Cross Correlation of all pairs of columns in a huge dataset

Catoni Lorenzo

Mangiacapre Marco Lucio

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 issues 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 holding the same amount of information.
- By detecting these correlations between observed quantities, we could reduce performance overhead and save a lot of space!

Task definition

Analyse a huge set of measured computer systems performance metrics to calculate their correlation.

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.), so they are 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 (100s cols per Ms rows)
- Metrics in the same row are extracted from the same machine

Input format 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

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
  - It means column 0 and 1 are not correlated

Pearson correlation coefficient

“It is a measure of linear correlation between two sets of data“

$$p_{X,Y} = \frac{Cov(X,Y)}{\sigma_X * \sigma_Y}$$

Problem: the naïve implementation requires scanning the dataset three times

Statistical definitions

- Expectation, i.e. mean of a random variable:
  - $E[X] = \frac{1}{N}\sum_{i=1}^{N} X_i$
- Standard deviation:
  - $\sigma_X = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - E[X])^2}$
- Covariance:
  - $Cov(X,Y) = \frac{1}{N}\sum_{i=1}^{N}(X_i - E[X])(Y_i - E[Y])$

Pearson correlation coefficient (cont'd)

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

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

Implementation and performance analysis

Testing environment

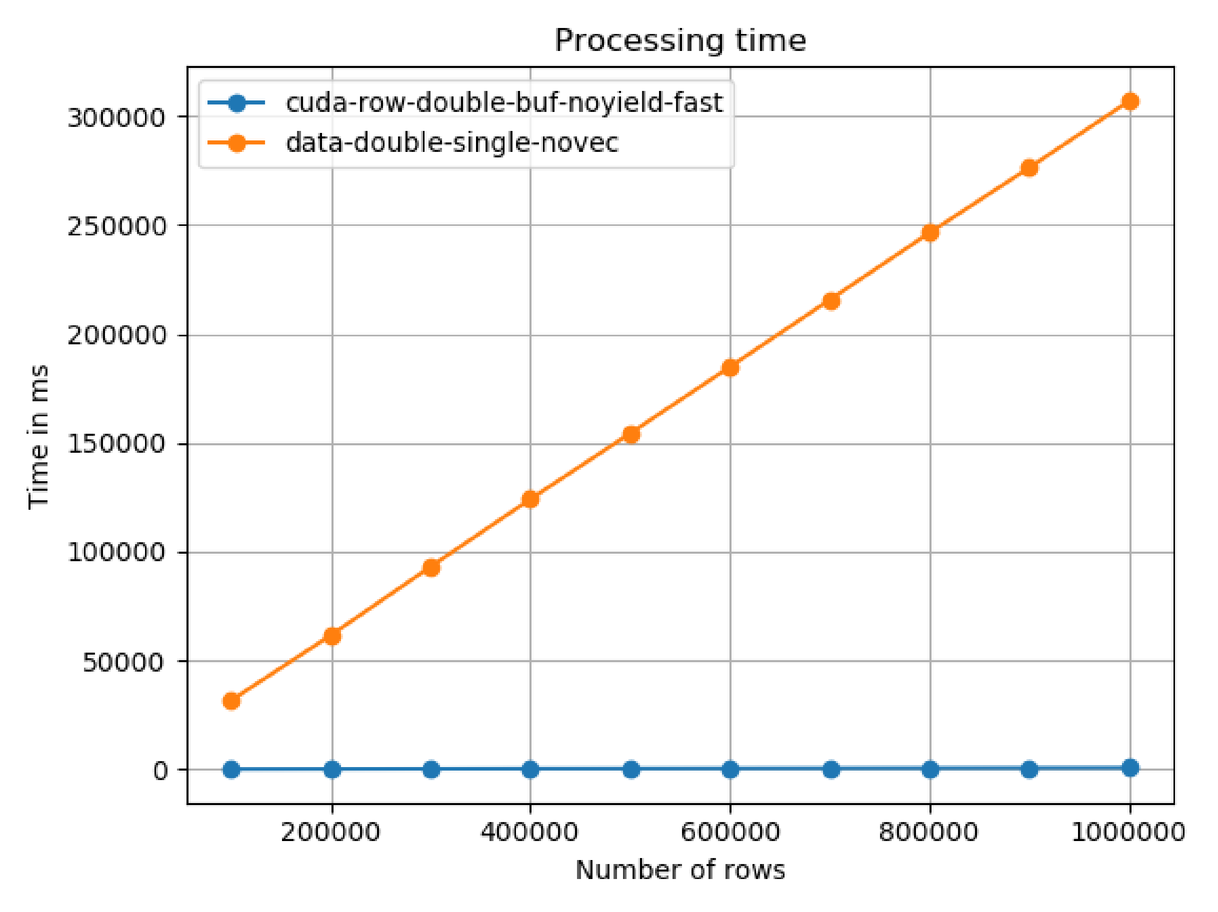
- All benchmarks and measures have been performed on a remote otherwise idle server with the following specs:
  - GPU: GeForce GTX 1080 Ti
  - CPU: Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz, 8 core
  - RAM: 32GB
  - HDD: ST2000DM001-1ER1 (2TB)
- Performance measurement tools:
  - time: to gather general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

Device query

Device 0: "NVIDIA GeForce GTX 1080 Ti"

• 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
• 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)

Total obtained speedup



Type	Time (ms)	Speedup
Worst	306'832.348	
Best	659.789555	~465.04x

The developed application

Repeately does:

- Read the dataset row by row
- Convert strings to the required floating-point type
- Group together many parsed rows in a single chunk
- Perform computations on the chunk as they are generated and update partial results

In the end:

- Merge the partial results and compute the PCC for each column pair
- Print the results

Algorithm: for each column pair

Initially:

Pre-allocate a struct to hold partial sums and rows count.

