UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Parallel computing

Time Series Pattern Recognition

Edoardo Sarri

7173337

February 2026

Contents

1 Introduction

This project is about the C++ implementation of a program that, given a pattern (i.e., a query), found the it closest sequential in a time series input. There are two implementation: the first is a sequential approach, described in Chapter 2; the second use the CUDA framework, scribed in Chapter 3, to obtain the maximum speed-up.

This is the repository of the project.

1.1 Approach

To find the best match the approach used is the sliding window: for each instance time we compute a distance between the query and the input. Afeter this we return the best part of the input that is closest to the query. If the Q is the lenght of the query and I is the lenght of the input, we have $I \gg Q$).

The distance from the query and the input window is the the Sum of Absolute Distance (SAD). If we use it within univariate data, if are at i-th instant time, and the query has a lenght of M, we compute the distance as $D(i) = \sum_{m=0}^{M-1} |Q[m] - T[i+m]|$, where the objective is to find the index i_{min} such that $D(i_{min})$ is minimized. Since we are dealing with multivariate data with D dimensions, we extend the metric to sum differences across all dimensions: we compute $D(i) = \sum_{m=0}^{M-1} \sum_{k=0}^{D-1} |Q[m,k] - T[i+m,k]|$.

1.2 Dataset

We selected the Human Activity Recognition (HAR) dataset. This is a multivariate time-series dataset with $D = 6$ dimensions per timestamp: 3-axial linear acceleration (*acc_X*, *acc_X*, *acc_X*) and 3-axial angular velocity (*gyro_X*, *gyro_Y*, *gyro_Z*).

The dataset consists of 10299 samples (7352 for training and 2947 for testing). Each sample captures 2.56 seconds of activity recorded at 50Hz, resulting in a window length of 128 timestamps. In this scenario we concatenate the dataset samples to generate a continuous stream of multi-dimensional data points, resulting in a total of circa 7,9 million floating-point values to process.

The result of this idea is that the dataset is downloaded in `data/input.txt` file, where each samples are concatenate and each element of one sample is rappresented by a line with 6 float value (the dimension of HAR), that are respectively *acc_X*, *acc_X*, *acc_X*, *gyro_X*, *gyro_Y*, *gyro_Z*.

1.2.1 Multiplier

To address the limited size of the original dataset the system includes a data augmentation mechanism controlled by a *multiplier M*.

The mechanism is implemented as a pre-processing step during data download: the system iterates through the complete set of original samples `M` times; the first iteration preserves the original raw values; for every subsequent iteration required to reach the target multiplication factor, the system generates a new variation of the entire dataset, by injecting uniform random noise ϵ (by default $\epsilon \pm 0.01$) into the signal values.

1.3 Input and query

To define the input and the query we have to address two aspects:

- Input
  We concatenate all samples of the dataset, both the training and test partiotion. When the download script has finished the *input.txt* file has the following format: each value of the same timestamp is separated from a single space; to time stamp are separated from the $\backslash n$ character.

  We perform a data-augmentation process to increment the number of sample because our dataset was too small. We can define the number of copy (1 by default) and the noise added (± 0.01 by default). The new sample is define as $new = old + \epsilon$, where $\epsilon \in [-0.01, 0.01]$.

- Query
  We sample a casual index of the input and take a query of Lenght M. The default choice for the lenght of the action is $M = 64$, but it's also possible to specifie it at compilation time; by deafult we are evaluating 1,28 second of data and search the closest part of action in the dataset. We add to this query a Gaussian noise to ensure that $SAD > 0$. This perturbation is obtain from a Normal Distribution with $\mu = 0$ and $\sigma = 0.01$; this garantee the robustness of our approach.

1.4 Tools

To validate and test our software we use two main tools: the sanitizers and the profiler.

1.4.1 Sanitizers

A sanitizer is a dynamic code analyzer integrated in the compiler that allow us to discover runtime errors in out code. The tool instruments the code to store metadata for each variable and this allow to detect runtime errors that the compiler can't see because it works in a static way.

The sanitizer is useful during the development phase. In the release version we compile without the sanitizer because it introduces complexity and brings to a performance degradation.

The sanitizers tool that we used are the following:

- Address (ASan)
  Is useful for the detection of memory corruption errors like buffer overflow, use after free and memory leak.
- Undefined (UBSan)
  Is useful for the detection of undefined behavior in the code, i.e., portion of code that isn't conformant with the standard.
- Memory (MSan)
  Detects the use of uninitialized memory. In MacOS it isn't supported, but it is present anyway.

