Computer Architecture Project Report

Lorenzo Catoni Leonardo Giovannoni Marco Lucio Mangiacapre

Contents

1 Introduction

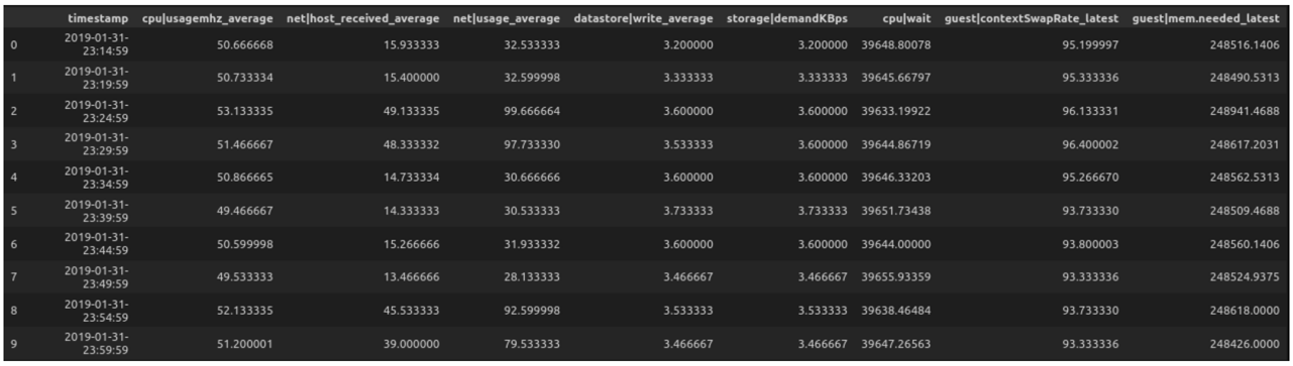
The code for the whole project can be browsed here `https://github.com/PuffoTrillionarioGonePublic/CA-Project`. Here, we will reports the description of our project with the main results obtained during our study.

2 Background

To manage a data centre is not an easy task: thousands of machines must be constantly monitored to obtain performance metrics in order to find performance issue that correspond to energy and money waste. Collecting all possible metrics require a huge amount of memory to store results and more data does not correspond necessarily to more information: many measures could be correlated and own the same amount of information. By detecting these correlations between observed quantities, we could reduce performance overhead and save a lot of space!

2.1 The 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 can than be used to reduce the number of metrics measured in future samples.
In input we get a single csv file with an unknown number of rows and columns, the input file may be too large to fit in memory (see Figure 1 for an example).



	timestamp	cpu|usagemhz_average	net|host_received_average	net|usage_average	datastore|write_average	storage|demandKBps	cpu|wait	guest|contextSwapRate_latest	guest|mem.needed_latest
0	2019-01-31-23:14:59	50.666668	15.933333	32.533333	3.200000	3.200000	39648.80078	95.199997	248516.1406
1	2019-01-31-23:19:59	50.733334	15.400000	32.599998	3.333333	3.333333	39645.66797	95.333336	248490.5313
2	2019-01-31-23:24:59	53.133335	49.133335	99.666664	3.600000	3.600000	39633.19922	96.133331	248941.4688
3	2019-01-31-23:29:59	51.466667	48.333332	97.733330	3.533333	3.600000	39644.86719	96.400002	248617.2031
4	2019-01-31-23:34:59	50.866665	14.733334	30.666666	3.600000	3.600000	39646.33203	95.266670	248562.5313
5	2019-01-31-23:39:59	49.466667	14.333333	30.533333	3.733333	3.733333	39651.73438	93.733330	248509.4688
6	2019-01-31-23:44:59	50.599998	15.266666	31.933332	3.600000	3.600000	39644.00000	93.800003	248560.1406
7	2019-01-31-23:49:59	49.533333	13.466666	28.133333	3.466667	3.466667	39655.93359	93.333336	248524.9375
8	2019-01-31-23:54:59	52.133335	45.533333	92.599998	3.533333	3.533333	39638.46484	93.733330	248618.0000
9	2019-01-31-23:59:59	51.200001	39.000000	79.533333	3.466667	3.466667	39647.26563	93.333336	248426.0000

