Pattern Recognition with C++ OpenMP and CUDA

Giulio Bazzanti Niccolò Biondi

Parallel Computing, April 2020

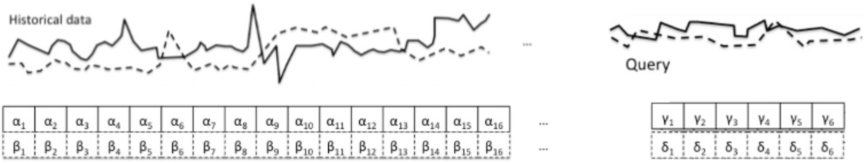
Table of Contents

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

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

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

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

Parallel solutions

- CUDA
- OpenMP

Sequential

Sequential approach

Algorithm 1 Pattern Recognition

```
Generate the Historical Data
Generate all the Queries
Create a vector R with length equal to Queries number
for query q in Queries do
    Create a vector R_q with length len(Data) - len(q) + 1
    Compute R_q[i] = sum_{j=0}^{len(q)} |D[i + j] - q[j]|
    R[q] = arg min R_q
end for
return R
```

Where:

- Line 5: Create a vector R_q with length len(Data) - len(q) + 1
- Line 6: Compute $R_q[i] = \sum_{j=0}^{len(q)} |D[i+j] - q[j]|$
- Line 7: $R[q] = arg \min R_q$

Sequential approach

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

Sequential approach

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

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

Parallelism levels

- Parallelism on Queries
- Parallelism on Historical 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*

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

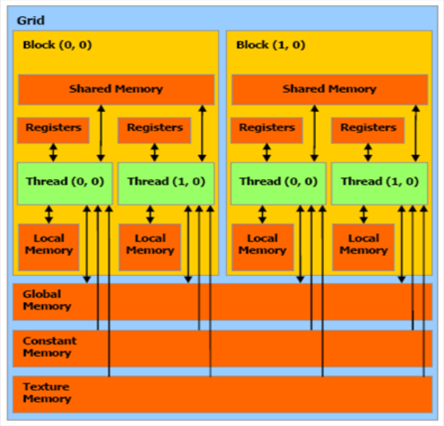
Lock method

- No minimum SAD value search
- 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

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

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

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

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

Constant Implementation

- Use the Constant memory to store the Queries
- In each block Shared memory is written a Historical Data chunk
- 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

$$\textbf{if } (idx - r) \geqslant 0 \textbf{ and } (idx - r) < len(R)$$
$$R_q[idx - r] += |Ds[threadId] - Qc_q[r]|$$

Constant Implementation

- Use the Constant memory to store the Queries
- In each block Shared memory is written a Historical Data chunk
- 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

$$\textbf{if } (idx - r) \geqslant 0 \textbf{ and } (idx - r) < len(R)$$
$$R_q[idx - r] += |Ds[threadId] - Qc_q[r]|$$

Possible race condition $\Rightarrow$ *atomic* write

Tiling Implementation

- Both Historical Data and Queries are stored in Shared memory
- Each thread loads in Shared memory Historical Data and Queries chunks

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 ($t < q$)

$$\textbf{if } (idx - (r + p \cdot T)) \geqslant 0 \textbf{ and } (idx - (r + p \cdot T)) < len(R)$$

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

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 ($t < q$)

$$\textbf{if } (idx - (r + p \cdot T)) \geqslant 0 \textbf{ and } (idx - (r + p \cdot T)) < len(R)$$
$$R_q[idx - (r + p \cdot T)] += |Ds[threadId] - Qs[r]|$$

Possible race condition $\Rightarrow$ *atomic* write

Experiments

Testing Hypothesis

One Historical Data vector and multiple Queries

- Pattern Recognition tested on:
  - Variable Historical Data length
  - Variable Query length
  - Variable Queries number
  - Variable CPU and GPU threads

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

Testing Hypothesis

One Historical Data vector and multiple Queries

- Pattern Recognition tested on:
  - Variable Historical Data length
  - Variable Query length
  - Variable Queries number
  - Variable CPU and GPU threads

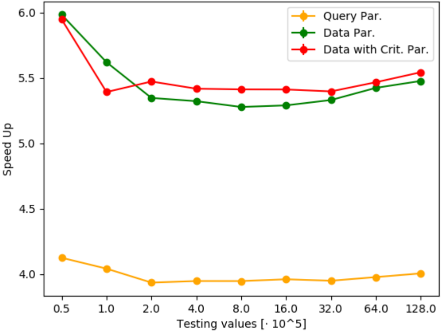
Hyper-parameter	Default Value
Historical Data	100000
Query	1000
Queries Number	10
Threads Number	12
Kernels per Block	128

Stabilize results

For each test configuration are performed 10 run for stable results

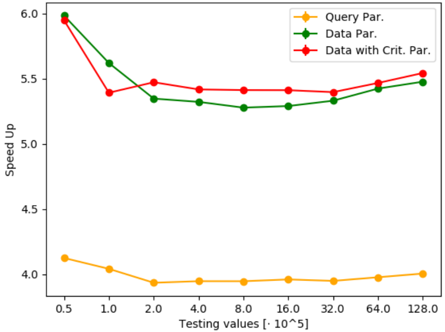
Historical Data length

- Historical Data with size in $[50000, 1280000]$



Historical Data length

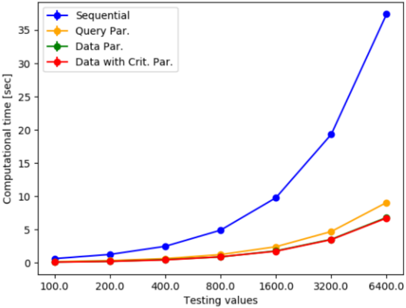
- Historical Data with size in $[50000, 1280000]$
- Table compares the OpenMP and CUDA implementations w.r.t. the Sequential one (with mean and std)

