Pattern Recognition with C++ OpenMP and CUDA

Giulio Bazzanti, Niccolò Biondi
Dipartimento di Ingegneria dell'Informazione (DINFO)
University of Florence, Florence, Italy
giulio.bazzanti@stud.unifi.it, niccolo.biondi1@stud.unifi.it

***Abstract*—Pattern Recognition is the task of comparing some queries to some data searching for the most similar matches between them. Those matches are performed according to a measure, such as the Sum of Absolute Differences (SAD). In this review we exploit how this task can leverage CPU (OpenMP) and GPU parallelism (CUDA). We report our experimental results comparing several implementations with a sequential one (C++) evaluating trough Speedup and Efficiency measures.**

***Index Terms*—Pattern Recognition, Parallel Computing, OpenMP, CUDA, Performances Evaluation Metrics**

I. Introduction

Pattern Recognition is the task where some queries must be searched inside a Historical Data. When this data is 1D this task is referred to signal processing, while 2D Pattern Recognition is often performed on images, where it can be used as baseline for other techniques, such as Template Matching. The goal of Pattern Recognition is to find the subset of data which matches with each query. Since it's difficult that this subset exists, we adopt the *Sum of Absolute Difference* (*SAD*) metric to measure the similarity between data and query. In this way we can find the *nearest* subset of data for each query.

In this paper we study the effectiveness of various Patter Recognition implementations: starting from a sequential one (*C++*), we develop some CPU (*OpenMP*) and GPU (*CUDA*) parallel methods. In order to examine the improvements of those parallel solutions, we compare the reported performances in term of Speedup, Relative Speedup and Efficiency.

II. Methods

In this section we present our Pattern Recognition solutions, from the generation of data (both Historical and Queries ones) to the experiments configuration. In addiction we also briefly describe the different proposed parallelism techniques, OpenMP and CUDA.

A. Data Generation

Pattern Recognition requires a Historical Data where we need to find the most similar pattern to some Queries in according to the SAD metric, like is shown in Figure 1. To generate these data we sample them from an uniform distribution with a seed for easily reproduce experiments and check if every implementation reports the same SAD value and same Historical Data subset.

The process of data generation differs if data has to be fed to CPU or GPU. In the former scenario we generate random vector of integers, such that $v \in [0, 99]$, for both Historical Data and Queries. In the latter case we create data vectors directly on device without storing in host memory, thanks to a CUDA library named *Curand*. This API makes the program faster because there are no needs to copy data from host to device, that is the typical GPU program bottleneck. In this case Historical Data and Queries values are floating point because they are sampled from an uniform distribution with $[0, 1]$ support.

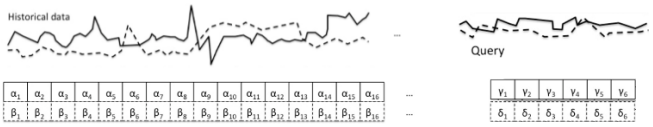

Fig. 1: Pattern Recognition task. Its aim is to find the most similar subset of the Historical Data that matches with each Query according to SAD metric.

B. Proposed Implementations

The Pattern Recognition task takes in input one Historical Data and some Queries. The goal is to find patterns in data that match with each Query. To validate these similarities we adopt the SAD metric that compares each Query with every Historical Data sub-sequences with the same Query length. Thus, we compute all those SAD values and we store these results in some vectors, one different for each Query, with the following dimension:

$$len(Result) = len(Data) - len(Query) + 1 \tag{1}$$

The entire process can be summarized as follows:

$$R_q[i] = \sum_{j=0}^{len(q)} |D[i+j] - q[j]| \tag{2}$$

with $i = 0, 1, \ldots, len(Result)$. $R_q[i]$ is the i-th position of the result vector corresponding to the Query q; $D[i]$ is the starting point of the Historical Data subset in which SAD has to be compute with the Query q. From equation 2 we can observe how the Pattern Recognition task can be parallelized: different threads can compute SAD values using different $D[i]$ starting point. In this way, each thread will store its resulting SAD in different position of R_q without race conditions. Given these vectors, we extract the best SAD for one Query as the minimum of the corresponding result vector. In addiction to

it, we can obtain the starting index of the Historical Data subsequence, because it corresponds to the best SAD index in this result vector.