For each chunk:

Count rows number.

Sum values, squared values and product values.

Finally:

Merge partial results obtained by all worker threads.

Calculate means and the last step to obtain covariance.

First developed application

Single-thread C++ application producing chunks of 1000 stored as a std::vector of std::vector and immediately analysing them.

Data in the same column are stored in the same std::vector.

Performances were ugly despite using vectorized instructions:

- ~55s to analyse 256 columns per 1M rows

Code optimizations

- 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

Execution bottlenecks

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 algorithm

- Computation bottleneck: >50% execution time

On CPU improvement

- 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.

First GPU solution

Single CPU thread

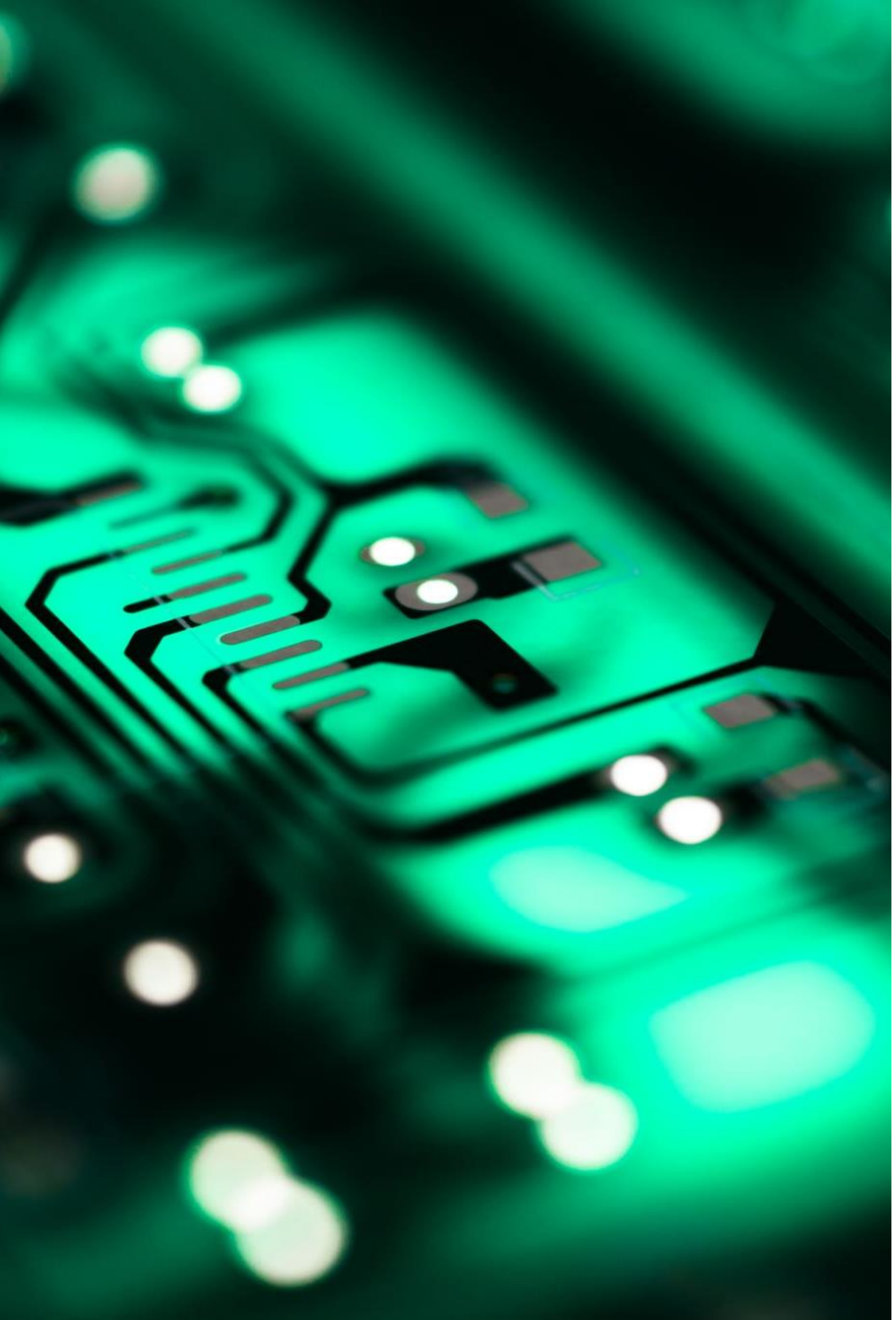
Chunks are processed as they are filled

Chunks are organised as a vector of vectors

GPU is used to process as much data as possible in parallel

Problem: bad memory management

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

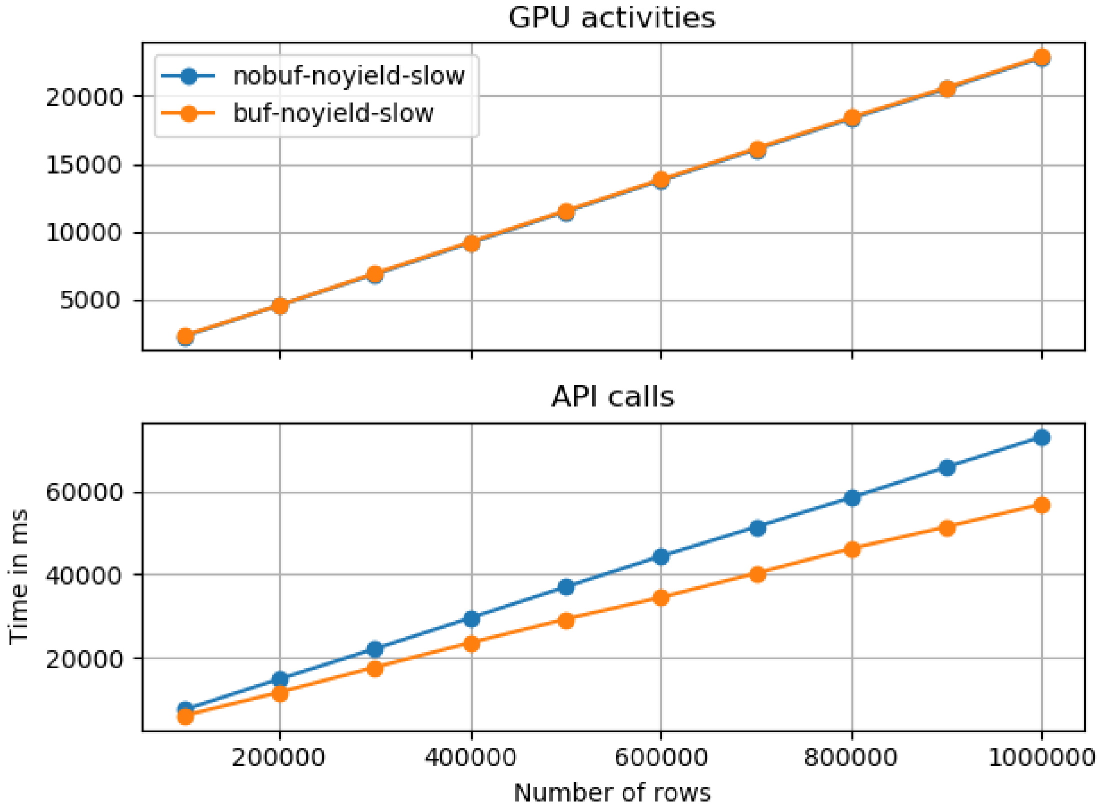


Improve GPU memory management

- Allocating and freeing memory on GPU is a very expensive task!
- 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

Reusing the same CUDA buffer

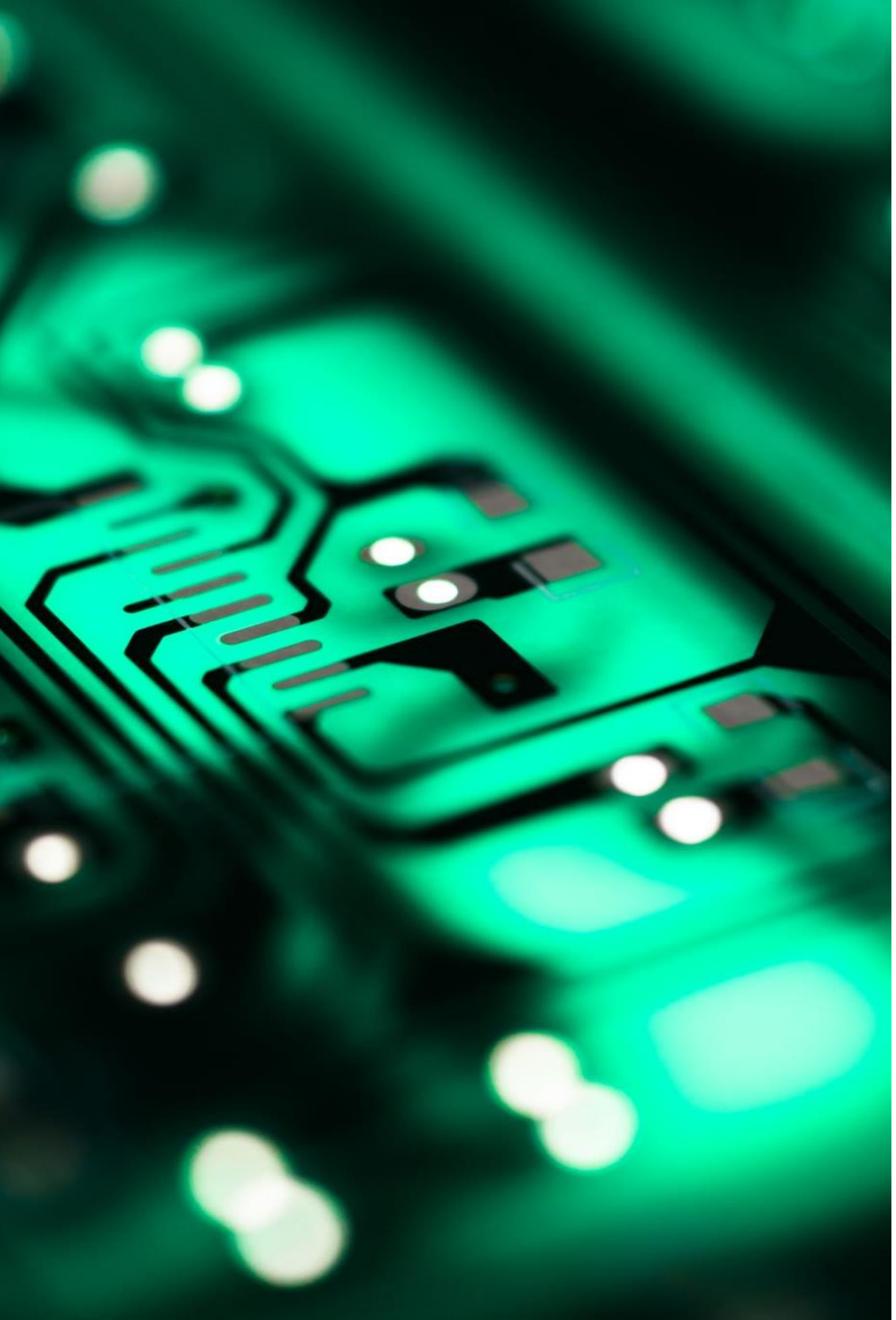
cuda-col-double-nobuf-noyield-slow cuda-col-double-buf-noyield-slow