Figure 1: Example of input file

As output we will produce a single text file with all coutple of columns indexes and the Pearson correlation coefficient [1] of the two columns.
Recall that the Pearson correlation coefficient is a measure of linear correlation between two series of data, so for example in output we may produce the line: `(0, 1):  0.0013`, meaning that column 0 and 1 are not correlated (see Listing 1 for an example).

```
(0,1) 0.253106
(0,2) -0.170867
(0,3) -0.630296
(0,4) 0.225982
(0,5) 0.626356
(0,6) 0.342535
(0,7) -0.114682
(1,2) -0.294529
(1,3) -0.504529
(1,4) -0.375457
(1,5) 0.12661
(1,6) -0.228954
(1,7) 0.719545
(2,3) 0.728292
(2,4) -0.414364
```

```
(2,5) 0.239953
(2,6) 0.488402
(2,7) 0.32773
(3,4) -0.120147
(3,5) -0.0289303
(3,6) -0.123404
(3,7) 0.156105
(4,5) 0.447088
(4,6) -0.250147
(4,7) -0.65898
(5,6) 0.177765
(5,7) 0.0640271
(6,7) -0.180228
```

Listing 1: Example of an ouput file

2.2 The Hardware Setup

All benchmarks and measures have been performed on a remote otherwise idle server with the following specs:

- **GPU**: GeForce GTX 1080 Ti [2]
- **CPU**: Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz, 8 core [3]
- **RAM**: 32GB
- **HDD**: ST2000DM001-1ER1 (2TB)

2.2.1 GPU proprerties

The specs of the gpu card used in this study can be seen in Listing 2.

```
\$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce GTX 1080 Ti"
  CUDA Driver Version / Runtime Version          11.6 / 10.1
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 11178 MBytes (11721179136 bytes)
  (028) Multiprocessors, (128) CUDA Cores/MP:    3584 CUDA Cores
  GPU Max Clock rate:                            1683 MHz (1.68 GHz)
  Memory Clock rate:                             5505 Mhz
  Memory Bus Width:                              352-bit
  L2 Cache Size:                                 2883584 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384,
     16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)
    >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.6, CUDA Runtime Version = 10.1,
    NumDevs = 1
Result = PASS
```

Listing 2: Properties of the grapic card used in this study

2.3 The Tools we used

The tools we used for benchmarks and performance analysis are the following:

- **time** [4]: a Linux utility command to get statistics about execution time of a process
- **nvprof** [5]: is part of the CUDA package, this tool allows you to collect and view profiling data of CUDA-related activities on both CPU and GPU, including kernel execution, memory transfers, etc.
- **perf** [6]: a Linux tool that is capable of statistical profiling of the entire system (both kernel and userland code)

That's how we used `perf` to profile respectively CPU and GPU executables:

- `$ sudo perf record ./exe inputs/dataset-256-1000000.csv >> /dev/null`
- `$ sudo perf record ./exe inputs/dataset-256-1000000.csv >> /dev/null`

And to visualize the results: `$ sudo perf report`.
Note: since perf can't measure directly execution on GPU, the time spent on GPU was obtained thanks to the NVIDIA profiler nvprof.

3 The Algorithm

3.1 One Pass Pearson Correlation Coefficient

By definition the Person correlation is just a normalized measurement of the covariance, the formula for the correlation coefficient is:

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

where:

- cov is the covariance
- ρ_X is the standard deviation of X
- ρ_Y is the standard deviation of Y

But there is one problem: following the classical definition requires 2 passes of the input data. With some substitutions it's possible to obtain a formula that requires only one pass, the formula is the following:

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

3.2 Implementation

The used algorithm can be summarized in the following steps:

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

4 Source Code

All tested code is publicly available on GitHub at the following address: `https://github.com/PuffoTrilli onarioGonePublic/CA-project-enhanced`. In order to limit performance differences introduced by different code styles and improve source code maintainability notwithstanding the necessity of testing different solutions (e.g. GPU vs. GPU), CPP macros have been used to change the compilation results depending on compiler supplied option, but maintain a single copy of the source code.

5 CPU Results

We will very briefly discuss the CPU solution first.

5.1 Performance Bottlenecks

Using the performance measures analysis we detected fourd different performance bottleneck, that we can group in two categories:

- Intrinsic in the development context:
  - Disk access time: depending on hardware
  - CSV parsing: due to text format structure and corner case to handle, only slightly optimized by improving the csv parser
  - String to number parsing: due to the complexity and variability of floating-point representations
- Depending on our implemented solution:
  - Computation bottleneck: more than 50% execution time spent on chunks processing

5.2 Performance Improvements

To improve the performance we used the following strategies:

- String to double/float casting can be improved by using multiple CPU thread
- Producer-consumer pattern:
  - Main thread extract .csv rows and put them in a lock-free queue
  - A fixed number of workers cyclically poll for new rows to convert strings to numbers and fill chunks to analyse, filled chunks are then stored in a separate queue
  - After filling a few chunks (or when row polling fails), workers start analysing a few of them on CPU or GPU, depending on program configuration.
  - After having parsed all the input, the main thread start acting like a worker.

