Pattern Recognition with CUDA and Python

Andrea Bigagli - andrea.bigagli1@edu.unifi.it

February 2026

Abstract

This project studies the parallelization of a time series pattern matching algorithm based on the Sum of Absolute Differences (SAD) metric using two different parallel computing paradigms. A GPU implementation based on CUDA and a CPU implementation based on Python multiprocessing are developed and compared against a sequential baseline. Experimental results on the ECG5000 dataset show that the CUDA version achieves significantly higher speedup than the CPU-based approach, highlighting the impact of architectural differences and parallelization overhead.

1 Introduction

Time series pattern matching is a common operation in several application domains, including signal processing and biomedical data analysis. When applied to large datasets, this task becomes computationally expensive due to the large number of sliding windows and distance computations required.

The structure of the problem is naturally data-parallel, since each window can be processed independently. For this reason, time series pattern matching represents a suitable benchmark to evaluate different parallel computing paradigms.

This project, developed within the *Parallel Computing* course, investigates the same algorithm implemented using two heterogeneous parallel approaches: a GPU-based solution using CUDA in C++ and a CPU-based solution using Python multiprocessing. The two implementations are compared against sequential baselines in order to analyze execution time, speedup, and scalability on real-world ECG data.

2 Problem Description

The computational problem addressed in this project consists in performing pattern matching on time series data using the Sum of Absolute Differences (SAD) metric. Given a long time series S of length N and a shorter reference pattern Q of length M, the SAD value associated with a starting position $start$ is defined as:

$$SAD(start) = \sum_{j=0}^{M-1} |S[start + j] - Q[j]| \,. \quad (1)$$

The objective is to compute the SAD value for all valid starting positions $start \in [0, N - M]$ and to identify:

- $bestIdx$, the starting index that minimizes the SAD value;
- $bestSAD$, the minimum SAD value obtained.

This problem exhibits a high degree of inherent data parallelism, since the computation of the SAD value for each window is completely independent of all the others. As a result, the workload can be naturally decomposed by assigning different windows of the time series to different processing units, making the problem well suited for both GPU-based and CPU-based parallel implementations.

3 Algorithm Overview

The pattern matching task is based on a sliding-window evaluation of the Sum of Absolute Differences

(SAD) metric. Given the input signal S and the pattern Q, the algorithm evaluates all valid windows of length M in S and selects the one that minimizes the SAD value.

The computation consists of two nested loops: the outer loop iterates over all valid starting positions $start$, while the inner loop accumulates the absolute differences between corresponding elements of S and Q. The final result is obtained by selecting the minimum SAD value and its corresponding index. This structure is well suited for parallel execution, since each window can be processed independently.

Algorithm 1 summarizes the reference version of the SAD-based pattern matching algorithm.

Algorithm 1 SAD-based time series pattern matching (reference algorithm)

```
Require: Time series $S$ (length $N$), pattern $Q$ (length $M$)
Ensure: $bestIdx$, $bestSAD$
 1: $bestSAD \leftarrow +\infty$, $bestIdx \leftarrow -1$
 2: for $start \leftarrow 0$ to $N - M$ do
 3:     $sad \leftarrow 0$
 4:     for $j \leftarrow 0$ to $M - 1$ do
 5:         $sad \leftarrow sad + |S[start + j] - Q[j]|$
 6:     end for
 7:     if $sad < bestSAD$ then
 8:         $bestSAD \leftarrow sad$, $bestIdx \leftarrow start$
 9:     end if
10: end for
11: return $bestIdx$, $bestSAD$
```

4 Dataset and Experimental Setup

The experimental evaluation is performed on the ECG5000 dataset, which contains time series representing electrocardiogram (ECG) signals from the UCR Time Series Classification Archive (`https://www.timeseriesclassification.com/description.php?Dataset=ECG5000`). Each sequence represents a single heartbeat resampled to a fixed length of $M = 140$ samples. The dataset is derived from an ECG record originally obtained from PhysioNet and then preprocessed by extracting individual beats and interpolating them to equal length.

The input signal S is obtained by concatenating multiple ECG sequences, resulting in a time series of length $N \approx 630{,}000$. The reference pattern Q corresponds to a single ECG sequence of length $M = 140$.

To avoid trivial perfect matches, a small artificial modification is applied to the pattern Q. The expected best match for the considered configuration is $bestIdx = 1400$ with $bestSAD = 1$, which is used to validate the correctness of all implementations.

All experiments are conducted on the same machine in order to ensure fair comparison between the different approaches. The system configuration is summarized in Table 1.