Calls	Name
1004	CudaMalloc
1004	CudaFree

Calls	Name
8	CudaMalloc
8	CudaFree

API calls speedup: ~1.43



Algorithm Optimization

- Penalty:
  - Recalculating for each column the sum and the sum of squares for every column pair introduces a lot of unnecessary computations
- Improvement:
  - Calculate the sum and the sum of squares only once for each column

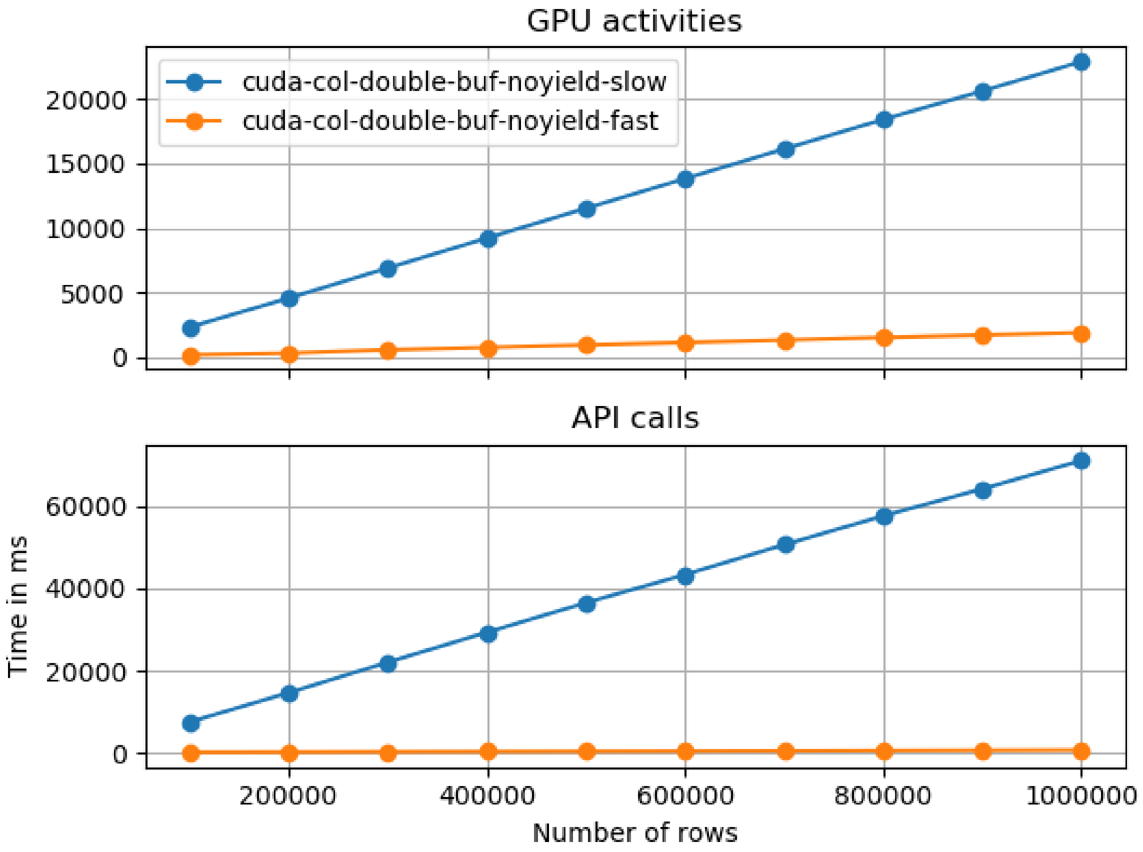
Algorithm Optimization (cont’d)