5.2.1 Code optimizations

In order to improve code performances we applied many code optimization, some of them were studied by us and others were introduced by the compiler. The applied optimizations are the following:

- Introduced by us:
  - Reduction of memory references by using local variables as accumulators
  - Avoid recomputing expression by holding results in local variables
- Introduced by the compiler:
  - Loop unrolling
  - Function inlining
  - Instruction reordering
  - Introduction of vectorised instruction where possible
  - Memory access reduction by using register
  - All available legal (i.e. respecting IEEE floating point standards) optimization

6 GPU Results

6.1 Data collection methodology

All displayed results have been obtained by performing multiple executions of all the pairs (executable, input dataset) to get a statistically meaningful mean of the execution times, together with the corresponding confidence interval. Tens of configurations have been tested but only a bunch of them are here displayed for brevity. Execution times where gathered using the tools presented in the previous sections and analysed in order to find where the performance was lost.

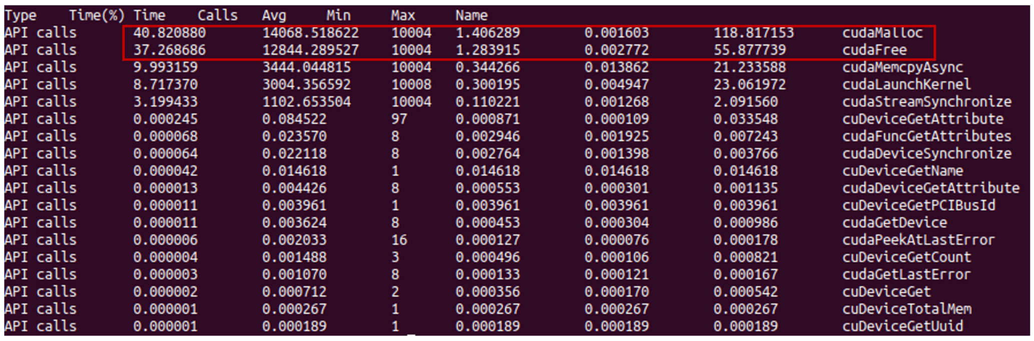
6.2 Fist implementation

The first GPU solution was structured in the following way:

- Input data read and parsed in CPU
- Chunks are processed asynchronously as they are filled
- Chunks are organised as a vector of vector
- GPU is used to process as much data as possible in parallel

6.2.1 Problem

From the output of nvprof (Figure 2 we can clearly see that the majority of the API calls is spend during memory management, namely `cudaMalloc` and `cudaFree`.



```
Type     Time(%) Time     Calls    Avg      Min      Max      Name
API calls        40.820880         14068.518622      10004    1.406289          0.001603          118.817153        cudaMalloc
API calls        37.268686         12844.289527      10004    1.283915          0.002772          55.877739         cudaFree
API calls        9.993159          3444.044815       10004    0.344266          0.013862          21.233588         cudaMemcpyAsync
API calls        8.717370          3004.356592       10008    0.300195          0.004947          23.061972         cudaLaunchKernel
API calls        3.199433          1102.653504       10004    0.110221          0.001268          2.091560          cudaStreamSynchronize
API calls        0.000245          0.084522          97       0.000871          0.000109          0.033548          cuDeviceGetAttribute
API calls        0.000068          0.023570          8        0.002946          0.001925          0.007243          cudaFuncGetAttributes
API calls        0.000064          0.022118          8        0.002764          0.001398          0.003766          cudaDeviceSynchronize
API calls        0.000042          0.014618          1        0.014618          0.014618          0.014618          cuDeviceGetName
API calls        0.000013          0.004426          8        0.000553          0.000301          0.001135          cudaDeviceGetAttribute
API calls        0.000011          0.003961          1        0.003961          0.003961          0.003961          cuDeviceGetPCIBusId
API calls        0.000011          0.003624          8        0.000453          0.000304          0.000986          cudaGetDevice
API calls        0.000006          0.002033          16       0.000127          0.000076          0.000178          cudaPeekAtLastError
API calls        0.000004          0.001488          3        0.000496          0.000106          0.000821          cuDeviceGetCount
API calls        0.000003          0.001070          8        0.000133          0.000121          0.000167          cudaGetLastError
API calls        0.000002          0.000712          2        0.000356          0.000170          0.000542          cuDeviceGet
API calls        0.000001          0.000267          1        0.000267          0.000267          0.000267          cuDeviceTotalMem
API calls        0.000001          0.000189          1        0.000189          0.000189          0.000189          cuDeviceGetUuid
```