Table 1: System specifications used for the experiments

Parameter	**Specification**
Operating System	Windows 11
CPU	Intel Core i7-13620H
Logical processors	16
Memory	16 GB RAM
GPU	NVIDIA RTX 4050
Environment	Visual Studio 2022 + CUDA Toolkit

4.1 Dataset Tool and Input Generation

Input files used by the implementations are generated through a dedicated preprocessing tool written in C++. The tool reads the ECG5000 test file, where each row contains a label followed by $M = 140$ floating-point samples. To simplify computation and ensure consistent behavior across C++/CUDA and Python, samples are converted to integers by applying a fixed scaling factor and rounding:

$$x_{int} = \text{round}(x \cdot 1000).$$

The long signal S is built by concatenating a fixed number of rows (in our configuration, 5000 rows), resulting in $N \approx 5000 \cdot 140 \approx 630{,}000$ elements. The query pattern Q is extracted from a single selected row (row index 10) and a minimal artificial perturbation is introduced (incrementing the first element) to avoid a trivial perfect match. The tool produces two plain-text files: `S.txt` containing one integer per line (the concatenated signal) and `Q.txt` containing the pattern values.

This preprocessing step ensures that all implementations run on the exact same inputs and allows reproducible performance measurements and correctness checks.

5 CPU Implementations

CPU implementations are used as baselines to validate correctness and to compute speedup values for the parallel solutions. Two versions are considered: a sequential implementation and a parallel CPU implementation based on OpenMP.

5.1 Sequential Baseline

The sequential baseline directly implements the reference SAD-based pattern matching algorithm described in Section 1. All valid starting positions are evaluated in a single thread, and the global minimum is updated at each iteration. Although conceptually simple, this approach is computationally expensive, since it performs $(N - M + 1) \cdot M$ absolute-difference operations and does not exploit any form of parallelism.

5.2 OpenMP Parallel Implementation

The OpenMP version parallelizes the outer loop over the starting positions using a `parallel for` directive. Each thread evaluates a subset of windows independently and maintains a private local minimum. After the parallel region, a final reduction step selects the global minimum among all local minima. This strategy avoids data races during the SAD computation phase and limits synchronization overhead to a single reduction step.

6 CUDA Implementation

The CUDA solution maps each valid starting position of the sliding window to one GPU thread, which independently computes the corresponding SAD value. Threads are organized into blocks and blocks into a grid, allowing the computation to scale to large input sizes. Each thread writes its result into a device output array of size $N - M + 1$.

After kernel execution, the output array is copied back to the host, where the final pair $(bestIdx, bestSAD)$ is obtained by a linear scan. This design keeps the device-side computation fully data-parallel and avoids introducing synchronization among threads.

Three kernel variants are implemented to evaluate different memory placements for the pattern Q: a baseline version using **global memory**, an optimized version using **constant memory** (`__constant__`), and a variant that loads the pattern into **shared memory** at the block level. All variants perform the same computation and differ only in how the pattern is accessed.

Kernel execution time is measured using CUDA events, while end-to-end execution time also includes device-to-host transfers and the final reduction on the CPU. This distinction makes it possible to separate the pure computational speed of the GPU kernel from the overall application cost.

7 Python Multiprocessing Implementation

A second parallel implementation is developed in Python using the `multiprocessing` module in order to exploit multiple CPU cores for a CPU-bound workload. This choice avoids the limitations of Python threading due to the Global Interpreter Lock and allows independent processes to execute in parallel on different cores.

The same SAD-based reference algorithm is adopted. The range of valid starting positions is partitioned into fixed-size chunks, and each worker process is assigned one or more chunks. For each assigned chunk, the worker computes a local minimum $(bestIdx_{loc}, bestSAD_{loc})$. After all workers complete, the main process performs a final reduction to obtain the global minimum.

The choice of the number of processes and the chunk size has a significant impact on performance. Very small chunks increase scheduling and communication overhead, while very large chunks may lead to load imbalance. On the target system, performance improves up to a moderate number of processes, after which the overhead of process management and inter-process communication limits scalability.

Execution time is measured as the total wall-clock time of the Python program, including process creation, computation, and final reduction. This provides a realistic measure of the overall performance of the process-based parallel approach on CPU.

8 Results

This section reports the experimental results obtained for the different implementations of the SAD-based pattern matching algorithm. Execution times are reported for the sequential CPU baseline, the OpenMP parallel version, the CUDA GPU implementations, and the Python multiprocessing version. Speedup values are computed with respect to the sequential CPU baseline in order to compare the effectiveness of the different parallelization strategies.

8.1 Execution Time Comparison

Table 2 reports the end-to-end execution times for all implementations on the ECG5000 dataset ($M = 140$). For CUDA, end-to-end time includes kernel execution, device-to-host memory transfer, and the final reduction on the CPU.