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

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

Improving the algorithm

cuda-row-double-buf-noyield-slow cuda-row-double-buf-noyield-slow



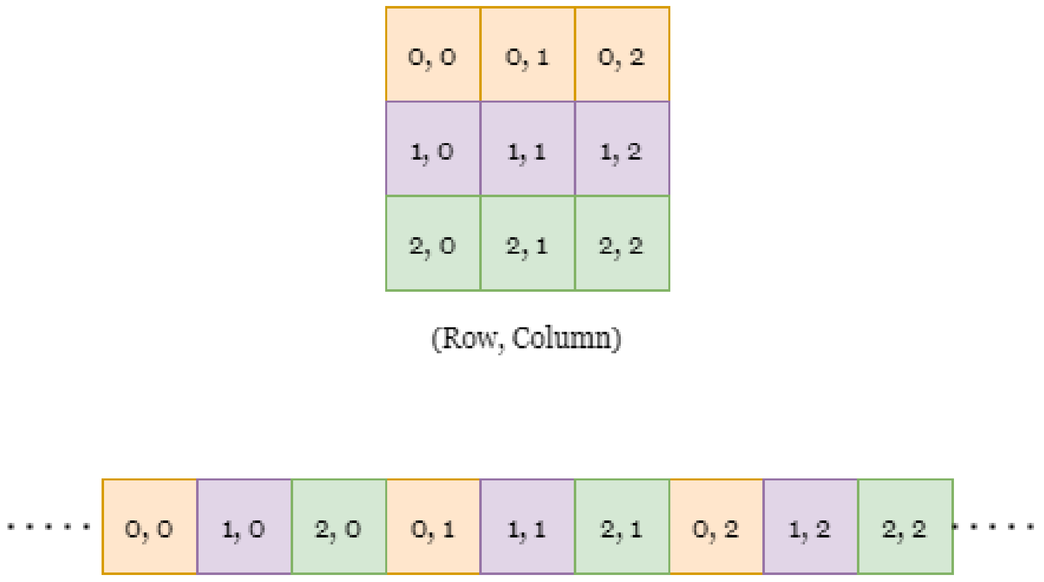
Type	Time (ms)	Speedup
Old	21'815.131146	
New	1'860.50899	~11.73x