Figure 2: Output of nvprof on the first implementation

6.2.2 Solution

We can reduce the number of calls to `cudaMalloc` and `cudaFree` in two ways:

- By storing a whole chunk in a single continuous buffer we can reduce memory management operations and speed up copy
- Instead of reallocating the device buffer on every iteration, we can allocate once at the start and always keep using the same one

As can be seen in Figure 3 doing so improves the time spend in API calls, yielding a speedup of 1.43. The total number of malloc and free goes down from 10004 to just 8!

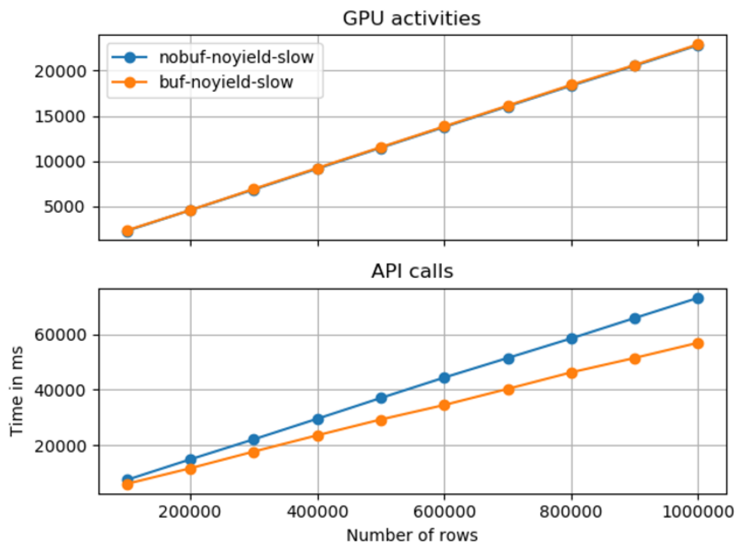


Figure 3: Plot of execution time of the first version vs the buffered one

6.3 Buffered implementation

6.3.1 Problem

We noticed that the main kernel function could be optimized: we were recalculating for each column the sum and the sum of squares for every column pair thus introducing a lot of unnecessary computations.

6.3.2 Solution

We can calculate the sum and sum of squares only once for each column, and then for every column just calculate the mean of the product. Doing this introduces a great performance improvement, as shown in Figure 4, yielding a speedup of 11.73 times.

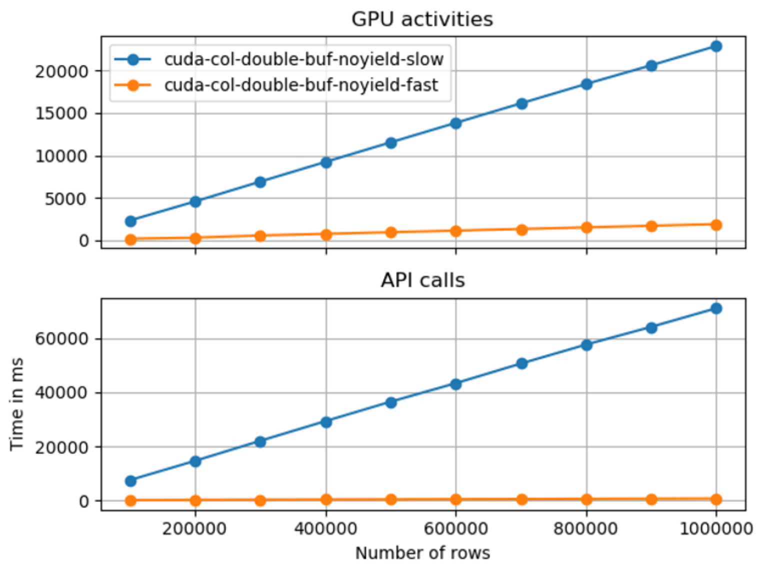
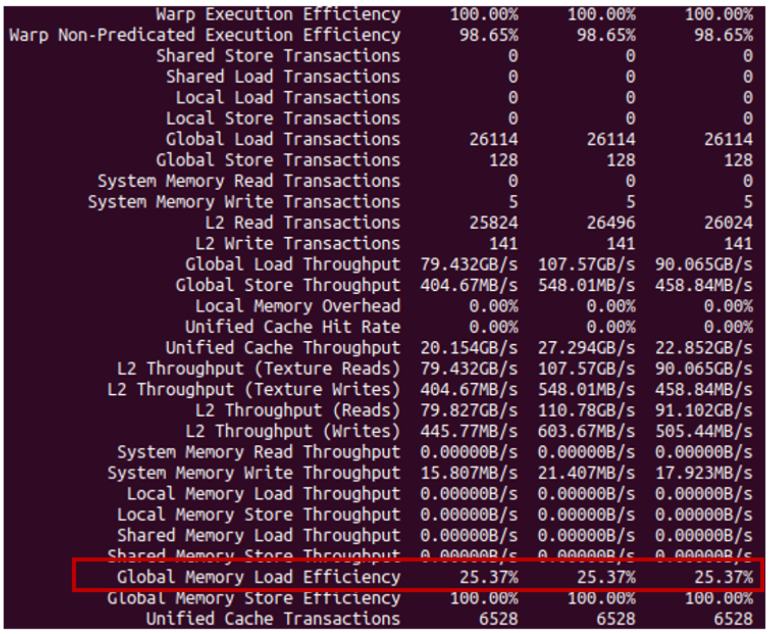


