UNIVERSITÀ
DEGLI STUDI
FIRENZE

Pattern Recognition on Time Series with CUDA and Python

Parallel Computing

Andrea Bigagli 7036221

A.A. 2025/2026

Introduction

- Pattern recognition on time series is a common task in **signal processing** and **biomedical analysis**
- Brute-force pattern matching is computationally expensive on large signals
- Each window comparison is independent
- The problem is naturally data-parallel and well suited for parallel architectures
- **Goal:** compare different parallel programming models on the same algorithm and dataset

Problem Definition

Input:

- Long time series $\boldsymbol{S}$ of length $\boldsymbol{N}$
- Short query pattern $\boldsymbol{Q}$ of length $\boldsymbol{M}$

For each valid position $\boldsymbol{start} \in [0, N-M]$, compute:

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

Output:

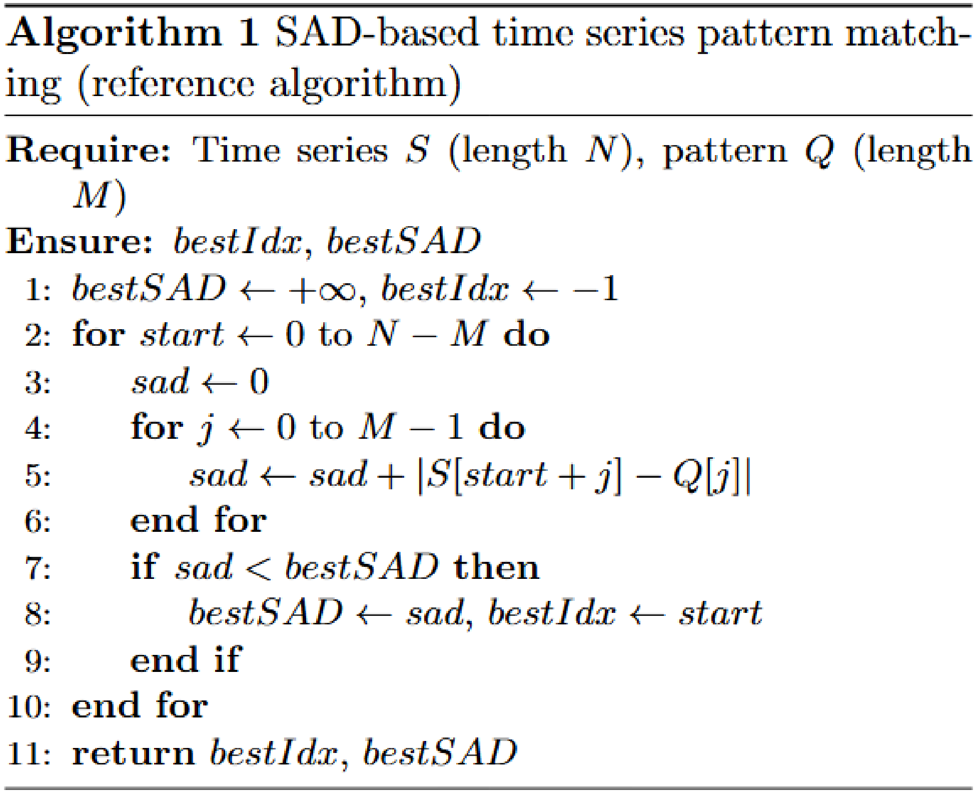
- $bestIdx = \arg\min SAD(start)$
- $bestSAD = \min SAD(start)$

