Pattern Recognition with C++ OpenMP and CUDA

Giulio Bazzanti Niccolò Biondi

Parallel Computing, April 2020

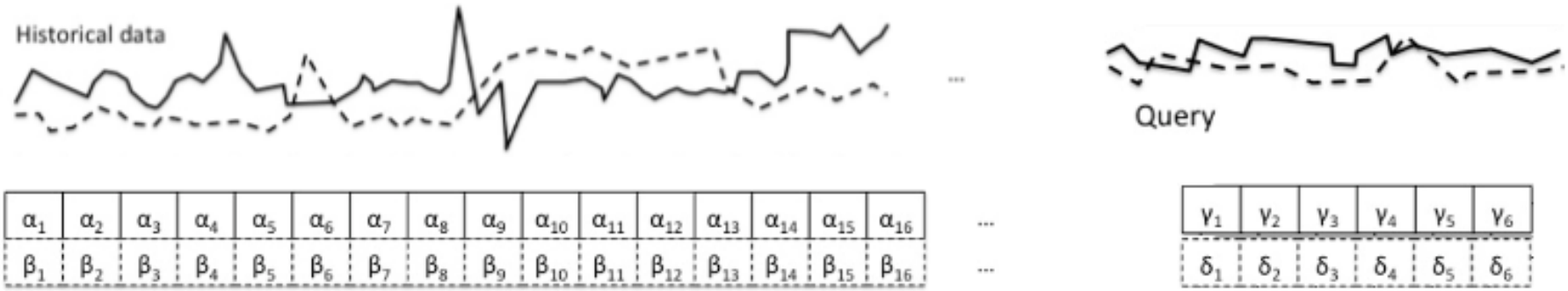
Table of Contents

Introduction

SAD and Pattern Recognition

Pattern Recognition

- Search some Queries inside Historical data
- The goal is to find the data subset which matches query
- Difficult to find the exact subset



Parallel Pattern Recognition

- Sum of absolute differences (SAD)
  - Similarity measure between data subsets
  - Calculated by the absolute difference between each subset values
  - Find the *nearest* data subset
- SAD computation for Patter Recognition can be parallelized, since

$$R_q[i] = \sum_{j=0}^{len(q)} |D[i+j] - q[j]|$$

- Two different parallel implementations:
  - CUDA
  - OpenMP

Sequential

Sequential approach

Algorithm 1 Pattern Recognition

1: Read the Data
2: Read all the Queries
3: Create a vector R with length equal to Queries number
4: **for** query q in Queries **do**
5: Create a vector R_q with length len(Data) - len(q) + 1
6: Compute $R_q[i] = \sum_{j=0}^{len(q)} |D[i+j] - q[j]|$
7: $R[q] = arg \min R_q$
8: **end for**
9: return R

Sequential approach

- Analyze all queries
- Scrolls across the Historical Data
- Has to find min SAD value for each query
- Total computation time $\Rightarrow O(nrq) + O(nq)$

Problems

- very long Historical Data
- a lot of queries

OpenMP

OpenMP approach

- Use OpenMP library to leverage the CPU multi-threads computation
- Use OpenMP *for* loop, *critical* section, *parallel* section
- Two different parallelism levels
  - Parallelism on queries
  - Parallelism on data

Parallelism on queries

- Is the higher parallelism level in Pattern Recognition
- Each thread computes one query thought a OpenMP *for* loop
- Each thread analyzes its query like in the Sequential approach
- Each thread computes the min SAD value for the query

Parallelism on data

- Use *parallel* section to analyze Historical Data in parallel
- Each thread computes SAD values w.r.t. the same Query
- Threads wait the other threads termination through implicit *barrier*
- Two different approaches
  - Privatization method
  - Lock method

Parallelism on data

Privatization method

- Private variables for each thread
- Thread stores min SAD privately
- Each result vector (R_q) length depends on threads number