Figure 4: Plot of execution time of the buffered version vs the fast algorithm one

6.4 Fast algorithm version

6.4.1 Problem

Nvprof shows that this version has a very poor memory load efficiency (Figure 5), meaning that some time is lost due to a non optimal access patters to the memory.



```
             Warp Execution Efficiency    100.00%    100.00%    100.00%
Warp Non-Predicated Execution Efficiency     98.65%     98.65%     98.65%
               Shared Store Transactions          0          0          0
                Shared Load Transactions          0          0          0
                 Local Load Transactions          0          0          0
                Local Store Transactions          0          0          0
                Global Load Transactions      26114      26114      26114
               Global Store Transactions        128        128        128
          System Memory Read Transactions          0          0          0
         System Memory Write Transactions          5          5          5
                   L2 Read Transactions      25824      26496      26024
                  L2 Write Transactions        141        141        141
                  Global Load Throughput  79.432GB/s  107.57GB/s  90.065GB/s
                 Global Store Throughput  404.67MB/s  548.01MB/s  458.84MB/s
                   Local Memory Overhead      0.00%      0.00%      0.00%
                  Unified Cache Hit Rate      0.00%      0.00%      0.00%
                Unified Cache Throughput  20.154GB/s  27.294GB/s  22.852GB/s
            L2 Throughput (Texture Reads)  79.432GB/s  107.57GB/s  90.065GB/s
           L2 Throughput (Texture Writes)  404.67MB/s  548.01MB/s  458.84MB/s
                   L2 Throughput (Reads)  79.827GB/s  110.78GB/s  91.102GB/s
                  L2 Throughput (Writes)  445.77MB/s  603.67MB/s  505.44MB/s
           System Memory Read Throughput  0.00000B/s  0.00000B/s  0.00000B/s
          System Memory Write Throughput  15.807MB/s  21.407MB/s  17.923MB/s
            Local Memory Load Throughput  0.00000B/s  0.00000B/s  0.00000B/s
           Local Memory Store Throughput  0.00000B/s  0.00000B/s  0.00000B/s
           Shared Memory Load Throughput  0.00000B/s  0.00000B/s  0.00000B/s
          Shared Memory Store Throughput  0.00000B/s  0.00000B/s  0.00000B/s
            Global Memory Load Efficiency     25.37%     25.37%     25.37%
           Global Memory Store Efficiency    100.00%    100.00%    100.00%
               Unified Cache Transactions       6528       6528       6528
```

Figure 5: Nvprof showing that the fast algorithm version has a poor memory load efficiency

6.4.2 Solution

The problem relies in the fact that the matrix is stored by column, storing the matrix by row gives a better memory access patters and thus a better memory overall. This optimization yields a speedup of 2.82 times over the previous version. Figure 6 shows the layout of a matrix by columns and by rows, Figure 7 shows the time difference between the two version and Figure 8 shows the improvement on memory load efficiency.