Algorithm Overview

- Sliding-window pattern matching:
- For each valid position:
    - compute **SAD**
    - update global **minimum**
- Two nested loops
- Time complexity:

$$O((N - M + 1) \cdot M)$$

- Same reference algorithm for all implementations



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

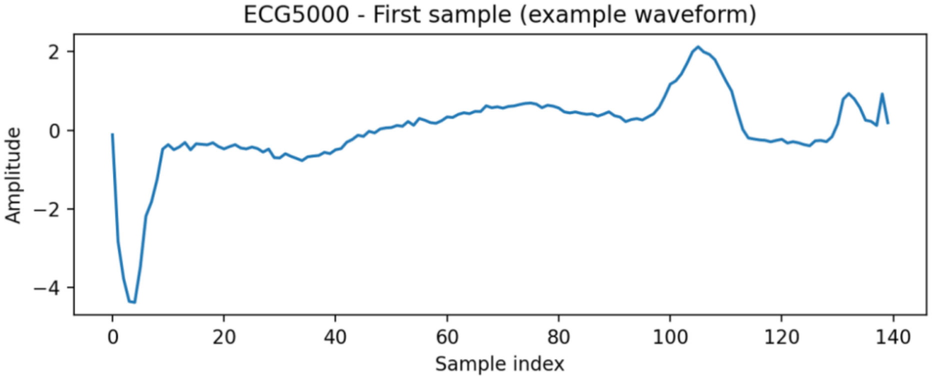
```
Require: Time series $S$ (length $N$), pattern $Q$ (length $M$)
Ensure: $bestIdx$, $bestSAD$
$bestSAD \leftarrow +\infty$, $bestIdx \leftarrow -1$
for $start \leftarrow 0$ to $N - M$ do
    $sad \leftarrow 0$
    for $j \leftarrow 0$ to $M - 1$ do
        $sad \leftarrow sad + |S[start + j] - Q[j]|$
    end for
    if $sad < bestSAD$ then
        $bestSAD \leftarrow sad$, $bestIdx \leftarrow start$
    end if
end for
return $bestIdx$, $bestSAD$
```

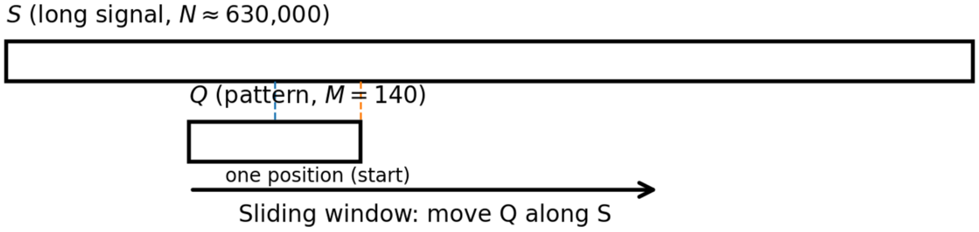
Dataset: ECG5000

The dataset used in this project is :

- Dataset: **ECG5000** (UCR Time Series Classification Archive)
- Each sample: **140 values** (single heartbeat)
- Real ECG signals (biomedical time series)
- Pattern Q: one ECG sample
- Signal S: concatenation of many ECG samples
- Final size:

$$|S| \approx 630{,}000,\ |Q| = 140$$





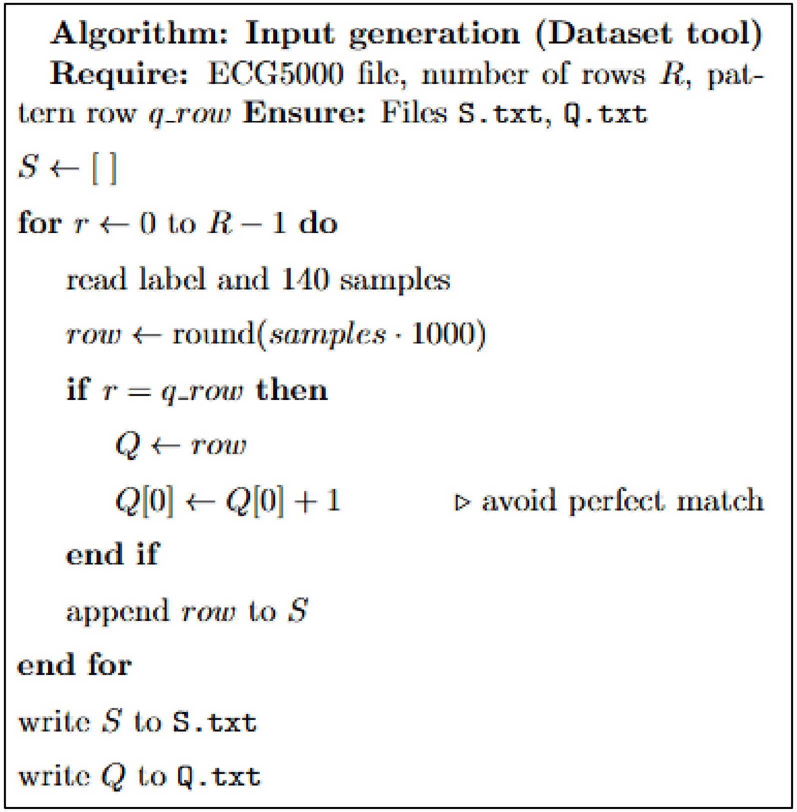
Source: UCR Time Series Classification Archive (ECG5000)

