsing strings to obtain floating points:
  - About 22% of execution time without considering parsing time was spent executing std::strf() or std::strd() to get number from string. It is more than the time spent on GPU (about 14% of the time with 256 columns).

Applied optimizations

- Due to the extremely simple nature of the task (i.e.: text parsing and performing nothing more than a huge amount of floating point sums and products) the only optimizations we could rely on were compiler optimizations:
  - Loop unrolling and inlining
  - Instructions reordering
  - Floating point (aggressive) optimizations
  - And much many listed on GCC manual
- However, none of them made the on CPU computation time comparable to the GPU results.

Speedup over the number of threads (GPU)



GPU thread count	speedup
1	1.0
2	1.977180
4	3.849154
8	7.328337
16	13.518071
32	23.729698
64	41.179231
128	65.242591
256	82.319421
512	91.024612
1024	96.112646
2048	122.257789
4096	139.391881
8192	156.984066
16384	172.601079
32768	171.362547
65536	172.594878

Conclusions

- The given task is a naturally parallelizable and the limiting factor is the sequential part of the execution.
- GPU-based solutions outperform CPU-based solution with large datasets (many rows) of many (>=64) columns with highly optimized code (i.e. CPU relying on vectorised instructions).
- Without relying on CPU compiler optimizations, GPU-based solutions outperform CPU-only solution with large datasets of at least 32 columns.

Conclusions (cont'd)

- To process a dataset of 256 columns and 1’000’000 rows GPU requires only about 2.7s, while disk read time requires about 10s.
- Looking for further optimizations is worthless to reduce execution time because the IO+parsing bottleneck would make them imperceptible.
- It is not surprising that we had no way to improve GPU execution time since the whole task require no more than simple products and sums of floating point values, a natural task for a GPU processor.