1.4.2 Profiler

Profiling is a technique for analysing the performance of our software: it is useful to understand where the code spends most of its time and how many functions are called.

There are two techniques that profilers use: sampling, that is more efficient but less precise; instrumentation, that is very accurate but more expensive.

In our project we used the GPerfTools. We install it with HomeBrew, but this doesn't install the *pprof* tool for the analysis. The new version was found here, and it is developed by Google in Go.

1.5 Execution

The execution of the project requires two steps:

1. To download the input we have to exec the `exec/download_input.sh` script. This Python script orchestrates the dataset preparation:
   - Downloads the original UCI HAR Dataset (ZIP format) from the official repository.
   - Extracts the raw inertial signals from the archive.
   - Processes the data concatenating the training and test sets.
   - Applies the data augmentation, as we said in Section 1.2.1, to generate the final `data/input.txt` file used by the C++ application.

2. Finally we can execute one the script in `exec/` folder. Which script depends on our interest: see README for more details.

2 Sequential

In this Chapter we define the structure and the implementation detail of the sequential version of our code. This baseline is usefull to study the performance differences with the parallel version described in Chapter 3; this analaysis will be in chapter 4.

Thi version of code is in sequential folder within the main repository.

2.1 Pipeline

Supposing that the file already is in `data/input.txt`, out sequential version follow the following steps:

1. Data Loading
   We allocate memory and load in it the input file. We reorganized the data into a Array of Structures (AoS) memory pattern. Implementation is in data_loader.cpp file.

2. Query Generation
   We extract a subsequence of lenght M as the query. As described in Section 1.3, we add to this subsequence a Gaussin noice ($\mu = 0, \sigma = 0.01$) to avoid that the SAD distance is $SAD = 0$, i.e. that the query is not exactly the same of our input subsequence. In the code there is a seed to allow the reproducibility of the code; we use the seed 78 for our experiments. The implementation is in query_generator.cpp file. This step return also the start index used for extraction.

3. Pattern Matching
   We use a sliding window to scan the input, calculating a SAD of lenght M for each istant time. The implementation is in SAD_distance.cpp file.

4. Reporting
   We return the index of the best match and the relative SAD value. For verification purposes, we also display the ground truth index (the start index of the query).

2.2 Data Loading

Given the large size of the input dataset ($\approx$ 3.5 GB in our experiments), standard C++ streams (`std::ifstream`) introduce significant overhead due to the double coping (in the kernel memory and after in the user's buffer) and repeated system calls. To address this bottleneck, we adopt a Memory Mapped I/O (`mmap`) strategy: the file is mapped directly into the process virtual address space, allowing the application to access the

file content as a contiguous byte array.

In this way we increase the efficincy, but we lost the semplicity of stream parsing and we have to do all stuff by hand. To gain the optimization we must return to the pointer apporch of C.

Additionally, since the parsing logic accesses memory strictly linearly, we utilize the `madvise` system call with the `MADV_SEQUENTIAL` flag. In this way we instructs the kernel to optimized memory usage and prefetch the subsequent pages.

2.3 Memory Layout

For the sequential implementation, we adopt the Array of Structures (AoS) layout. In this configuration, all D dimensions of a single timestamp are stored contiguously in memory: the dataset is represented as a flat vector like $[x[1,1], \cdots, x[1,D], x[2,1], \cdots, x[2,D], \cdots, x[N,1], \cdots, x[N,D]]$. If we want access to the d-th dimension of t-th timestamp element we can use $f(t,d) = buffer[t \cdot D + d]$.

Since we adopt a sliding window approach, this memory patter technique allows us to maximize the spatial location during the distance calculation. The algorithm, described in Section 2.4, infact have a inner loop through the D dimension, and a outside loop through the input lenght N; when we load in cache the d-th dimension of the t-th timestamp, we laod also the near values in the same cache line, and these values are processed in the next iterations.

2.3.1 Vectorization

This structure of the data allows us to implement an auto-vectorization without writing assembly code. With the correct compiling flags (i.e., `-march=native` and `-O3`), the compiler generates specific instructions for the host CPU.

However we have to give a hint to the compiler that we are sure that vectorializzation is possible. In particoular we have to declar `restrict` clause in SAD_distance.cpp file to unlock full SIMD operations: in this way we inform the compiler that the `data` and `query` arrays never overlap.

2.3.2 Padding

Our data are a collection of float numbers and for each timestamp we have $D = 6$ dimension; we can see the data like a $N \times D$ matrix, where N is the number of elements. The cache lines are composed tipically by 64 Byte and the CPU works better if the data are alligned with this value. Since a float is load in 4 Byte, 6 floats occupies 24 Byte; we can optimize the performance introducing two more dimension (i.e., two more floats) in the way one time stamp occupies 32 Byte in the cache line.