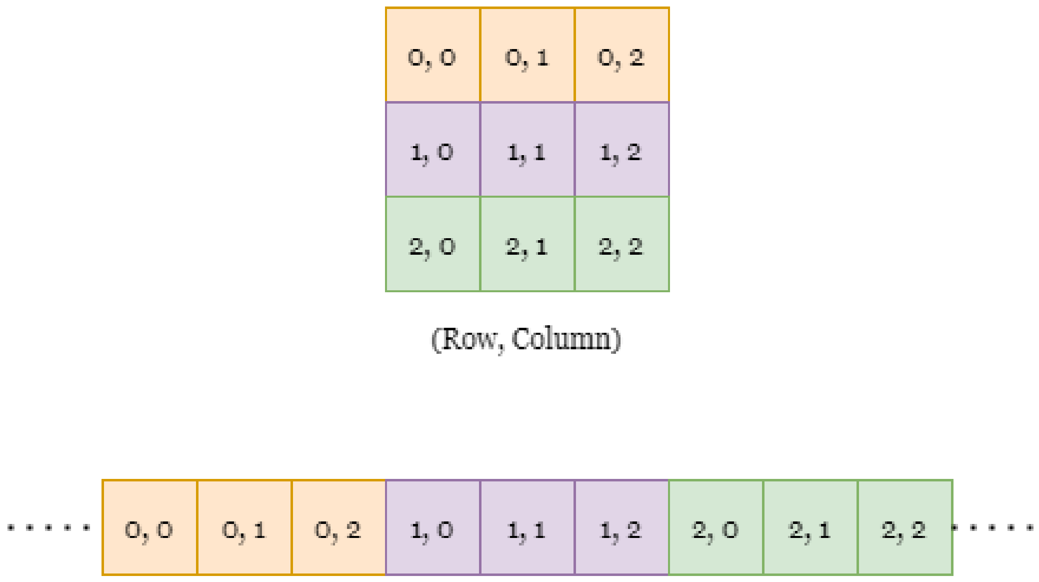
Problem: poor memory load efficiency

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

Storing by columns VS storing by row

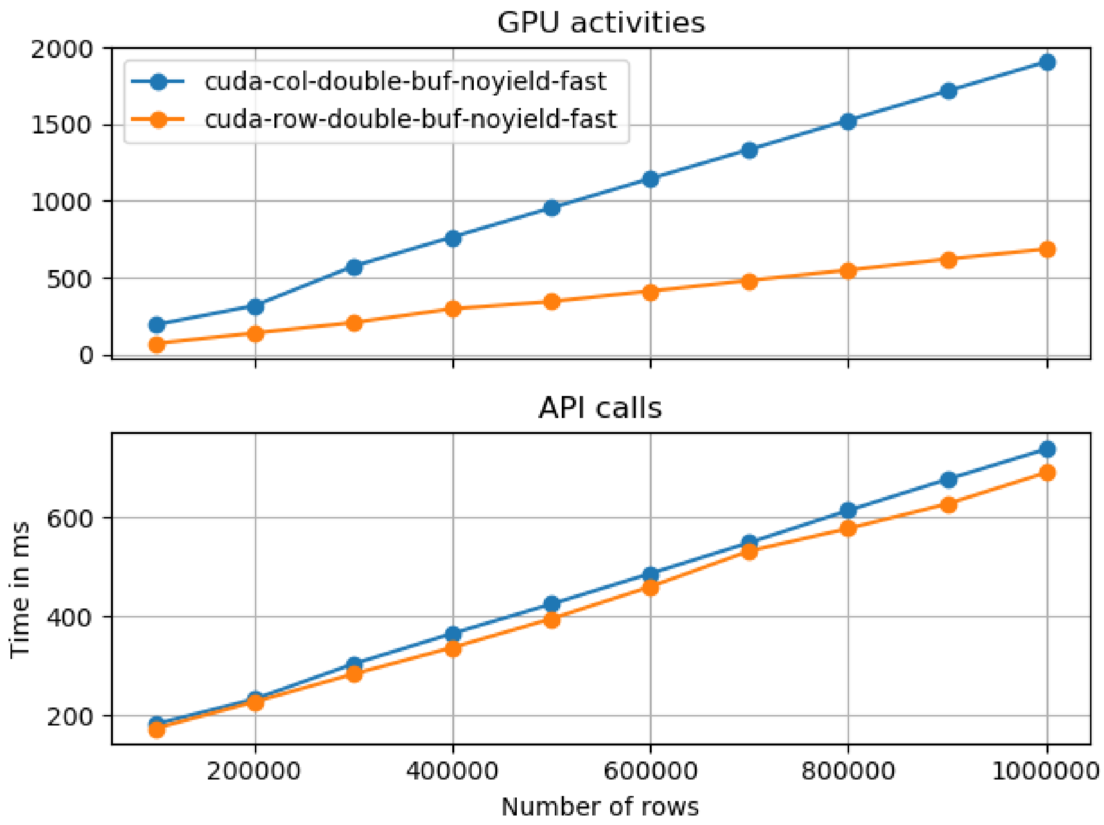
Fortran VS. C matrix memory Layout





Storing by columns vs storing by row

cuda-col-double-buf-noyield-fast cuda-row-double-buf-noyield-fast



Type	Time (ms)	Speedup
Old	1'860.50899	
New	659.789555	~2.82x

Storing by columns vs storing by row (cont'd)

COL

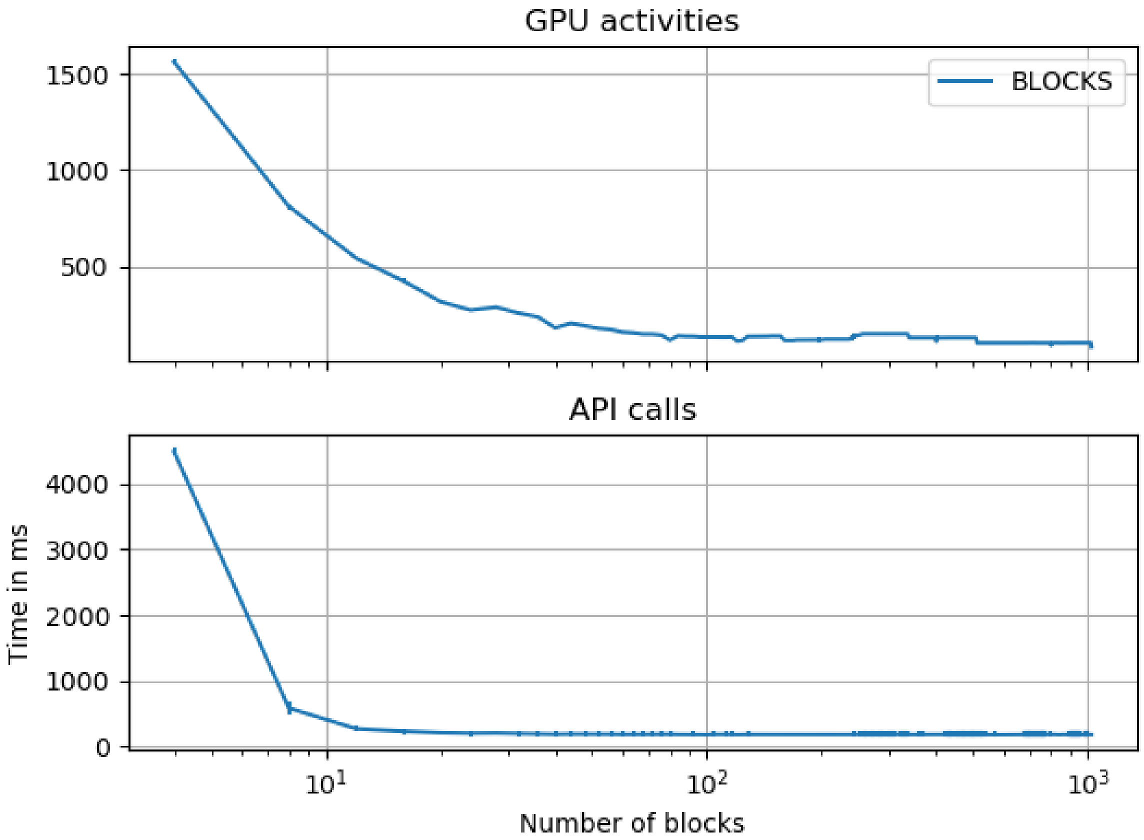
Warp Execution Efficiency	100.00%	100.00%	100.00%
Warp Non-Predicated Execution Efficiency	98.65%	98.65%	98.65%
Shared Store Transactions	0	0	0
Shared Load Transactions	0	0	0
Local Load Transactions	0	0	0
Local Store Transactions	0	0	0
Global Load Transactions	26114	26114	26114
Global Store Transactions	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

