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), finds the closest subsequence in a time series input. There are two implementations: the first is a sequential approach, described in Chapter 2; the second uses the CUDA framework, described 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 time instant we compute a distance between the query and the input. After this we return the best part of the input that is closest to the query. If Q is the length of the query and I is the length of the input, we have $I \gg Q$).

The distance between the query and the input window is the Sum of Absolute Differences (SAD). Let 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 $D(i) = \sum_{m=0}^{M} \sum_{d=0}^{D} |Q[m,d] - T[i + m, d]|$.

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_Y, acc_Z) 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 is that the dataset is downloaded to the `data/input.txt` file, where all samples are concatenated. Each element of a sample is represented by a line with 6 float values (the dimensions of HAR), which are respectively acc_X, acc_Y, acc_Z, $gyro\_X$, $gyro\_Y$, $gyro\_Z$.

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 partitions. When the download script has finished the *input.txt* file has the following format: each value of the same timestamp is separated by a single space; two timestamps are separated by the $\backslash n$ character.

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

- Query
  We sample a random index of the input and extract a query of length M. The default choice for the length of the action is $M = 64$, but it's also possible to specify it at compilation time. The data preparation script extracts this query, adds Gaussian noise ($\mu = 0, \sigma = 0.01$) to ensure $SAD > 0$, and saves it to `data/query.txt`. The original start index is reported by the script for verification.

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 allows us to discover runtime errors in our code. The tool instruments the code to store metadata for each variable and this allows detecting 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 leads to performance degradation.

The sanitizer tools that we used are the following:

- Address (ASan)
  It is useful for the detection of memory corruption errors like buffer overflow, use-after-free and memory leaks.
- Undefined (UBSan)
  It is useful for the detection of undefined behavior in the code, i.e., portions of code that are not conformant to the standard.
- Memory (MSan)
  Detects the use of uninitialized memory. On macOS it is not supported, but it is present anyway.

1.4.2 Profiler

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

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

In our project we used GPerfTools. We install it with HomeBrew, but this doesn't install the *pprof* tool for the analysis. The new version, found here, 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 execute 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 by concatenating the training and test sets.
   - Applies data augmentation, as described in Section 1.2.1, to generate the final `data/input.txt` file used by the C++ application.

  - Extracts a random query sequence, applies noise, and saves it to `data/query.txt`, printing the ground truth index to standard output.

2. Finally we can execute one of the scripts in `exec/` folder. The choice of script depends on the specific requirement: see the README for more details.

2 Sequential

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

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

2.1 Pipeline

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

1. Data Loading
   We allocate memory and load the input file into it. We reorganize the data into an Array of Structures (AoS) memory pattern. The implementation is in the data_loader.cpp file.

2. Query Loading
   We load the query sequence of length M from the file *data/query.txt*. As described in Section 1.3, the noise generation and extraction are handled by the external data preparation script. The implementation is in the `query_loader.cpp` file.

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

4. Reporting
   We return the index of the best match and the relative SAD value. The ground truth index is displayed by the data generation script for manual verification.

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 copying (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 efficiency, but we lose the simplicity of stream parsing and have to handle everything manually. To gain this optimization, we must return to the C pointer approach.

Additionally, since the parsing logic accesses memory strictly linearly, we utilize the `madvise` system call with the `MADV_SEQUENTIAL` flag. In this way, we instruct the kernel to optimize 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 to access the d-th dimension of the t-th timestamp element, we can use $f(t,d) = buffer[t \cdot D + d]$.

Since we adopt a sliding window approach, this memory pattern technique allows us to maximize spatial locality during the distance calculation. The algorithm, described in Section 1.1, in fact has an inner loop through the D dimension, and an outer loop through the input length N; when we load into the cache the d-th dimension of the t-th timestamp, we also load the neighboring 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 vectorization is possible. In particular, we have to declare the `restrict` clause in the 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 is a collection of float numbers and for each timestamp we have $D = 6$ dimensions; we can see the data like a $N \times D$ matrix, where N is the number of elements. The cache lines are typically composed of 64 Bytes and the CPU works better if the data are aligned with this value. Since a float is loaded in 4 Bytes, 6 floats occupies 24 Byte; we can optimize the performance introducing two more dimensions (i.e., two more floats) so that one timestamp occupies 32 Byte in the cache line.

This padding approach doesn't change the SAD distance result, but allows us to reduce cache misses and also optimize the vectorization 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 theory, but the real test shows different results. The Section 4.1.3 shows these results.

2.4 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, as we want to compare "apples to apples". On the GPU, thread divergence makes early abandoning less effective or counterproductive for this specific kernel structure. Thus, we have parameterized it with the `ENABLE_EARLY_ABANDONING` compilation flag, which is `false` by default.

To see this analysis go to Section 4.1.1.

3 Parallel

In this chapter, we analyze the choices made for the parallel version of the program. The main goal of this version is to maximize the throughput, i.e., obtain the maximum speed-up related to the sequential version, described in Chapter 2.

3.1 Memory Layout

GPUs are designed 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 data layout, and the Structure of Arrays (SoA) matches the GPU coalesced memory access perfectly.