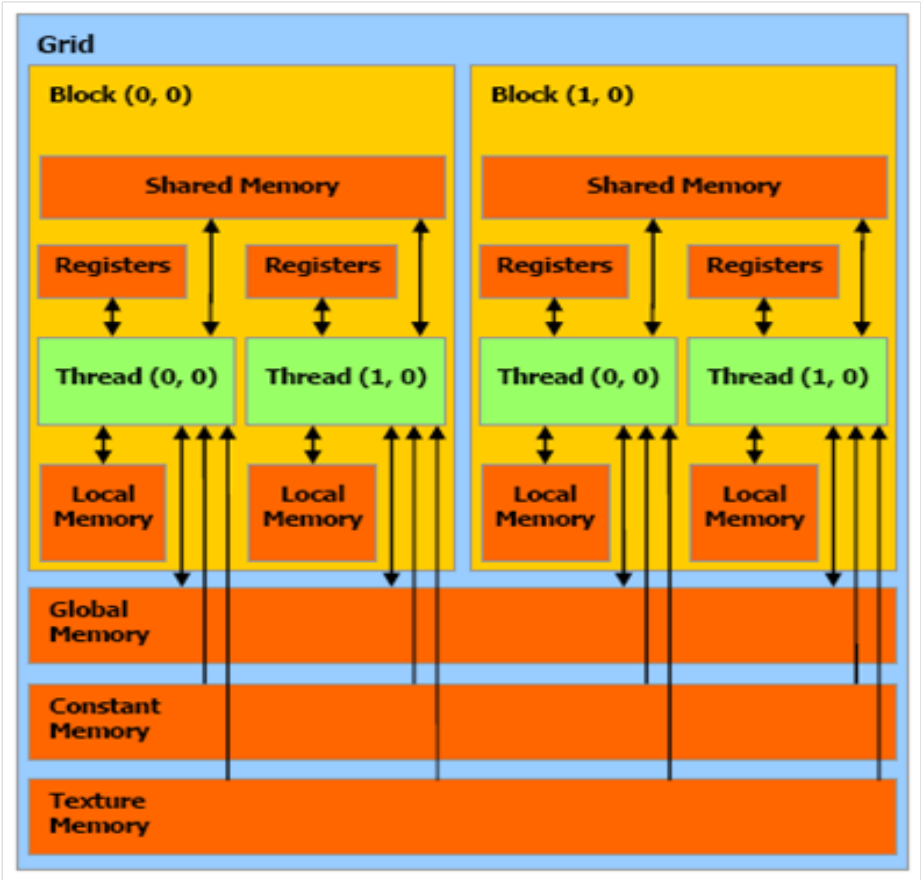
Lock method

- Measures how well-utilized the processor are
- Shared variable stores query min SAD
- Critical section (with *flush*) to write SAD values

CUDA

CUDA approach

- Each block has a Shared memory
- Each tread has a personal and block id
- The total blocks number depends on Historical Data length and threads per block
- Four different implementations



Naive Implementation

- The Queries and Historical Data are stored in Global memory
- Each thread reads a Historical Data chuck and compare it with all queries
- The Result vector is in global memory

Problem: a lot of reads/writes in/from Global memory

Private Implementation

- The Queries and Historical Data are stored in Global memory
- Each thread has a private variable
- The threads compute SAD values w.r.t. its data chunk
- Make local SAD computations and write once

Problem: a lot of reads in Global memory

Constant Implementation

- Use the Constant memory to store the Queries
- In each block Shared memory is written a Historical Data chunk
- Each block thread loads a value from Historical Data
- A thread can see only a fragment of data to compute SAD
  - Each thread in each block compute the remaining SAD values for a query
  - Possible race condition $\Rightarrow$ *atomic* write

$$R[idx - r + q \cdot len(R)] += Ds[threadId] - Qc[r + q \cdot len(Q)]$$

Tiling Implementation

- Both Historical Data and Queries are stored in Shared memory
- Each thread loads in Shared memory Historical Data and Queries chunks
- For each Historical Data chuck threads compute SAD w.r.t. all the Queries
  - Need multiple Queries loads in Shared memory (at least 1 for each Query)
  - Different Query portions loaded in the same block Shared memory ($t < q$)
- Possible race condition $\Rightarrow$ *atomic* write