We propose three different solutions of CPU parallelism in OpenMP, four for the GPU computations and one sequential implementation in C++ that we need to evaluate the parallel ones.

III. Sequential Implementation

In this section is presented the sequential implementation, that we develop for the Pattern Recognition task.

Giving one vector for the Historical Data and some Queries, our sequential implementation processes one Query at times (*for* loop) and, for every Query, it creates the corresponding SAD values vector. To do that, we perform another loop on the entire Historical Data vector and, for each element in it, we compute the SAD between the Query and the subset of Historical Data from this element until the last Query value. After that we store it in the result vector. The only aspect to be noticed is that we can perform the loop on the Historical Data for *len(Result)* element, since the last ones does not have so much successors to compute SAD with the Query.

The bottleneck of this implementation is the search of the minimum SAD value in each result vector. This process is performed by looking all the values in every vectors and extracting their minimum. To reach better performances we adopt a C++ library, that extract the minimum value in a vector and its position in it, that is *algorithm*.

The total computational time for this implementation will be $O(nrq) + O(nr)$, where n is the number of Queries, r the result length and q is the length of each Query. $O(nrq)$ is the computational time required for computing all the result vectors and $O(nr)$ for all the minimum SAD searches.

IV. OpenMP

In this section we introduce our three implementations of Pattern Recognition with CPU parallel computations. This kind of parallelism is reached thanks to the OpenMP library, which defines some clauses to leverage the multi-processors computation. In our approach, we define two different parallelism levels: parallelism on Queries and parallelism on data. In addition, for this last type of parallelism, we also study the difference in performance of using critical section or privatization methods.

A. Parallelism on Queries

The high level of parallelism for the Pattern Recognition task is at Queries level. As we described in in Section III, the outer loop is on Queries, so we define an OpenMP *parallel* section in which each thread computes one Query thought an OpenMP *for* loop. For every Query, the thread compute SAD values sequentially and it writes these values on the corresponding result vector. Finally, it recovers the minimum SAD value and its ID (with which we can also extract the corresponding Historical Data subset) for each Query with the same minimum function as in the Sequential implementation. OpenMP requires to specify the scope of the variables used by the function at the beginning of the *parallel* section where the threads will be generated. So, for this implementation, we define all data in input to the function as shared since they are hyper-parameters that every threads has to read (e.g Historical Data and Queries).

B. Parallelism on Data

The other implementations are with data level parallelism. To achieve this, we compute each Query sequentially (with a typical for loop) and we generate threads inside that with an OpenMP *parallel* section. In such way multiple threads can leverage data in parallel with an OpenMP *for* loop on the Historical Data. OpenMP gives to each thread chunks of Historical Data with which it can compute the SAD values corresponding to the Query. When a thread consumes all of its data, it will receive dynamically from OpenMP other chunks to analyze, if they are available. Otherwise, it will wait the end of the other threads works at the (implicit) *barrier* at the end of the *parallel* section.

Here we exploit two different solutions to face with the Pattern Recognition task. The former is with some private variables for each thread (we called it *Privatization method*). The latter is with critical sections (we called it *Lock method*).

OpenMP allows to define some private data for each thread that is what we leverage for our Privatization method. We initialize for every threads three private variables, two for storing the minimum SAD value and its corresponding starting position in the Historical Data that each thread computes and one that stores the thread ID. In this way, each thread can compute the SAD value for the Query and it can store in the first two variables the corresponding minimum SAD and its ID.

At the end of the *for* loop on the Historical Data, that is when every threads consume its data chunks, they write their private values on the Query result vector. The result vector for each Query has now the same length has the number of threads generated by the OpenMP *parallel* section, not longer as we report in 1. This causes that the minimum search will be faster with respect to the other implementations.