3.1.1 SoA Memory Pattern

The main problem of the AoS in the GPU is given by the warp (i.e., a group of 32 threads) parallel execution: they execute the same instruction 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 loads its first dimension data in memory, it also loads the first dimension of the other threads; when these other threads search for their first dimension data in memory, the data will be already present.

3.1.2 Inline mapper equation

Thus, the data is stored in a Structure of Arrays pattern, so in a linear long array T where the data related to each dimension are contiguous. 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 size of the dataset, i.e. the number of total timestamps.

3.2 Memory Hierarchy

In GPU programming, it is important to define the memory hierarchy. The objective is to minimize data transfer from the main global memory, which is the slowest, to the thread registers. 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 cannot be too long: in our case, considering the default length of 64 described in Section 1.3, we have circa 1.5 KB, so it fits easily within the 64KB constant memory limit. This guarantees a huge performance benefit: we do not have to transport 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 cooperate. We use it to perform a block-wise reduction and to implement 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 $\times query\_length$.

  - 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 shift the data to the disk during the transfer from the CPU memory to the GPU memory. This is addressed thanks to pinned memory: we have to say to the host OS that that

memory cannot be paged.

2. Query Loading (Host)
   The Host loads the query vector Q from `data/query.txt`. The query data is then uploaded to the GPU Constant Memory.

3. 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.

4. 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 retrieved from 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 $D(tid) = \sum_{d=0}^{D-1} \sum_{m=0}^{M-1} |Q[m, d] - T_{shared}[tid_{local} + m]|$.

5. Block Reduction (Device)
   This is the critical optimization step: writing every thread's result to Global Memory isn't efficient since the global memory is the slower memory and implies the transfer of all data from the global memory to the host memory, and we know that the PCI bus is the main bottleneck. To avoid this problem we perform a block-grain reduction: we define the best index i_j^* for each block j using 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 if we perform the reduction sequentially on the CPU. In this scenario, the choice of the *block_size* value is critical.

6. 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.

7. 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 i-th thread 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 least one thread for each timestamp, so we can define the dimensions of the grid, i.e., the number of blocks, 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 threads per warp, the smallest chunk of thread that is allocated on cores.

- Reduction efficiency
  The kernel output is composed by $N/block\_dim$ partial results: a larger $block\_dim$ 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 too big, we are limiting the scheduler's ability to switch contexts and hide memory latency because we can have too few blocks within an 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 physically 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 theoretically in Section 3.1.2, becomes practically $tid = blockIdx.x \times blockDim.x + threadIdx.x$, where: *blockIdx.x* defines the number of blocks preceding 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 analyze both the sequential and parallel code.

The experiments were conducted using these settings:

- The dataset was downloaded with a multiplier of 50. This scaling factor, described in Section 1.2.1, was determined empirically to ensure that the execution time of the search algorithm is not too rapid. It implies that the *input.txt* file is circa 3.5 GB in size.
- The query length default is 64 timestamps.

The server specifications 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 analyze the results of the sequential version described in Chapter 2.

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

4.1.1 Early Abandoning

We tested the sequential performance with and without the Early Abandoning optimization. As we said in Section 2.4, the correct baseline is with this technique disabled.

With early abandoning enabled we obtain a circa 2.87× speedup: the matching time, i.e. the execution time for searching in the input file, with it enabled is about 8.84 seconds, and without is circa 25.35 seconds.

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 bound:

- Pattern matching: This dominates the total execution time with circa 65%.
- Data loading: Data loading accounts for circa 30% of the execution time.

4.1.3 Padding

As we said in Section 2.3.2, we can introduce a padding to the data to align 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 our matching algorithm: its execution time with padding is circa 35.8 seconds, but without 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 padding values, since they are the 33% of the real data: if we work with a longer data length, probably the padding increases the speed of our code; in this case the cpu prefers to handle itself the load in memory of the unpadded data.

4.1.4 Query Length

If we use a longer query we will obtain a greater execution time of the matching algorithm. We have performed three experiments:

- **32 timestamps:** The execution time is circa 10.4 seconds.
- **64 timestamps (default):** The execution time is circa 26.2 seconds.
- **128 timestamps:** The execution time is circa 56.3 seconds.

4.1.5 Data Loading

The first approach for the data loading step used the classic C++ stream. The time required for the parsing of the input text file was approximately 47 seconds. With 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 by executing our CUDA code.

The first difference between this and the sequential execution can be seen by analyzing the total execution time of code. With the default 64 timestamps query length 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 load time, i.e. the time that the CPU takes to bring the data from the disk to the memory. Let's take one experiment 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 becomes irrelevant.

4.2.2 Query Length

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

In the parallel version changing the query length 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 512 query length we have matching time of 0.328047 seconds and with a huge 2048 query length we have 1.215921 seconds.

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

4.2.3 Block Size

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

- 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.

With a block size of 8, we have a waste of hardware resources because the warp size is 32. Instead, with a block size of 1024, we have a huge barrier synchronization cost since also only one thread has a longer execution time all other 31 warps have to wait for the warp of that thread.