Input Generation (Dataset Tool)

- Dedicated C++ tool to generate inputs for all implementations
- Reads ECG5000 rows (label + 140 samples)
- Converts samples from float to int:

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

- Builds
  - S: concatenation of many rows (es, 5000)
  - Q: one selected row (es, row 10)
- Adds a small perturbation to Q to avoid a perfect match
- Outputs two files: **S.txt** and **Q.txt** (one integer for line)



Algorithm: Input generation (Dataset tool)

Require: ECG5000 file, number of rows R, pattern row q_row **Ensure:** Files `S.txt`, `Q.txt`

```
$S \leftarrow [\,]$
for $r \leftarrow 0$ to $R - 1$ do
    read label and 140 samples
    $row \leftarrow \mathrm{round}(samples \cdot 1000)$
    if $r = q\_row$ then
        $Q \leftarrow row$
        $Q[0] \leftarrow Q[0] + 1$          ▷ avoid perfect match
    end if
    append $row$ to $S$
end for
write $S$ to S.txt
write $Q$ to Q.txt
```

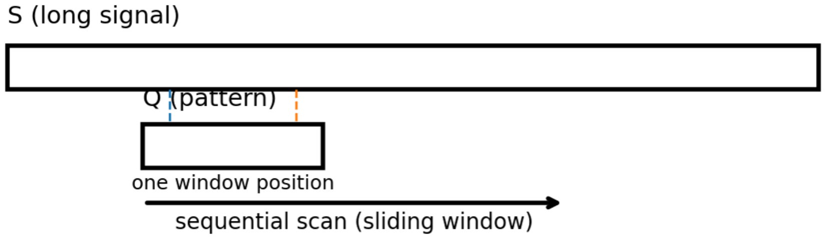
CPU Sequential Baseline

- Reference implementation (correctness + baseline)
- Direct implementation of the reference algorithm
- Sliding-window SAD computation
- One window processed at a time
- Used as correctness reference
- Used as baseline for speedup computation
- Worst-case complexity:

$$O((N - M + 1) \cdot M)$$

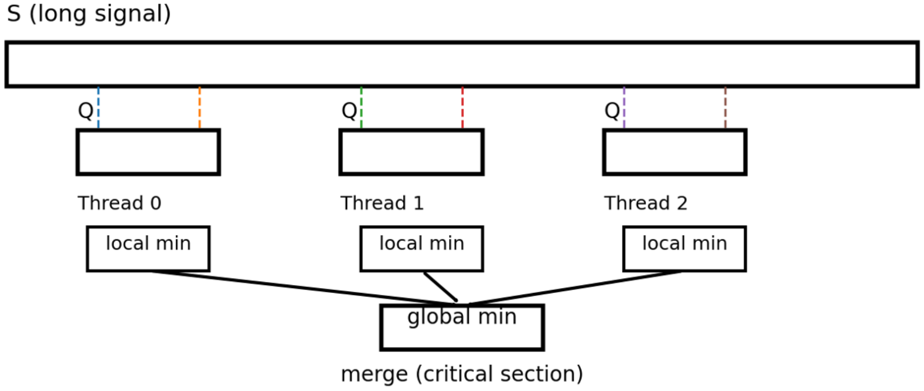
$$\text{for } start = 0, \ldots, N{-}M : \quad SAD(start) = \sum_{j=0}^{M-1} |S[start + j] - Q[j]|$$

$$(bestIdx, bestSAD) = \arg \min_{start \in [0,\, N-M]} SAD(start)$$



CPU Parallel Implementation (OpenMP)