$$R[idx - r + p \cdot T + q \cdot len(R)] += |Ds[threadId] - Qs[r]|$$

Experiments

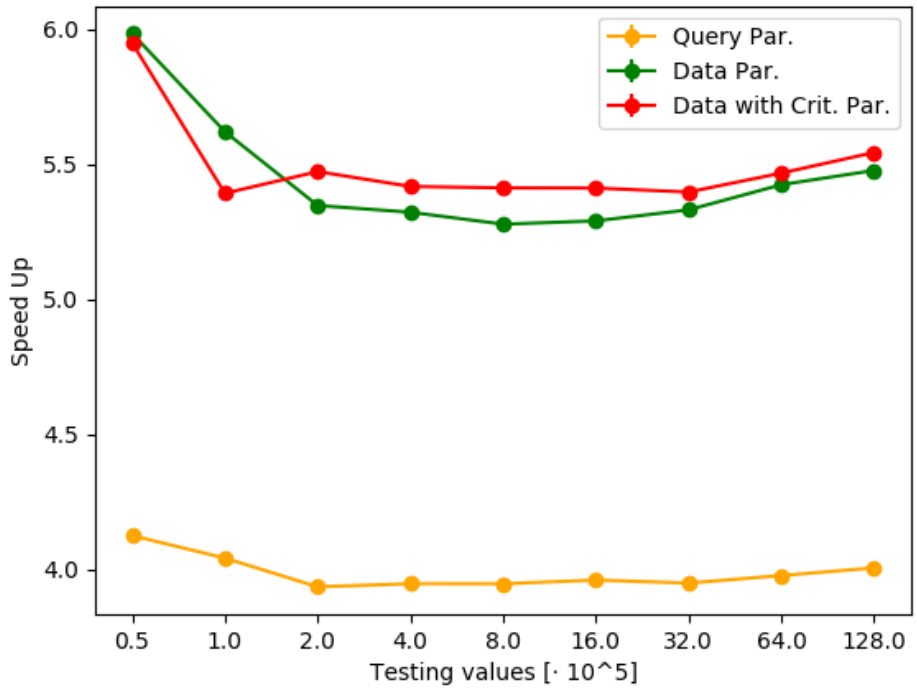
Testing Hypothesis

- One Historical Data vector and multiple Queries
- Pattern Recognition implementations can be testing on:
  - Variable Historical Data length
  - Variable Query length
  - Variable Queries number
  - Variable CPU and GPU threads number

Hyper-parameter	Default Value
Historical Data	100000
Query	1000
Queries Number	10
Threads Number	12
Kernels per Block	128

- For each test configuration are performed 10 run for stable results

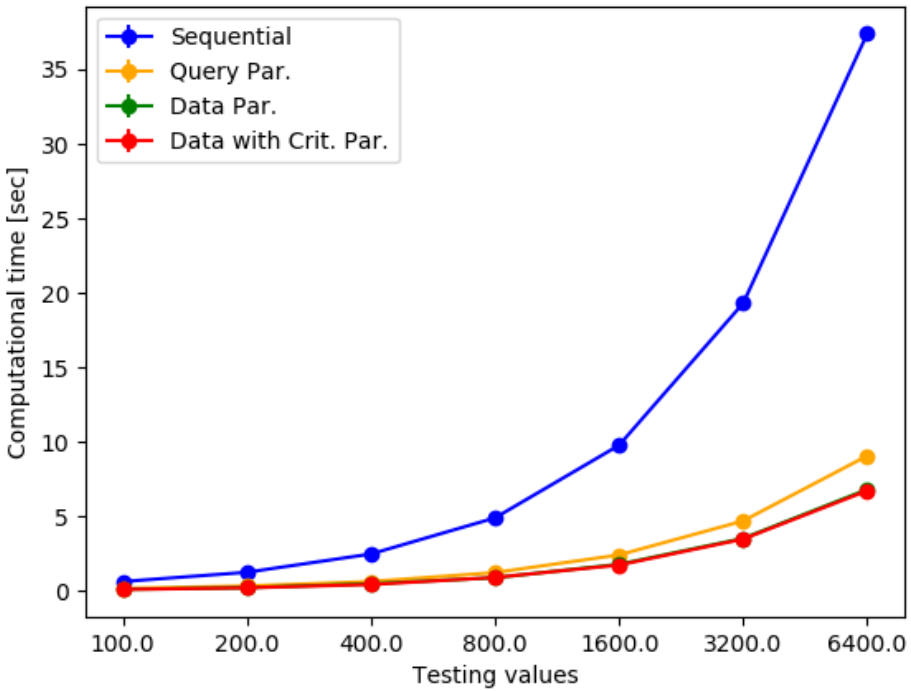
Historical Data length [50000, 1280000]