We propose also the last OpenMP method that does not store a result vector for every Query, but it stores just the minimum SAD and its position in the Historical Data: the Lock method. To achieve this we exploit OpenMP *critical* sections with which we define a shared variable between all threads used to store the Query minimum SAD value. Since there could be problems of race condition, we define a *critical* section where only one thread at a time can update both the global minimum SAD value and its ID. To avoid a lot of useless access to the *critical* section, we leverage the OpenMP *flush* clause to read the correct value of the variable. Only if a thread finds a smaller SAD value than the stored one, it tries to enter in the *critical* section, because we want that the threads access in this critical section the least possible. This last method avoids the minimum value computation of the result vector, that it is necessary for the other two implementations presented above.

V. CUDA

Here we describe our Pattern Recognition implementations in CUDA, a library that allows us to leverage the power of GPU parallelism. We propose four different methods those exploit some CUDA features, such as the Constant and Shared memory. All of these methods share the generation of the Historical Data and the Queries. Since the most expensive component of a GPU script is the copy of data from CPU to GPU (*cudaMemcpy*), we exploited the *curand* library. Thanks to this we can generate random vectors those are already stored in GPU without any copies. Furthermore we can seed the generator to make our experimental results reproducible, see Section VI. In addiction to data generation, CUDA requires also the definition of some dimensions, such as the number of threads per block and the number of blocks. The former size needs to be multiple of 32, while the latter one is determined by the Historical Data length and the block size as follows:

$$gridDim = \left\lceil \frac{len(HistoricalData)}{blockDim} \right\rceil \tag{3}$$

where *gridDim* is the number of blocks and *blockDim* is the number of threads per block. Moreover CUDA defines for each generated thread an id, that is used by the thread to access to the right chunks of data with the following rules:

$$index = threadId + blockId * blockDim \tag{4}$$

where *index* will be the position of the Historical Data vector where the thread has to operate, *threadId* is the thread id and *blockId* is the id of the block.

In the following descriptions the last phase, i.e. the minimum SAD computation, will be the same in all the four implementations, in order to not influence the comparisons between the several methods.

A. Naive Implementation

In the first implementation, that we call Naive, every threads read both Historical Data and Queries and they write the resulting SAD from the Global memory. That will be expensive because reading/writing from the Global memory is the slowest procedure for the GPU kernels. Since each thread reads a chunks of Historical Data, we define two nested loop, the outer one on the Query length and the inner on the Queries number. This procedure is the opposite to the OpenMP one presented in Section IV, where the loop on the number of Queries is the external one. That is done because we want to limit the number of Global memory access, in order to reduce the reading/writing slow procedures. Thus we define a private variable for each thread that stores a value from Historical Data, based on the thread id and the external loop value. So the thread can compute the SAD of this stored value with all the Queries that have to be checked. More in details, suppose that we are in position i of a Query q. So each thread reads from Global memory the value $HistoricalData[index + i]$, storing it in its private variable. After that, it can compute matches between this value with respect to all the Queries thanks to the inner loop. The current SAD value will be:

$$R[index + q * len(R)] += |tData - Q[i + q * len(Query)]| \tag{5}$$

where *tData* is the thread private value of Historical Data, and $q = 0, 1, \ldots, numQueries$ identifies the right position which the thread uses to read a Query value and the result vector index where the thread writes the SAD value. When each thread consumes its data chunk, there is a barrier (*__syncthreads*) that makes threads in a block wait until the end of the other threads jobs. Finally there is the minimum computation on the result vectors that is a bottleneck of our implementation, since only one thread per Query can compute it, avoiding possible race conditions.

B. Private Implementation

Since writing each time on the result vector can be expensive, we propose another solution that is similar to the OpenMP Privatization method. That is composed by two loops: the external loop on the Query and the inner one on the Query length. Furthermore we define for each thread a temporal variable to store the SAD value. Only when the thread finishes the SAD computation on a Query q, it will write the final value in $result[index + q * len(Result)]$. After that, as we said before, just one thread searches for the minimum SAD value and its id in each Query.