- Parallelize the outer loop over window start positions
- **Each thread computes SAD on its assigned windows** $(bestIdx_{local}, bestSAD_{local})$
- **Thread-local minimum**
- Global minimum obtained by **merging local minima**
- Synchronization only in the merge step (critical section)
- Scalability limited by synchronization + memory bandwidth



Independent windows processed in parallel; final minimum obtained by merging thread-local results.

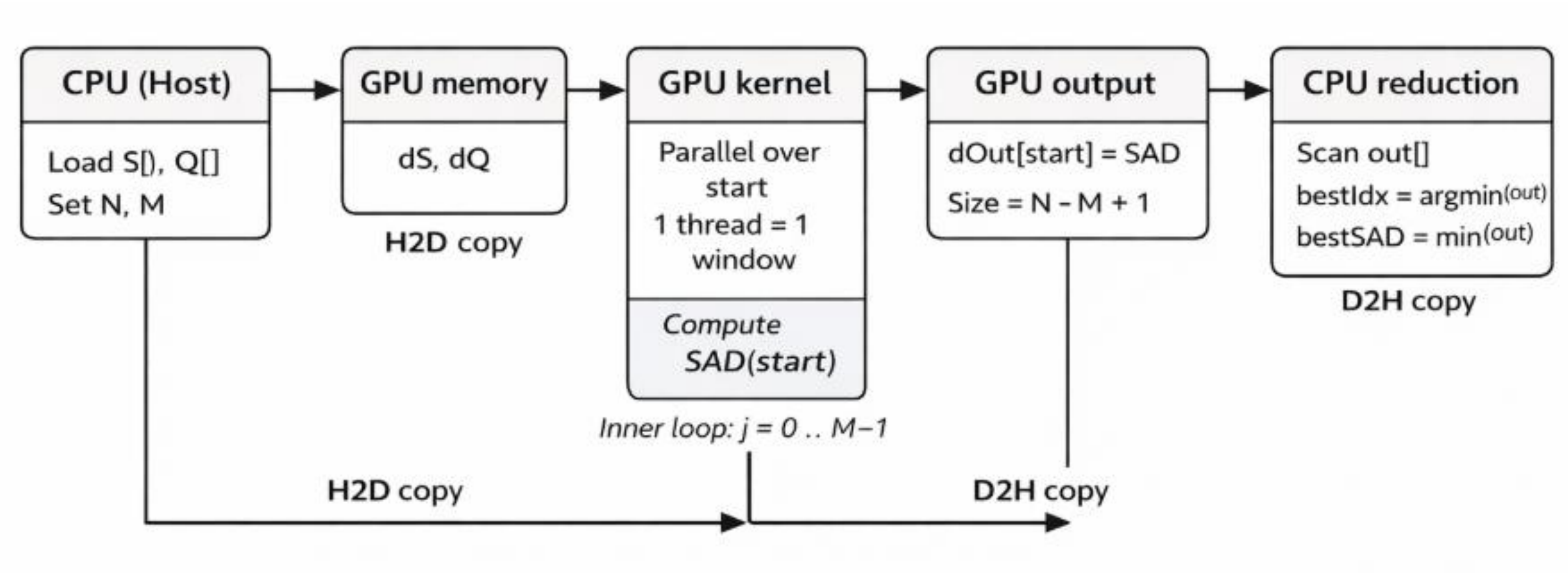
CUDA Implementation

- **Parallelize the outer loop** over window start positions (start)
- **1 GPU thread = 1 window** → computes one value SAD(start)
- **Inner loop sequential** inside each thread (j = 0..M-1, with M = 140 in ECG5000)
- Each thread writes one output value:
  → out[start] = SAD(start) (output size = N - M + 1)
- **No synchronization required** during SAD computation (windows are independent → no shared state)
- **CPU post-processing:** final reduction on out[]
  → bestIdx = argmin(out), bestSAD = min(out)