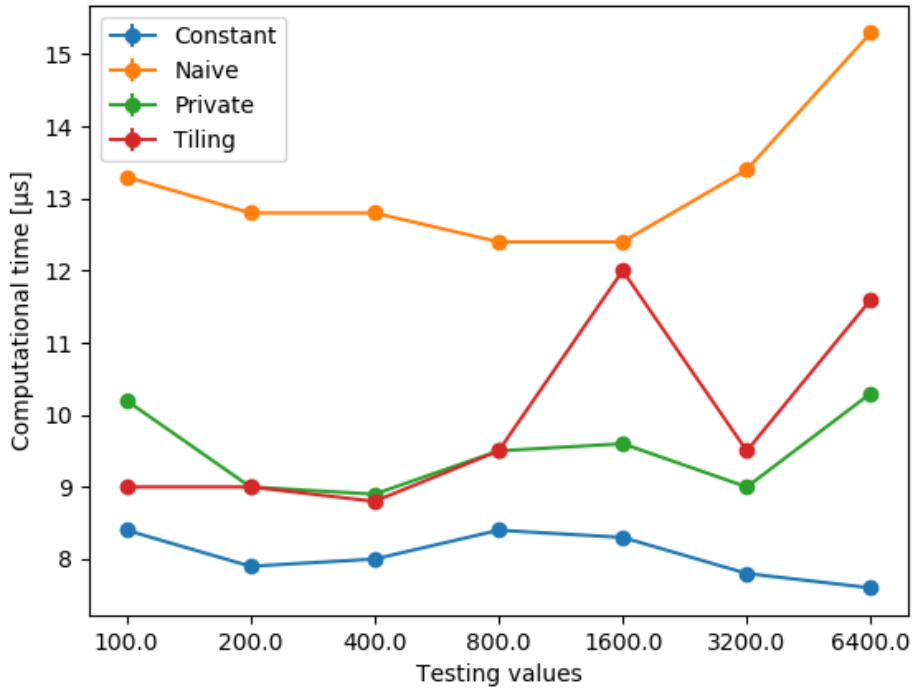


len(Data)	Sequential [*sec*]	OpenMP [*sec*]	CUDA [μsec]
50000	$3.20 \pm 0.37\%$	$0.53 \pm 0.02\%$	$7.6 \pm 2.57\%$
200000	$12 \pm 1.03\%$	$2.24 \pm 0.52\%$	$8.4 \pm 2.61\%$
800000	$47.8 \pm 0.02\%$	$9.05 \pm 1.02\%$	$13 \pm 3.46\%$
320000	$191 \pm 0.34\%$	$36 \pm 1.21\%$	$12.4 \pm 2.24\%$
1280000	$780 \pm 6.58\%$	$142 \pm 2.57\%$	$14.4 \pm 3.16\%$

Query length

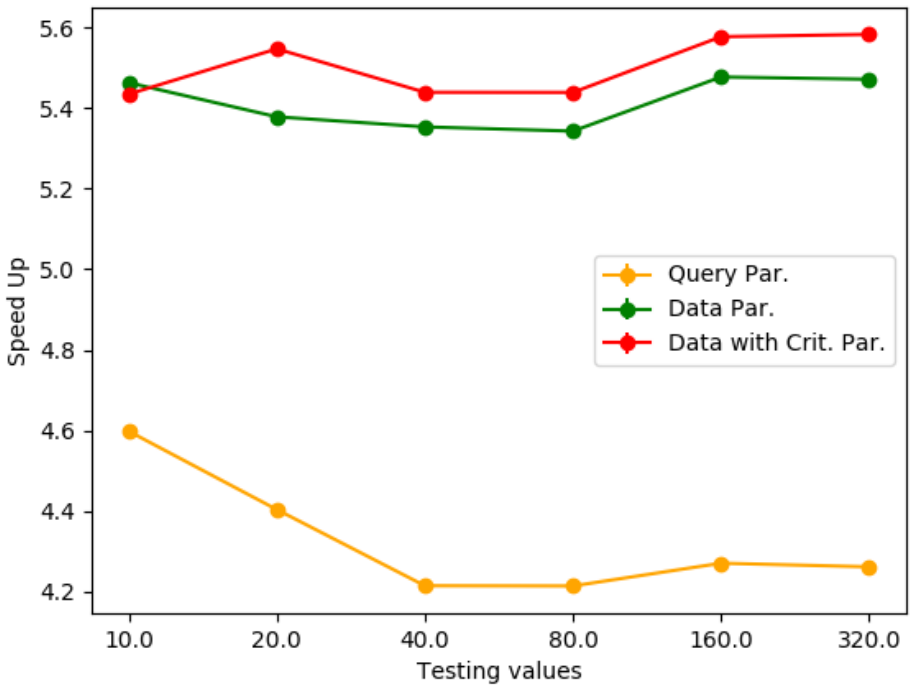
- Query length from 100 to 6400
- CPP computational times is reported in *sec*
- GPU ones in μsec