C. Tiling Implementation

Here we describe the Tiling implementation, that is a Pattern Recognition solution that exploit the GPU Shared memory. This kind of memory is shared for each block and reading/writing in/from it is faster than to the Global memory but it is smaller. Tiling is the process to load in the Shared memory the data from the Global one, allowing every threads in a block to analyze every Historical Data and Queries that are stored in the block Shared memory. In essence each thread of a block loads in Shared memory one value from Historical Data and from a Query. If the Query length is bigger than the tiling size (the portion of the Shared memory reserved for this data), the temporary SAD values of the result vector will be computed between the chunk of the Historical Data and the Query one. After that, the threads load the remaining part of the Query overwriting the old values, without changing the Historical Data. If we look again equation 2, we can observe that the difference from the index of *D* and *q* defines the index where the result has to be saved in *R*. Obviously this difference has to be smaller than the Result length and bigger than 0. Basing on that, given a Historical Data chunk we can compute values for several index of the result vector *R*. In particular, we define the following rule:

$$R[idx - r + p \cdot T + q \cdot len(R)] += |Ds[threadId] - Qs[r]| \tag{6}$$

where T is the dimension of the Shared memory portion for the Historical Data, *idx* is the position in result vector where the thread can write, (i.e. its index defined in 4). The value p specifies the current portion of Query that is loaded in Shared

memory, q is the number of the current Query and r is used to write in all result vector positions based on tiling dimension, where $r = 0, 1, \ldots, T$. In addiction to these, as usual in this paper, we refer to R as the result vector, while Ds and Qs are the Shared memory Historical Data and Queries chunks respectively. Since different threads from different blocks can write in the same memory location of R, the sum in equation 6 was replaced with the CUDA *atomicAdd*. This function avoids race conditions, because just one thread can update values stored in R at once. At last we extract the minimum SAD value and its id, with the same implementation presented above.

D. Constant Implementation

This implementation exploits both the GPU Constant and Shared memory. The Constant memory is only-read cache from Global device memory. This can boost the performance of our implementation since thanks to it we can leverage some caching improvements. Thus we load in the Constant memory all the Queries of the Pattern Recognition task, instead of the previous method where the Queries require different uploads in the Shared memory for the same Historical Data chunk. In this way each thread reads sequential memory locations in the Constant memory. This solution simplifies 6 as follows:

$$R[idx - r + q \cdot len(R)] += |Ds[threadId] - Qc[r + q \cdot len(Q)]| \tag{7}$$

where Qc corresponds to the vector of all the Queries loaded in the Constant memory, and *len(Q)* to the length of each Query. As before, the sum needs to be atomic since threads of different blocks can update the same index of R. The last phase of this method is the minimum SAD search, like in the other scenarios. Sometimes there is a drawback in using the Constant memory: the limited memory occupation. As consequence of this we ca not allocate a lot of Query in it.

VI. Experimental Results

In this section we discuss about the experiments that we bring during this project. In all the experiments we suppose that there are multiple fixed size Queries which have to be checked on a Historical Data vector. Anyway, we believe that the generalizations to more than one Historical Data vector and to Queries with several length will be easy to introduce. In addiction to these we talk about the evaluation metrics and our setup.

A. Experiments Configuration

We conducted our sequential and OpenMP experiments on a Intel i7-8750H CPU which has 6+6 (physical and virtual) cores. CUDA ones are performed on a NVIDIA GeForce GTX 1050 Ti.

To validate our results, we also replicate each one trial some times, and we report always values those are means with their corresponding standard deviations (std). In addiction to that we also seed all our random generators (both CPU and GPU ones) to yield reproducible experiments.

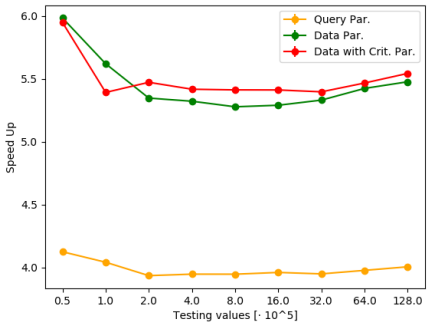

Fig. 2: Resulting Speedup for the OpenMP implementations on the Historical Data size test.