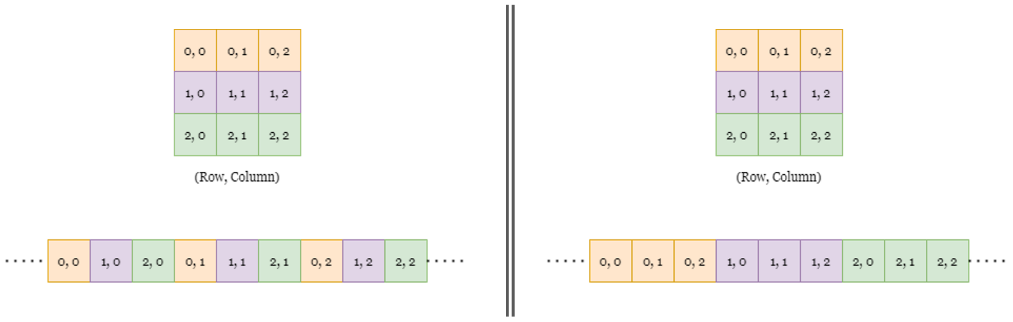


Figure 6: Layout of a matrix by columns and by row

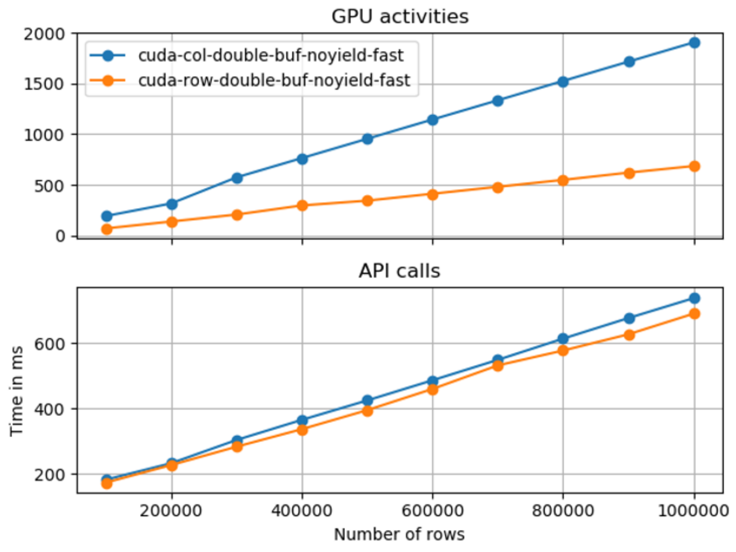
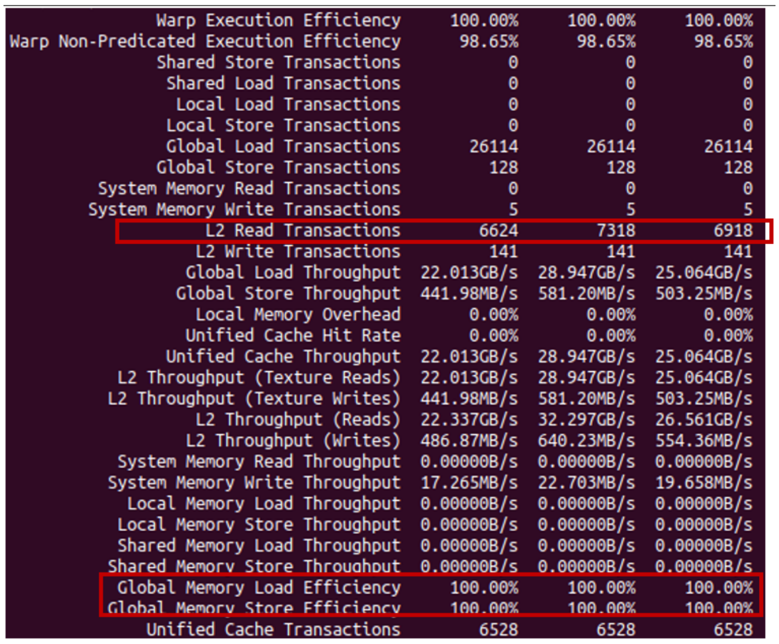