C++ / CUDA implementation details

- **CUDA Runtime API** for kernel launch and memory management
  (cudaMalloc, cudaMemcpy, cudaMemcpyToSymbol)
- **Host-side containers:** std::vector<int> for S and Q
- **Kernel timing:** cudaEvent_t to measure kernel execution time only
- **End-to-end timing:** std::chrono::high_resolution_clock (H2D + kernel + D2H + CPU reduction)
- **Output type:** long long to safely accumulate M = 140 absolute differences
- **Multiple memory variants implemented:**
  Global memory (baseline), Constant memory, Shared memory

CUDA Implementation: Execution Flow



CPU (Host)

Load S[), Q[]

Set N, M

GPU memory

dS, dQ

H2D copy

GPU kernel

Parallel over start

1 thread = 1 window

Compute SAD(start)

Inner loop: j = 0 .. M−1

GPU output

dOut[start] = SAD

Size = N – M + 1

CPU reduction

Scan out[]

bestIdx = argmin(out)

bestSAD = min(out)

D2H copy

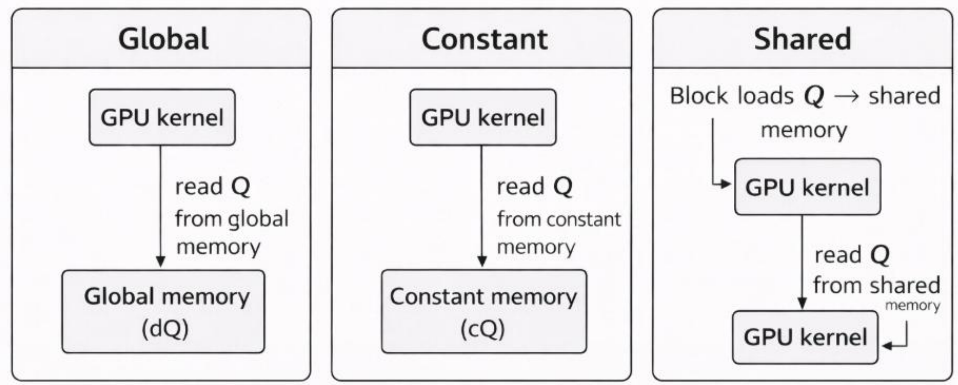
H2D copy

D2H copy

CUDA Memory Variants (Global vs Constant vs Shared)

The same CUDA kernel is evaluated using three different memory placements for the pattern Q, to analyze the impact of memory access on performance.

- Same kernel logic in all versions
  → only the memory placement of Q changes
- **Global (baseline):** read Q from global memory
- **Constant:** read Q from constant memory (cached, broadcast)
- **Shared:** load Q into shared memory (per block)
- Performance differences due to memory access patterns



Global

GPU kernel

read Q from global memory

Global memory (dQ)

Constant

GPU kernel

read Q from constant memory

Constant memory (cQ)