Table 2: End-to-end execution times (ECG5000, $M = 140$)

Implementation	**Time (ms)**
CPU Sequential	544.0
Python multiprocessing	150.0
CPU OpenMP (16 threads)	89.0
CUDA Global (total)	7.0
CUDA Shared (total)	6.9
CUDA Constant (total)	6.4

8.2 CUDA Kernel vs End-to-End Time

To better isolate the pure device-side computation, Table 3 reports both kernel execution time and end-to-end time for the CUDA variants.

Table 3: CUDA execution times (ECG5000, $M = 140$)

Version	**Kernel time (ms)**	**Total time (ms)**
Shared memory	4.4	6.9
Global memory	4.3	7.0
Constant memory	3.8	6.4

The constant memory version achieves the lowest kernel execution time, while the shared memory variant does not provide additional benefits with respect to the baseline. The gap between kernel and total time highlights the overhead introduced by memory transfers and host-side postprocessing.

8.3 Speedup

Figure 1 reports the speedup with respect to the sequential CPU baseline. For CUDA, both kernel-only and end-to-end speedup are shown in order to highlight the impact of communication and postprocessing overhead.

Table 4: Speedup with respect to the CPU sequential baseline (ordered by magnitude)

Implementation	**Speedup**
CUDA Constant (kernel)	143.1
CUDA Global (kernel)	126.5
CUDA Constant (total)	85.0
CUDA Global (total)	77.7
CPU OpenMP	6.1
Python multiprocessing	3.6

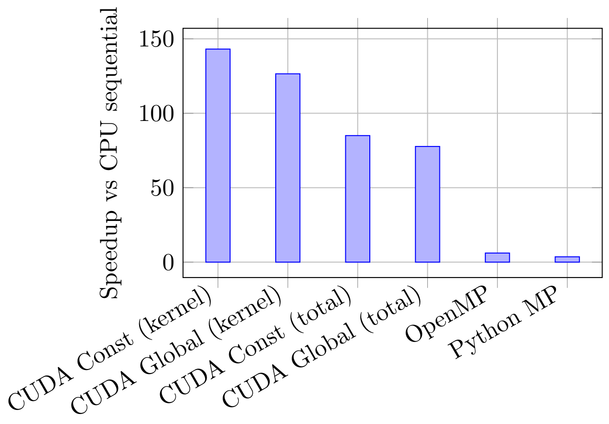

Figure 1: Speedup with respect to the CPU sequential baseline (ordered by magnitude).

The results show a clear performance hierarchy among the tested implementations. OpenMP provides a moderate speedup by exploiting thread-level parallelism on the CPU. The CUDA implementations significantly outperform both CPU-based approaches, with the constant memory variant achieving the best performance. The difference between kernel-only and end-to-end speedup highlights the non-negligible impact of data transfers and host-side postprocessing. The Python multiprocessing version provides limited scalability due to process management overhead and communication costs, remaining well below GPU performance.

9 Discussion

The experimental results highlight how the same reference algorithm exhibits very different performance depending on the chosen parallelization strategy and target architecture.

The GPU-based implementations clearly benefit from the high degree of data parallelism inherent to the SAD computation. The constant memory variant provides the best performance, as the pattern is small and shared by all threads, allowing efficient caching and broadcast. The shared memory variant does not introduce significant improvements, since the dominant cost is the access to the input signal in global memory and the additional synchronization overhead offsets the potential benefits.

The OpenMP implementation achieves a moderate speedup by exploiting thread-level parallelism on the CPU. However, scalability is limited by the lower degree of parallelism available on multi-core CPUs and by the overhead associated with thread management and reduction.

The Python multiprocessing version provides further insight into process-based parallelism on CPU. Although it achieves a speedup with respect to the sequential Python baseline, its performance remains significantly below both OpenMP and CUDA due to the overhead of process creation, scheduling, and inter-process communication. This comparison highlights the impact of the programming model and runtime overhead on the achievable performance, even when the underlying algorithm is the same.

10 Conclusions

This work studied the parallelization of a time-series pattern matching algorithm based on the Sum of Absolute Differences (SAD) metric using different programming models. The same reference algorithm was implemented in a sequential CPU version, a multi-threaded CPU version, a GPU-based CUDA implementation, and a Python multiprocessing version.

The experimental results show that the SAD computation is particularly well suited for data-parallel execution on GPUs, which achieve orders-of-magnitude speedup with respect to the sequential baseline. Among the tested CUDA variants, placing the pattern in constant memory yields the best performance, while the shared memory variant does not provide additional benefits in this specific scenario.

CPU-based parallelism provides a moderate speedup, while the Python multiprocessing implementation highlights the impact of runtime overhead on scalability. Overall, the results confirm that both the programming model and the underlying architecture play a key role in determining achievable performance.