Figure 7: Time difference between the column version and the row version



```
                     Warp Execution Efficiency     100.00%     100.00%     100.00%
     Warp Non-Predicated Execution Efficiency      98.65%      98.65%      98.65%
                      Shared Store Transactions           0           0           0
                       Shared Load Transactions           0           0           0
                        Local Load Transactions           0           0           0
                       Local Store Transactions           0           0           0
                       Global Load Transactions       26114       26114       26114
                      Global Store Transactions         128         128         128
                  System Memory Read Transactions           0           0           0
                 System Memory Write Transactions           5           5           5
                          L2 Read Transactions        6624        7318        6918
                         L2 Write Transactions         141         141         141
                        Global Load Throughput  22.013GB/s  28.947GB/s  25.064GB/s
                       Global Store Throughput  441.98MB/s  581.20MB/s  503.25MB/s
                         Local Memory Overhead       0.00%       0.00%       0.00%
                        Unified Cache Hit Rate       0.00%       0.00%       0.00%
                      Unified Cache Throughput  22.013GB/s  28.947GB/s  25.064GB/s
                 L2 Throughput (Texture Reads)  22.013GB/s  28.947GB/s  25.064GB/s
                L2 Throughput (Texture Writes)  441.98MB/s  581.20MB/s  503.25MB/s
                          L2 Throughput (Reads)  22.337GB/s  32.297GB/s  26.561GB/s
                         L2 Throughput (Writes)  486.87MB/s  640.23MB/s  554.36MB/s
                  System Memory Read Throughput  0.00000B/s  0.00000B/s  0.00000B/s
                 System Memory Write Throughput  17.265MB/s  22.703MB/s  19.658MB/s
                   Local Memory Load Throughput  0.00000B/s  0.00000B/s  0.00000B/s
                  Local Memory Store Throughput  0.00000B/s  0.00000B/s  0.00000B/s
                  Shared Memory Load Throughput  0.00000B/s  0.00000B/s  0.00000B/s
                 Shared Memory Store Throughput  0.00000B/s  0.00000B/s  0.00000B/s
                 Global Memory Load Efficiency     100.00%     100.00%     100.00%
                Global Memory Store Efficiency     100.00%     100.00%     100.00%
                     Unified Cache Transactions        6528        6528        6528
```

Figure 8: Nvprof shows the improved memory load efficiency

6.5 GPU: performance dependency on CUDA block size and number

CUDA allow us to specify different sizes for CUDA blocks and also to limit the number of CUDA blocks used by kernels. In order to measure the performances impact of different configurations, we decided to test our best application varying CUDA kernel launch parameters. For each displayed point in the following images, 15 repetitions of the processing of the dataset made by 256 columns and 100K rows have been considered. Confidence intervals have been set to 95%.

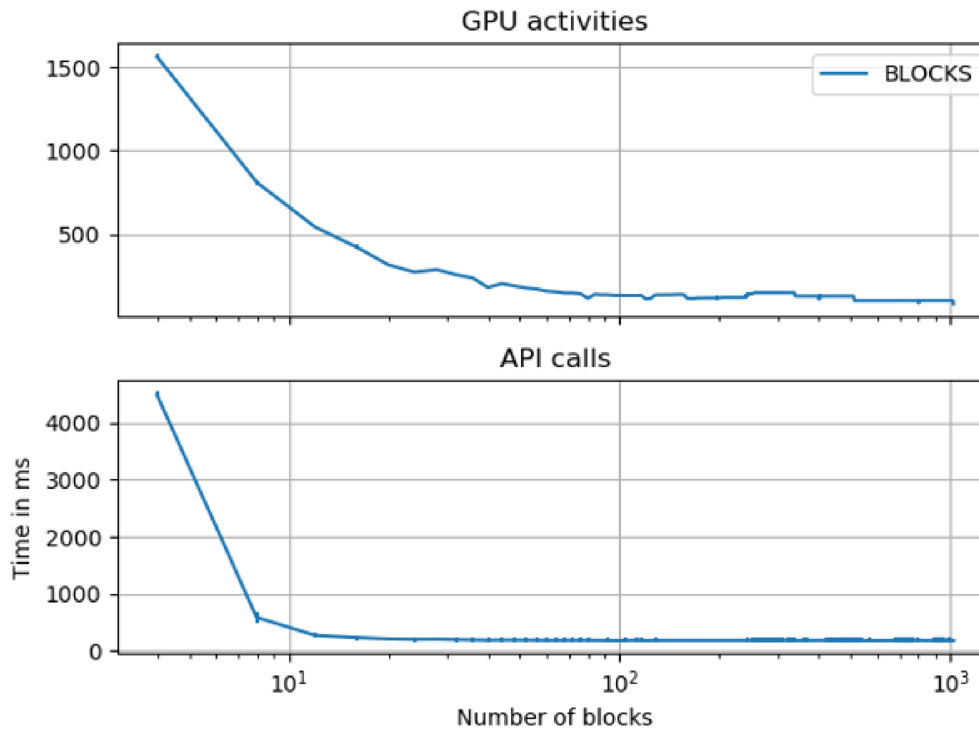


Figure 9: Execution time varying used blocks from 4 to 1020 with step of 4. Semilog x graph. Performances stabilize around 100 blocks. Block size is set equals to warp size (32).