This padding approach doesn't change the SAD distance result, but allow us to have

low cache miss and also optimize the vetorization described above. On the other hand we have a waste of memory of 25% due to padding. We implemented an automatic padding mechanism that adjusts the inner dimension of the dataset to be a divisor of the cache line size (typically 64 Bytes). This alignment ensures that data loads are efficient and fully utilize the memory bandwidth.

The above concept is correct in teory, but the real test say us a differnt thinks. The Section 4.1.3 show these results.

2.4 SAD distance

The SAD metric calculation is performed using a Brute Force Sliding Window approach. If Q be the query of length M, we search for the starting index i^* that minimizes the distance: $i^* = \arg\min_{i \in [0, N-M]} D(i)$, where = $D(i) = \sum_{m=0}^{M} \sum_{d=0}^{D} |Q[m,d] - T[i + m, d]|$.

2.4.1 Early Abandoning

To optimize the sequential execution, we usually implement an early abandoning approach: during the computation of $D(i)$, we maintain the current distance; if the partial local sum exceeds the current global minimum distance found (*min_dist*), the computation for the current window i is immediately halted.

However, this technique creates a significant algorithmic disparity when comparing with a parallel GPU implementation, but instead we want compare "apple with apple". On the GPU, thread divergence makes early abandoning less effective or counter-productive for this specific kernel structure. So we have parametrized it with the `ENABLE_EARLY_ABANDONING` compilation flag, that is `false` by default.

To see this analysis go to Section 4.1.1.

3 Parallel

In this Chapter we analyse the choice taken for the parallel version of the program. The main goal of this version in maximise the throughtput, i.e., obtain the maximum speed-up related to the sequential version, described in Chapter 2.

3.1 Memory Layout

The GPU are thinked to work with SIMT (Single Instruction, Multiple Threads) execution pattern, and this implies that the Array of Structures (AoS) doesn't match this pattern. We have to define a new layoyt of data, and the Structures of Array (SoA) match perfectly the GPU coaleshing memory access.

3.1.1 SoA memory pattern

The main problem of the AoS in the GPU is given by the warp (i.e., group of 32 thread) parallel execution: they execute the same istruction over different data. In particular, let T be the linear array of data that follow the SoA memory pattern, the first thread requires in order $t_0d_0, \cdots, t_0d_5$, the second requires $t_1d_0, \cdots, t_1d_5$, and so on. If the dimension of these data are contiguous in memory, when the first thread bring the its first dimension data in memory, it bring also the the first dimension of the other threads; when these other threads searchs for their first dimension data in memory, the data will be already present.

3.1.2 Inline mapper equation

So the data are stored in a Structures of Array patter, so in a linear long array T where the data related to each dimension are contigous. To define the index in this array of the d-th dimension of the t-th element we can use $Index(t, d) = d \cdot N + t$, where N is the dimension of the datataset, i.e. the number of total timestamps.

3.2 Memory Hierarchy

In GPU programming is important define de memory hierarchy. The objective is minimize the data trasfer from the main global memory, which is the slower, to the thread register. We define the hierarchy as follows:

- Global Memory
  Stores the full input dataset T in SoA format.

- Constant Memory
  Stores the Query Q. To ensure consistency and maximize cache efficiency, Q is also stored in **SoA layout**. Since the constant memory is small (i.e., usually 64KB) the query can't be to much longht: in out case, consider the default lenght of 64 described in Section 1.3, we have circa 1.5 KB, so it fits easily within the 64KB constant memory limit. This garantee a huge performance benefit: we haven't to tranport the query from the global memory to the local one for each thread; all threads in a warp use the same query, so we can do a single memory transaction to have this data in the local memory.

- Shared Memory
  This memory is much faster than the global memory and allows threads to co-operate. We use it to perform a block-wise reduction and to implment a tiling strategy:

  - Tiling: Stores a "tile" of the input data corresponding to the block's sliding window range. This allows threads to read input data from fast on-chip memory instead of Global Memory, reducing global memory accesses by a factor of circa $\times query\_lenght$.
  - Reduction: Stores the partial SAD results to perform the block-wise parallel reduction efficiently.

  The total shared memory usage per block is approximately 3.25 KB (Reduction: $256 \times 8B$ + Tiling: $319 \times 4B$), which is well below the hardware limit (typically 48 KB per SM), ensuring high occupancy.

- Registers
  Used for local accumulation of the SAD distance within each thread.