B. Performances Evaluation Metrics

To evaluate properly an experiment, we report two different metrics, i.e. Speedup and Efficiency. The former is computed as follows:

$$S_p = t_s / t_p \tag{8}$$

where t_s is the sequential time, t_p is the parallel time with p processors and S_p is the resulting Speedup. The Speedup can be super-linear ($S_p > p$), linear ($S_p = p$), sub-linear ($S_p < p$) or without Speedup ($S_p < 1$). Our results often report values close to the linear one, but in certain scenarios a super-liner Speedup. Sometimes the Speedup can be evaluated like Relative Speedup, because t_s can leverage some parallel implementations in background. For this reason, 8 can be defined in such way:

$$RS_p = t_1 / t_p. \tag{9}$$

In the OpenMP we consider t_1 as the parallel algorithm executed with a single thread. In the CUDA ones we consider t_1 as the Naive implementation. In this way we can better analyze the relative boost in performance that the other techniques bring.

The Efficiency of an algorithm using p processors is:

$$E_p = S_p / p. \tag{10}$$

This is a value in $[0, 1]$ and it measures how well-utilized the processors are in solving the Pattern Recognition task. Since the Pattern Recognition task includes the minimum search that is hardly-parallelizable, the efficiency E_p decreases as p increases.

C. Experiments

The Pattern Recognition task can be studied on several points of view: different Historical Data length, different length and number of Queries. In addiction, we exploit different threads configurations to evaluate our parallel implementations performances. To validate our experiments, we define a set of default values for each hyper-parameter. In such way, changing one of them, we can study the obtained results for each experiments. The default values are: 100000

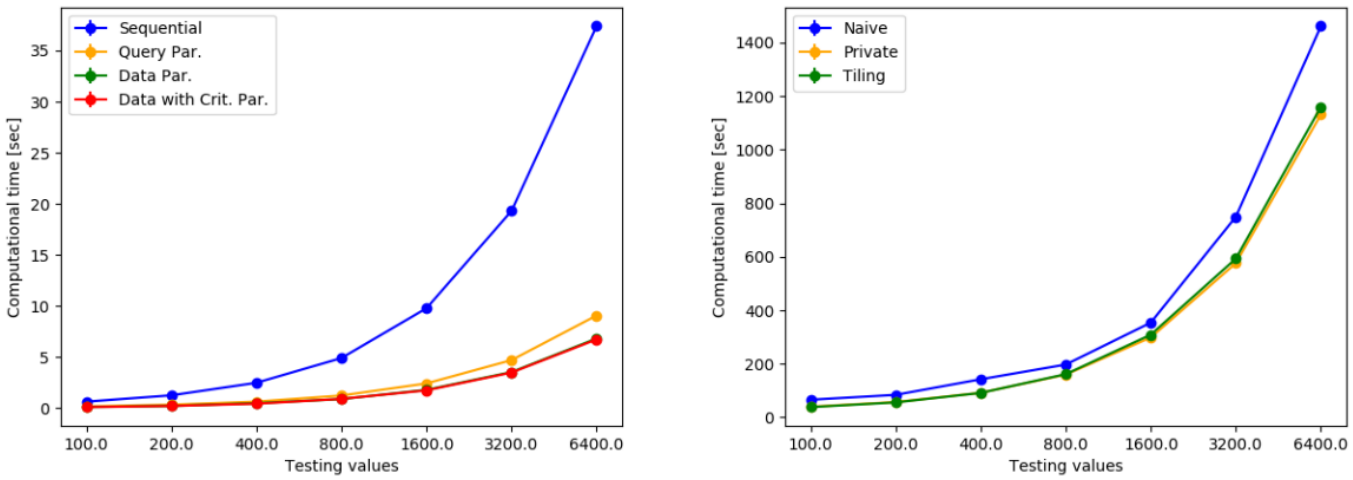

Fig. 3: On the left is reported the computational time on the Query length tests from the OpenMP implementations and on the right is reported the computational time on the CUDA proposed solutions.

for Historical Data vector length, 1000 for Query length, 10 total Queries, 12 threads number in the CPU implementation and 128 threads for each GPU block. Furthermore, we always perform 10 runs with the same configuration for computing mean and std of its computational time, extracting more stable results and reporting values with small std.