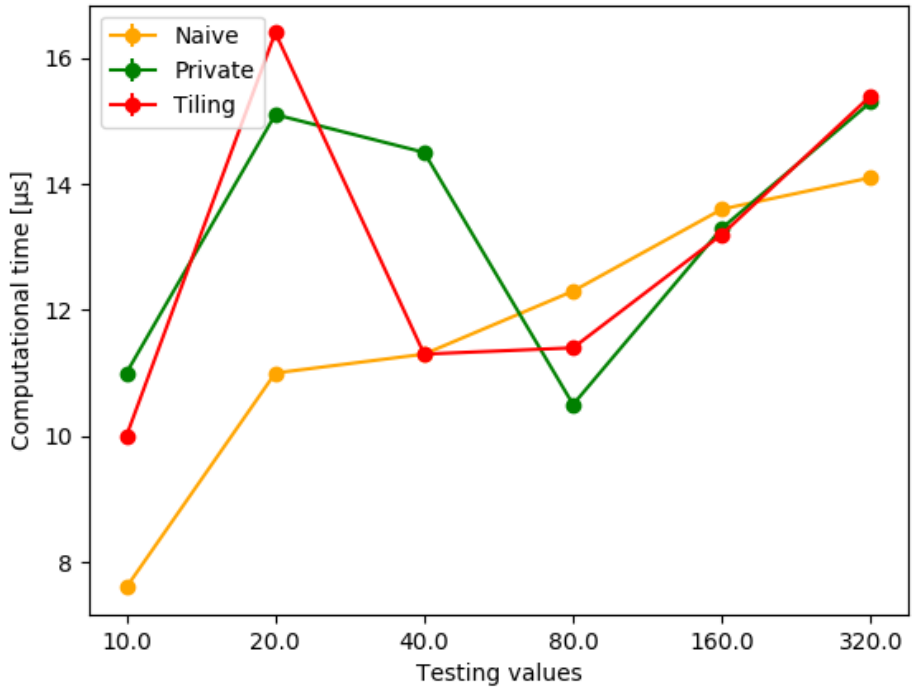




Queries number

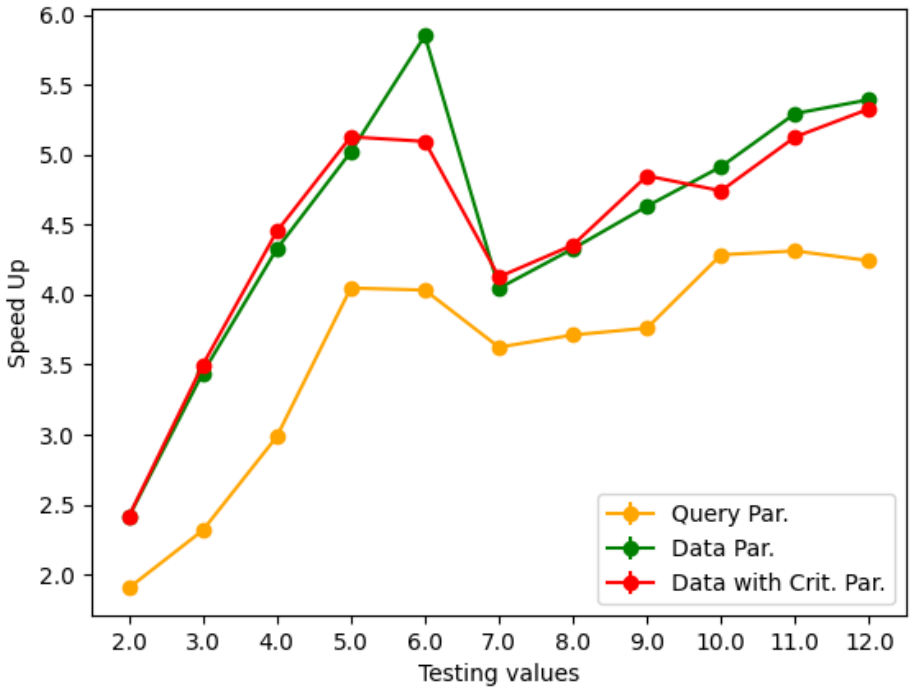
- Queries number in $[10, 320]$
- CUDA Constant implementation can be tested