3.3 Pipeline

The execution flow is designed to orchestrate the Host-Device collaboration, minimizing control divergence and maximizing memory throughput. The pipeline consists of the following strictly ordered stages:

1. Data loading (Host)
   The Host loads the input data using the efficient Memory Mapped I/O (`mmap`) strategy, identical to the sequential version described in Section 2.2. Since the raw file is in AoS format, the Host iterates through the mapped memory and transposes the data into the SoA layout. In this case we also have to solve the paging potential problem: we have to prevent that the host OS shfit the data to the disk during the traner from the CPU memory to the GPU memory. This is addressable tanks to the pinned-memory: we have to say to the host OS that that

memory cann't be pageable.

2. Index definition (Device)
   The first step for each thread is to define the index of the timestamp on which it will work. Since we may initialize more threads than the number of timestamps, each thread must verify if its index is out of boundaries, i.e. if $tid \geq (N - M)$, where N is the length of the input and M of the query.

3. Local accumulation (Device)
   Each thread accumulates in a local register the Sum of Absolute Differences of its related timestamp; this register resides in the fastest memory available (i.e., register file). As we said in Section 3.2, the query is retrived from the constant memory and, to minimize global memory latency, the data from the shared memory using a tiling strategy. For the tiling all threads in a block cooperate to load the required input segment from Global Memory into Shared Memory. The tile has $block\_size + query\_length - 1$ size. The distance is computed as $D(tid) = \sum_{d=0}^{D-1} \sum_{m=0}^{M-1} |Q[m,d] - T_{shared}[tid_{local} + m]|$.

4. Block Reduction (Device)
   This is the critical optimization step: writing every thread's result to Global Memory isn't efficint since the global memory is the slower memory and implies the transfer of all data from the global memory to the host memory, and we3 know that the PCI bus is the main bottleneck. To avoid this problems we perform a block-grain reduction: we define the best index i_j^* for each block j usgin the shared memory and the reduction pattern; in this way we have to bring in host memory only *grid_size* value, reducing the data transfer fo a *block_dim* factor. With the reduction we have a $\mathcal{O}(\log(block\_size))$ cost instead a linear cost is we performe the reduction sequentially in cpu. In this scenario i critical the coice of the *block_size* value.

5. Result write-back (Device)
   After the reduction, only the first thread of the block holds the block's minimum. This thread writes two values to Global Memory: the minimum SAD and its corresponding timestamp index.

6. Final reduction (Host)
   After the kernel execution, the Host retrieves the reduced output array from Global Memory. It then performs a linear scan over the block results to identify the global minimum distance and the corresponding starting index i^*, returning the best match found.

3.4 Grid

We implement a kernel where the thread i-th computes the SAD distance for a single sliding window starting at index i, so for the i-th timestamp.

The grid that mapped each thread in its timestamp is a 1D grid with 1D blocks. The size of the grid depends on the number of timestamps N and the number of thread *block_dim* for each block. We have to ensure that there are at leans one thread for each timestamp, so we can define the dimension of the grid, i.e. the number of block, as $dimGrid.x = \left\lceil \frac{N}{block\_dim} \right\rceil$.

3.4.1 Block Dimension

The critical choice is the dimension *block_dim* of each block. In our implementation we set by default $block\_dim = 256$, but this value can be set at compilation time. This choice is driven by:

- Hardware
  We can define at maximum 1024 thread per block.
- Warp
  Is a good choice set the block dimension as a multiple of 32, i.e. the number of thread for warp, the smallest chunk of thread that is allocated on cores.
- Reduction efficiency
  The kernel outputs is composed by $N/block\_dim$ partial results: i lager *block_dim* drastically reduces the volume of global memory writes and PCIe traffic (e.g., $block\_dim = 256$ reduces output size by 99.6% $(1 - 1/block\_dim)$).
- Synchronization
  Larger blocks (e.g., 1024) increase the synchronization latency at the barrier, as the entire block must wait for the slowest warp.
- Latency hiding
  If *block_size* is to to much big, we are limiting the scheduler's ability to switch contexts and hide memory latency because we can have to few block within a SM (GPU Streaming Multiprocessors (SMs) typically support up to 2048 resident threads).
- Shared memory hw limit
  The Shared Memory is logically shared among the thread within the block, but fisically is shared among all thread in a SM. Its usage per block is proportional to *block_size* ($2 \cdot block\_size \cdot$ sizeof(float)).

3.4.2 Index Mapping

The global thread index *tid* maps directly to the time-series index *i*. The thread equation, described theorically in Section 3.1.2, began pratically $tid = blockIdx.x \times blockDim.x + threadIdx.x$, where: *blockIdx.x* defines the number of blocks previus the current thread block; *blockDim.x* describes the dimension of the single block; *threadIdx.x* defines the local index of the thread in the block.