In Figure 2 we report the Speedup for the OpenMP tests where we increase the Historical Data size. As we can observe, as the Historical Data grows both the Privatization and the Lock implementations (Green and Red plot respectively) report similar Speedup values. In particular, the Lock implementation has higher performances because, in this scenario, there is not the minimum SAD search. The Query solution values (Orange plot) are slightly lower compared to the others. This because the Query implementation adopts a sequential SAD computation for each Query. In Figure 2 we do not report the CUDA implementations test values since there are almost incomparable. To show these huge differences, we report the computational time of the three implementations in Table I. In particular, in addiction to the sequential method, we insert the best two implementations for both OpenMP and CUDA methods for this kind of test: Privatization for OpenMP and Constant for CUDA. Another important aspect that underlines the gap between CUDA and OpenMP is the reported unit of measurement. The CUDA implementation is evaluated in micro seconds (μs) while the Sequential and OpenMP ones in seconds.

TABLE I: Test on the Historical Data length. From the first column are reported all the different vector size and the computational time means with std of the Sequential, OpenMP (Privatization) and CUDA (Constant) implementations.

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	$780 \pm 6.58\%$	$142 \pm 2.57\%$	$181.3 \pm 3.72\%$

Then we study several size for Queries length from 100 to 6400, the relative tests value are reported in the left figure in Figure 3. In this scenario OpenMP Query implementation (Orange plot) reports the same performances of the Privatization and Lock ones (Green and Red plot respectively) since the query is almost 800 length. When this size grows, the data parallelism methods achieve smaller times with respect to the Query methods. Otherwise in the right figure in Figure 3 we report the obtained computational time for CUDA implementations. We can notice that the Naive method gets higher results, while the Constant solution is the best one. Figure 3 shows how the query length negatively affects the computational times, since the parallelism on each query is not a nice solution for the Pattern Recognition task. This because each thread has to compute SAD values with respect to the entire Historical Data.

The OpenMP and CUDA experiments with different Queries number report the same behaviour as the ones that are analyzed for the tests with different Query length (described in Figure 3). In this testing configuration, we can not test the CUDA Constant implementation due to the smaller dimension of the device Constant memory: it is impossible to allocate more than 10 Query each composed by 1000 floating numbers in $[0, 1]$. This fact can be the only flaw of this implementation. However, in this kind of experiment, the other three CUDA algorithms reported almost the same performances.

At the end, we perform some tests on the threads number both for the OpenMP and CUDA solutions, that are collected in Figure 4. On the left figure, we represent the Speedup for all the OpenMP implementations. Here when the threads number are set to a value lower than 6, we observe that all the solutions achieve a super-linear Speedup. Increasing the threads number, the results do not report the same linearity as the others. With threads fixed to 6 there is the maximum. In the CUDA