Shared

Block loads Q → shared memory

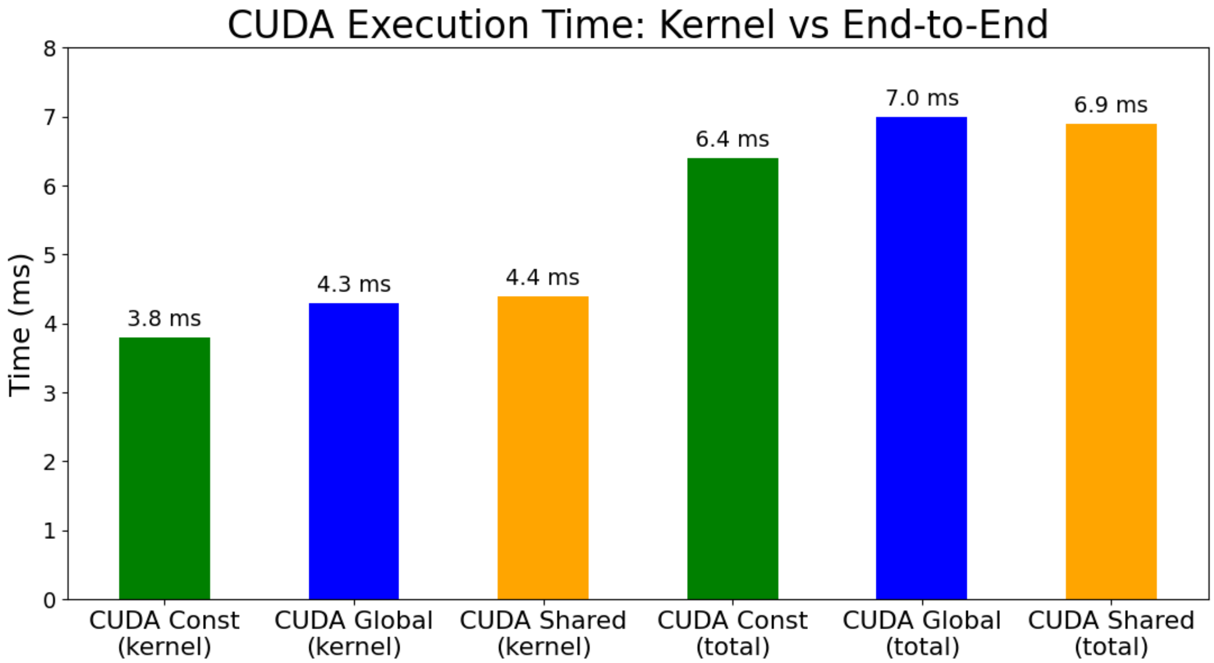
GPU kernel

read Q from shared memory

GPU kernel

Q is small and read-only → ideal for constant memory.

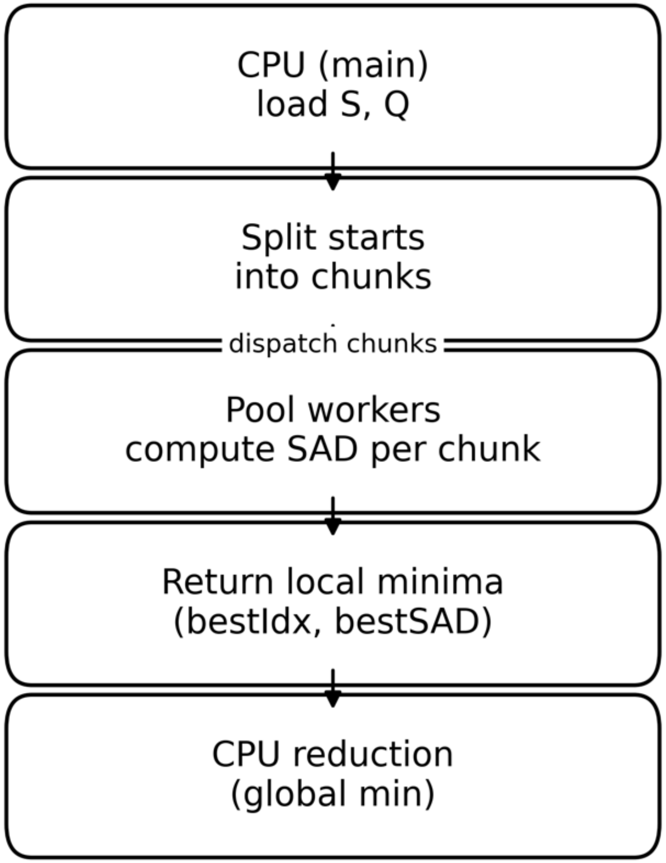
CUDA Execution Time (Kernel vs End-to-End)



- **Kernel** time isolates pure GPU computation.
- **End-to-end (total)** includes transfers + CPU reduction.
- Constant memory is the fastest kernel variant.