4 Analysis

In this Chapter we analyse the sequential code and the parallel one.

The experiments were conducted using these settings:

- The dataset was downloaded with a multiplier of 50. This scaling factor, that is described in Section 1.2.1, was determinated empirically and ensure that the execution time of the searching algorithm doeasn't be to much rapid. It implies that the *input.txt* file has circa 3.5 GB size.
- The query lenght default is 64 timestamps.

The server specific are the following:

- Architecture: x86-64
- Operating System: Ubuntu 22.04.4 LTS
- Kernel: Linux 6.8.0-52-generic

4.1 Sequential

In this Section we analysis the result of the sequential versione described in Chapter 2.

In the following section we analyse the detail, but in general we can say that the total execution time of the program is circa 38.8 for the default 64 query lenght, and 69.4 seconds if we use a 128 query lenght.

4.1.1 Early Abandoning

We tested the sequential performance with and without the Early Abandoning optimization. How we said in Section 2.4.1, the correct baseline is with this tecnique disable.

With ealing abandoning anable we obtain a circa 2.87$\times$ speedup: the matching time, i.e. the execution time for seearching in the input file, with it enables is about 8.84 seconds, and without is circa 25.35 second.

4.1.2 Profiling

The profiling analysis was performed using `gperftools`, as described in Section 1.4.2, on the sequential implementation. This analysis reveals two distinct phases and the I/O buond:

- Pattern matching: This dominates the total execution time with circa 65%.

- Data loading: The data loading execution time is circa 30%.

4.1.3 Padding

As we said in Section 2.3.2, we can introduce a padding to the data to alligned them with the cache line size. The experiments with and without padding show us that, in this case, introducing padding doesn't improve the efficiency of out matching algorithm: its execution time with padding is circa 35.8 seconds, but withou padding is circa 26.4 seconds.

The motivation of this could be, in this case, the extra effort that the CPU must do to process the trash value of padding, since they are the 33% of the real data: if we work with more and more lenght of the data, probabily the padding increase the speed of our code; in this case the cpu prefers handles it self the load in memory of the unpadded data.

4.1.4 Query lenght

If we use a longest query we will obtain a greatest execution time of the matching algorithm. We have perform three experiments:

- **32 timestamps:** The exectuion time is circa 10.4 seconds.
- **64 timestamps (deafult):** The exectuion time is circa 26.2 seconds.
- **128 timetamps:** The exectuion time is circa 56.3 seconds.

4.1.5 Data Loading

The first approch for the data laoding step used the classic C++ stream. The time required for the parsing of the input text file wax approximatly 47 seconds. Whit the new memory mapped I/O approach the time is now circa 13.32 seconds.

4.2 Parallel

In this Section we study the speed-up obtained execution out CUDA code and.

The first different between this and the sequential execution can be see analysing the total execution time of code. With the default 64 timestamps query lenght we have circa 14.114 seconds of execution time, and with 128 query length it is circa 14.931 seconds.

4.2.1 Loading Time

The bottleneck of our code is the laod time, i.e. the time that the CPU takes to bring the data from the disk to the memory. Let's take one experiments with our 3.5 GB

input file: the total execution time is 14.473834 seconds, but the data loading time is 14.349911 seconds; this means that the other steps time is began irrilevant.

4.2.2 Query Lenght

If we see the result in Section 4.1.4, changing the query lenght have. a huge impact on the matching algorithm execution time.

In the parallel version changing the query lenght practically doesn't change the total execution time: the data loading time, that is our bottleneck, remains the same; with a 64 query length we have a matching time of 0.053448 seconds, with a 5125 query lenght we have matching time of 0.328047 seconds and with a huge 2048 query lenght we have 1.215921 seconds.

The only problem with a very huge query lenght can be its sotre in constant memory: indeed with a 4096 query lenght the program throws an *ptxas error* with message *File uses too much global constant data*.

4.2.3 Block Size

By defualt we have a 256 block size value. Let's see if this is a good choice or there are other good value:

- 8: The kernel execution time is 0.190880 seconds.
- 32: The kernel execution time is 0.061503 seconds.
- 128: The kernel execution time is 0.056112 seconds.
- 256 (baseline): The kernel execution time is 0.055100 seconds.
- 1024 (max): The kernel execution time is 0.067674 seconds.

Instead, with a block size of 1024, we have a huge barrier synchronization cost since also only one thread have a longer exetution time all other 31 warp have to wait the warp of that thread.