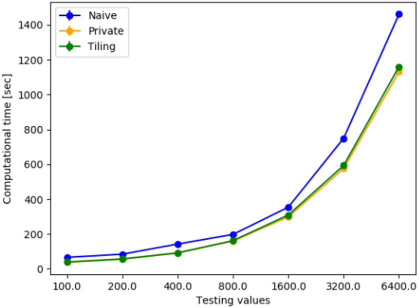


len(Data)	Sequential [*sec*]	OpenMP [*sec*]	CUDA [μsec]
50000	$3.20 \pm 0.37\%$	$0.53 \pm 0.02\%$	$178.6 \pm 5.81\%$
200000	$12 \pm 1.03\%$	$2.24 \pm 0.52\%$	$181.3 \pm 5.51\%$
800000	$47.8 \pm 0.02\%$	$9.05 \pm 1.02\%$	$171.2 \pm 8.61\%$
320000	$191 \pm 0.34\%$	$36 \pm 1.21\%$	$182.8 \pm 2.91\%$
1280000	$780 \pm 6.58\%$	$142 \pm 2.57\%$	$181.3 \pm 3.72\%$

Query length

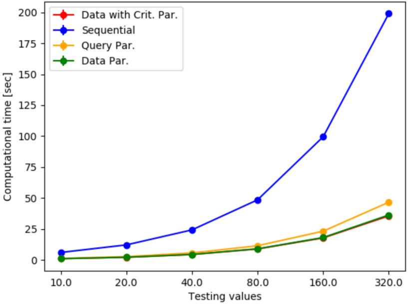
- Query length from 100 to 6400
- CPP computational times reported in *sec*; GPU ones in μsec
- No parallelism on queries $\Rightarrow$ **proportional** grown

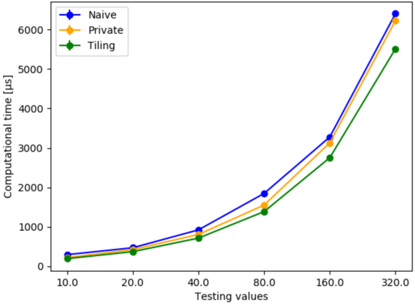




Queries number

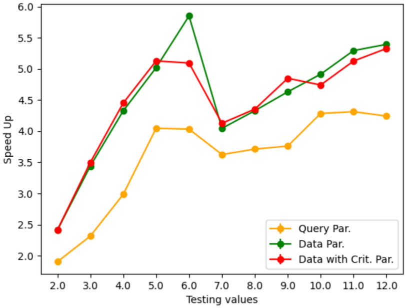
- Queries number in $[10, 320]$
- CUDA Constant implementation can not be tested

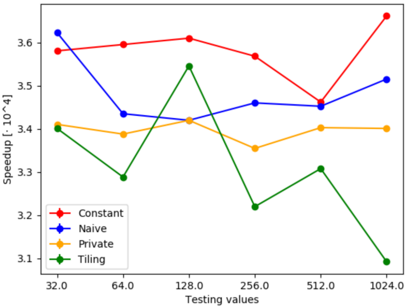




CPU and GPU threads number

- 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	5.87	5.09	$2.2 \cdot 10^4$	$2.8 \cdot 10^4$	$3.0 \cdot 10^4$	$3.4 \cdot 10^4$
RS_p	4.37	4.48	4.15	(-)	1.24	1.34	1.51
E	0.67	0.97	0.85	0.22	0.28	0.30	0.34

- 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

Evaluation Metrics

	OpenMP			*CUDA*			
Metric	Query	Privatization	Lock	Naive	Private	Tiling	Constant
S_p	4.03	5.87	5.09	$2.2 \cdot 10^4$	$2.8 \cdot 10^4$	$3.0 \cdot 10^4$	$3.4 \cdot 10^4$
RS_p	4.37	4.48	4.15	(-)	1.24	1.34	1.51
E	0.67	0.97	0.85	0.22	0.28	0.30	0.34

- 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

Evaluation Metrics

	OpenMP			*CUDA*			
Metric	Query	Privatization	Lock	Naive	Private	Tiling	Constant
S_p	4.03	5.87	5.09	$2.2 \cdot 10^4$	$2.8 \cdot 10^4$	$3.0 \cdot 10^4$	$3.4 \cdot 10^4$
RS_p	4.37	4.48	4.15	(-)	1.24	1.34	1.51
E	0.67	0.97	0.85	0.22	0.28	0.30	0.34

- 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

Evaluation Metrics

	OpenMP			CUDA			
Metric	Query	Privatization	Lock	Naive	Private	Tiling	Constant
S_p	4.03	5.87	5.09	$2.2 \cdot 10^4$	$2.8 \cdot 10^4$	$3.0 \cdot 10^4$	$3.4 \cdot 10^4$
RS_p	4.37	4.48	4.15	(-)	1.24	1.34	1.51
E	0.67	0.97	0.85	0.22	0.28	0.30	0.34

- 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