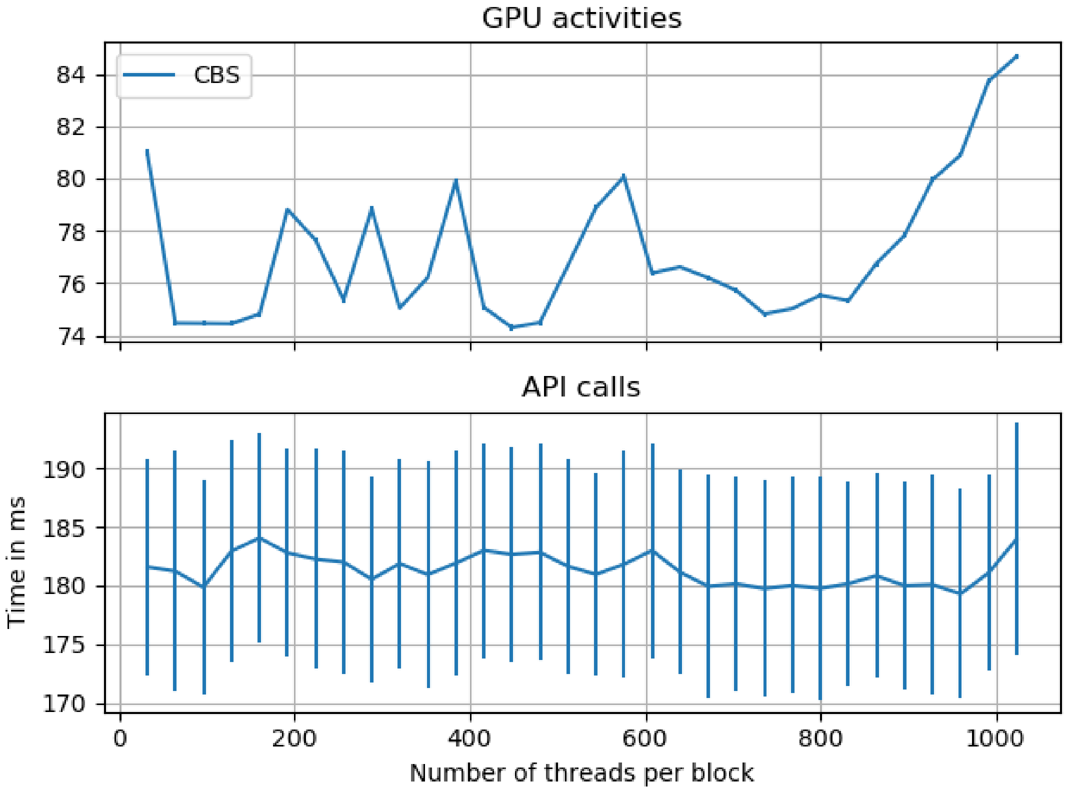
ROW

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

Varying CUDA block number and size

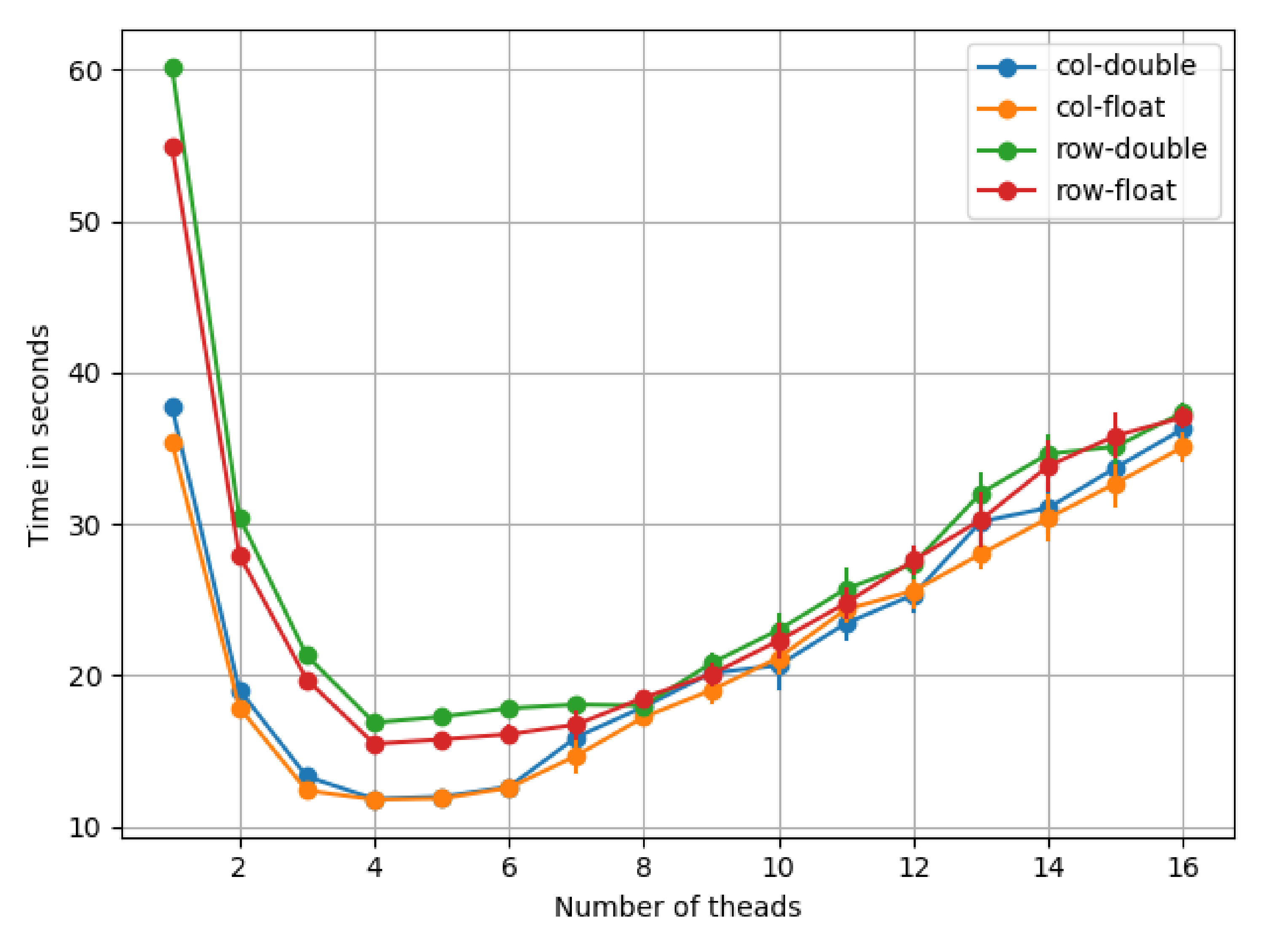
varying blocks | **varying threads per block**