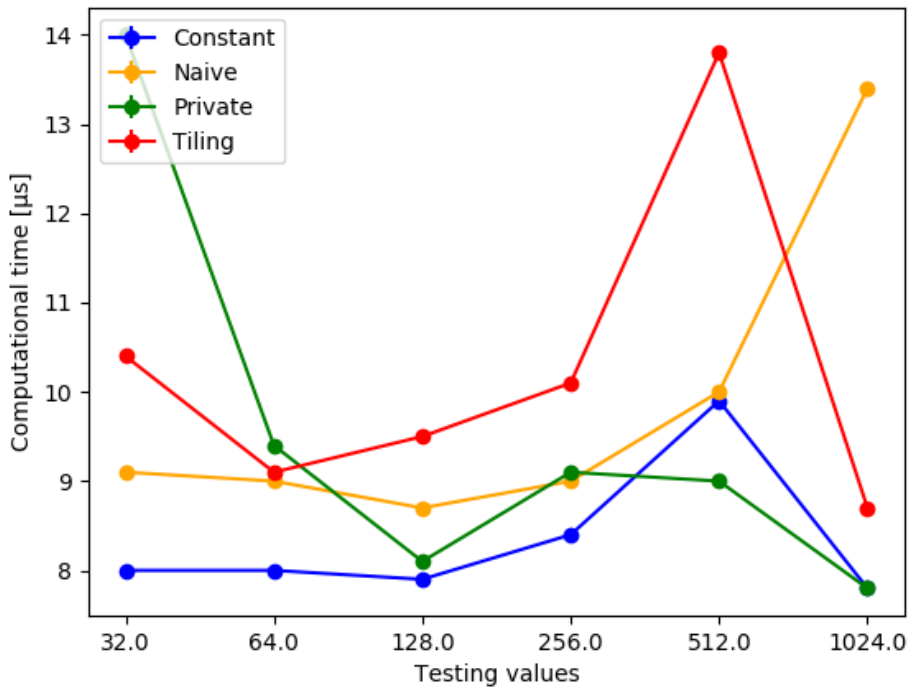




CPU and GPU threads number

- Threads number from 1 (for Relative Speedup) to 12
- GPU kernels per block from 32 to 1024 (device maximum)





Evaluation Metrics

	OpenMP			*CUDA*			
Metric	Query	Privatization	Lock	Naive	Private	Tiling	Constant
S_p	4.03	5.85	5.09	$7.1 \cdot 10^5$	$7.6 \cdot 10^5$	$6.5 \cdot 10^5$	$7.8 \cdot 10^5$
RS_p	4.37	4.48	4.15	(-)	1.07	0.91	1.10
E	0.67	0.97	0.85	7.09	7.59	6.49	7.79

- Reported performances for the parallel Pattern Recognition solutions. All the values are computed with a 100000 Historical Data vector, 1000 Query length, 10 total Queries, 6 CPU threads number and 128 CUDA kernels for each block.
- Relative Speedup is computed w.r.t. the baseline methods (Sequential and Naive)
- The values are the mean obtained in 10 different runs

Conclusions

Conclusions

- Experimental results underline the gap between CPU and GPU parallel implementations
- OpenMP Privatization method reports almost linear Speedup and Efficiency close to 1
- CUDA show its parallelism power in all the implementations, in particular thanks to the Shared and Constant memories

Thanks for the attention