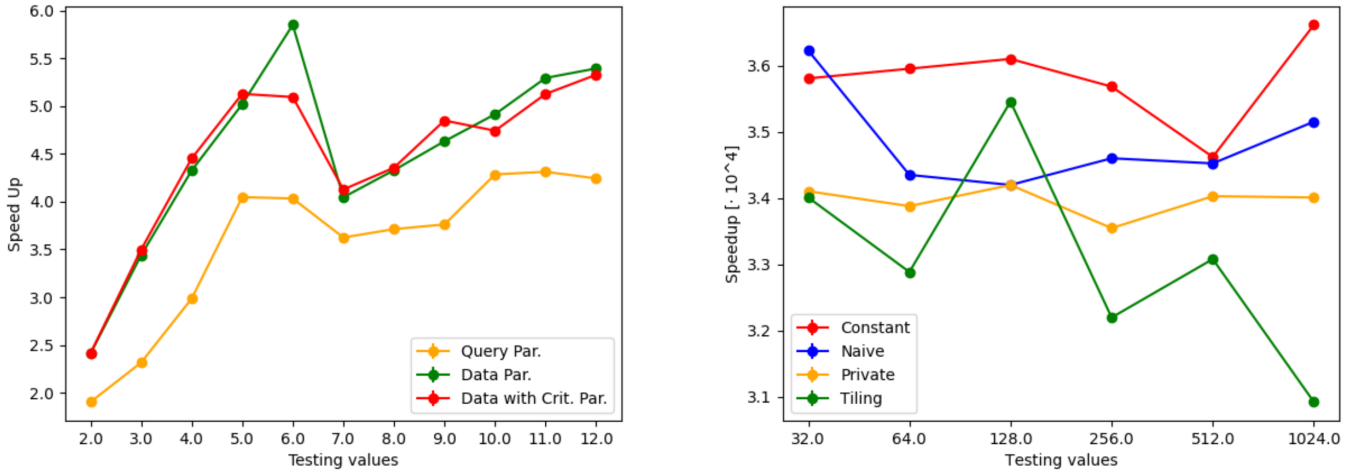

Fig. 4: On the left is reported the obtained Speedup of the OpenMP methods for several threads number and on the right is reported the CUDA algorithms Speedup ($\cdot 10^4$) with different device blocks size.

algorithms (the right figure), instead, we change the number of threads for each device block. The best performances in most of our implementations are reached with 128 block dimension, that is the reason we choose this value as the default one. From these experiments, it is easy to evaluate also the Efficiency and the Relative Speedup of our solutions. We summarise all the performance metrics in Table II for all our parallel solutions with the same parameters values. The Speedup is obtained with respect to the Sequential approach. The Relative Speedup, instead, is computed with respect to the baseline methods both for the CPU and the GPU implementations: the Sequential and the Naive solutions respectively. In this way we can observe the real improvements that complex methods can bring. The Efficiency is computed according to 10, with respect to the used processors, that is 6 for the CPU implementations and 100096 for the GPU ones (this follows 3). From Table II we can observe the heavy gap between the GPU and the CPU performances, i.e. almost 10^4 grater than the CPU algorithms. This demonstrate the GPU power and underline all possible benefits that those bring. In contrast to this the Efficiency shows that the CUDA algorithms report low performances. The reason is that, at the beginning of each test, we load on the device memory the queries. This underlines that the bottleneck of GPU implementations is the data transfer from host to device. The OpenMP implementations, instead, obtains satisfactory performances results, since their Efficiency is close to 1 and the Speedup is almost linear. The Privatization method has a maximum in the Speedup close to 6 with 6 threads. Here it achieves a 0.97 of Efficiency, since in this solution the minimum search is performed in result vectors with the same size to the threads number. This shows also how the sequential computations in a parallel algorithm break down the performances, and it underlines the importance of avoid them as much as possible. The minimum search in the Privatization solution, indeed, is on vectors with same size of the threads number, and here we do not adopt any Critical Section.

VII. Conclusions

In this paper we propose several implementations for achieve the Pattern Recognition task that leverage parallel computations due to its definition (see 2). In particular, we explore the OpenMP and CUDA features to face with this kind of task. Our aim was to demonstrate how the GPU implementations can boost performances with respect to the Sequential and CPU parallel ones.

We notice how Pattern Recognition can be analyzed on several points of view, thanks to its different settings. We have found interesting results when we analyzed different threads number on the OpenMP implementations, reaching an almost linear Speedup and a high processors Efficiency. However experimental results underline the huge gap in performances between this two approaches. CUDA values are almost incomparable to the OpenMP ones. Thanks to the Efficiency we can also deduce how data transfer from CPU to GPU is the bottleneck of CUDA solutions.

TABLE II: 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.

	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	3.89	5.36	4.47	(-)	1.24	1.34	1.51
E	0.67	0.97	0.85	0.22	0.28	0.30	0.34