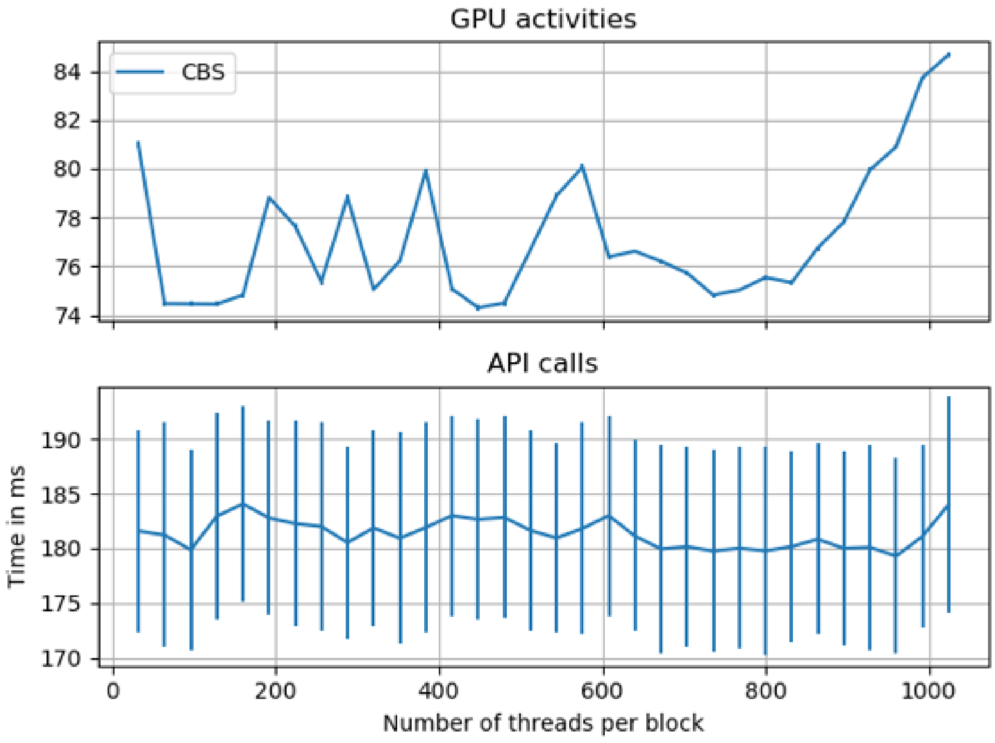


Figure 10: Execution time varying blocks size from 32 to 1024 (max block size) with step of 32. Performances reach the maximum around 128 threads per block (CUDA Cores/MP). No limitation is imposed on usable block number.

7 Conclusions

As a consequence of our study we can say that:

- The given task is a naturally parallelizable and the limiting factor is the sequential part of the execution, i.e. CSV parsing.
- String to number parsing can be divided among multiple CPU core, then improving parsing throughput.
- Managing GPU memory is much more expensive than managing ordinary RAM, however by storing chunks in a single buffer and by reusing pre-allocated buffers memory managing penalties are negligible.
- GPU performance are extremely sensitive to memory layout, by optimizing data structure organization cache usage is greatly improved and execution time is more than halved.
- GPU-based solutions outperform any CPU-based solution thanks to the extremely high number of operations GPU can execute concurrently.
- To process a dataset of 256 columns and 1'000'000 rows GPU requires only about 1.4s, while disk read time requires about 10s.
- The best obtained solution for GPU (thanks to an accurate algorithm design and aggressive compiler optimizations offered a x465 speedup!

References

[1] `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`.

[2] `https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877`.

[3] `https://www.intel.com/content/www/us/en/products/sku/80807/intel-core-i74790k-processor-8m-cache-up-to-4-40-ghz/specifications.html`.

[4] https://man7.org/linux/man-pages/man1/time.1.html.

[5] https://docs.alliancecan.ca/wiki/Nvprof.

[6] https://perf.wiki.kernel.org/index.php/Main_Page.