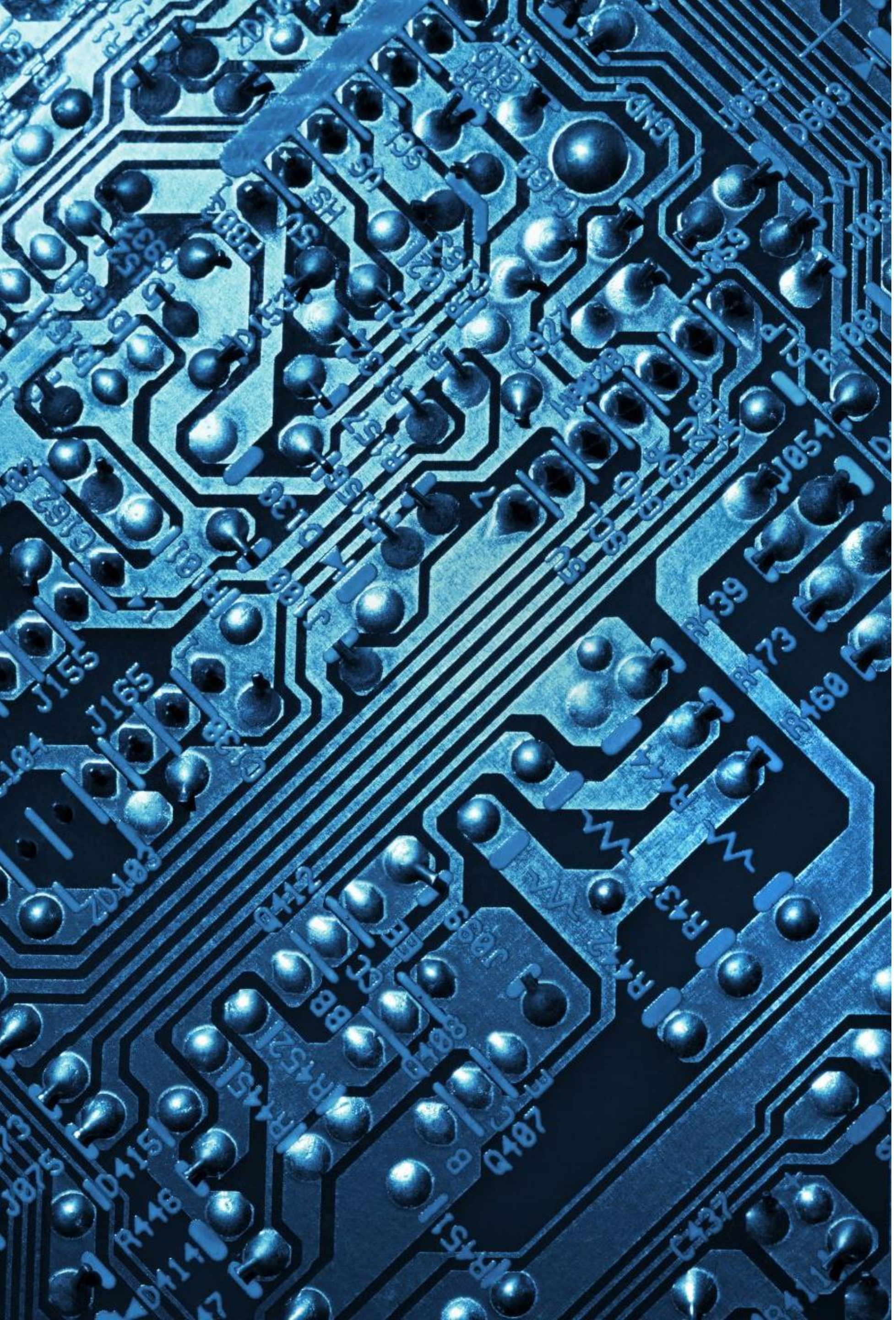




CPU: Intel Hypertrheading penalty

- The tested GPU offered 4 physical core +4 obtained thanks to Intel Hyperthreading technology.
- However, performance decreases when using more threads than physical cores.
- Here is show total execution time of CPU only solutions.





Conclusion

- The total obtained speedup in PCC calculus is x465 time faster, by:
  - Storing chunks continuously in a single buffer
  - Reusing GPU memory buffers
  - Performing computations on GPU in parallel
  - Optimizing memory layout
- Input parsing has been naturally enhanced by using CPU parallelism