Python Multiprocessing Implementation

- **Parallelize the outer loop** over window start positions
- Use **multiprocessing.Pool** to run workers on multiple CPU cores
- Split the search space into **chunks of windows**
- Each worker computes SAD on its chunk and returns a **local minimum**
- Main process performs the **final reduction** to get (bestIdx, bestSAD)
- Avoids the GIL, but introduces **process and communication overhead**
- Same algorithm as C++ versions, different parallel model


CPU (main)
load S, Q

Split starts
into chunks

dispatch chunks

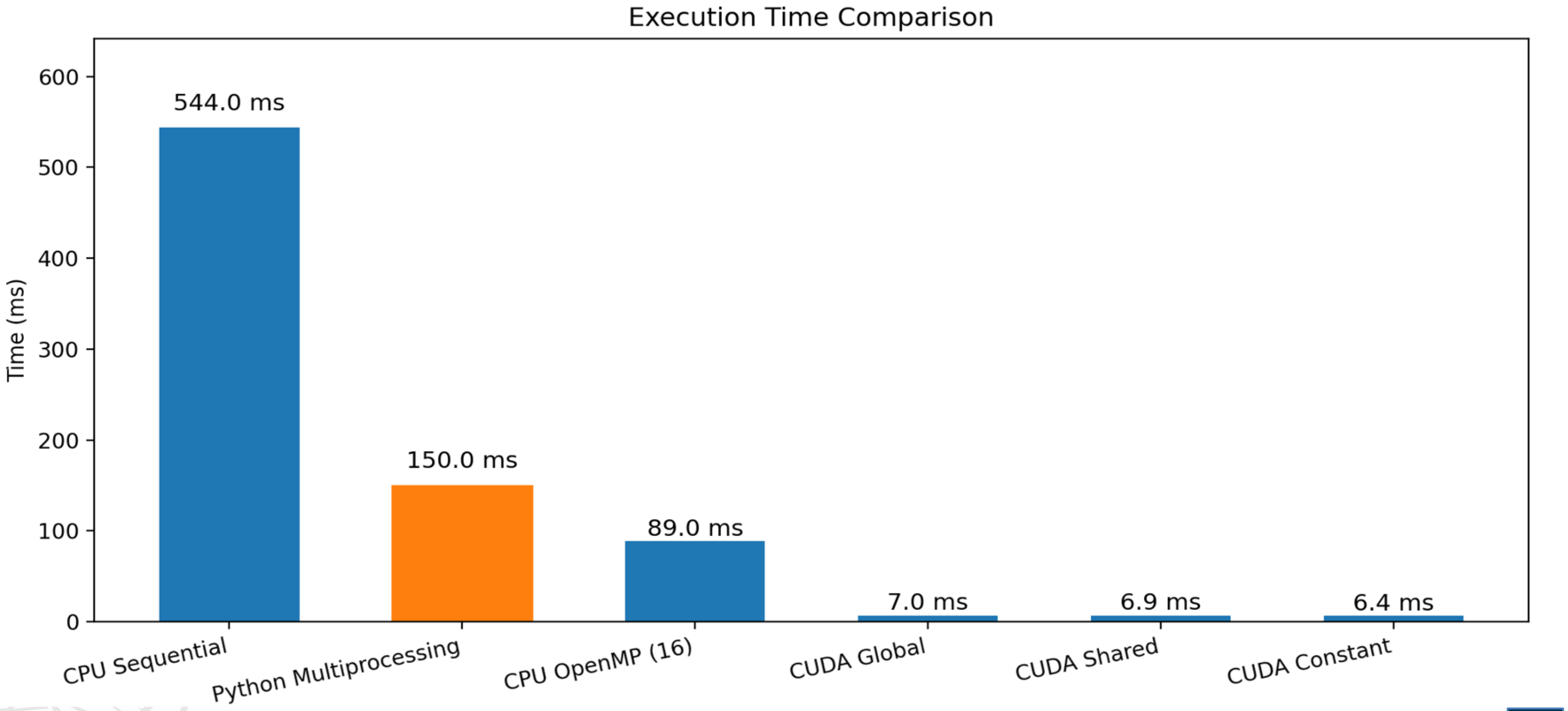
Pool workers
compute SAD per chunk

Return local minima
(bestIdx, bestSAD)

CPU reduction
(global min)

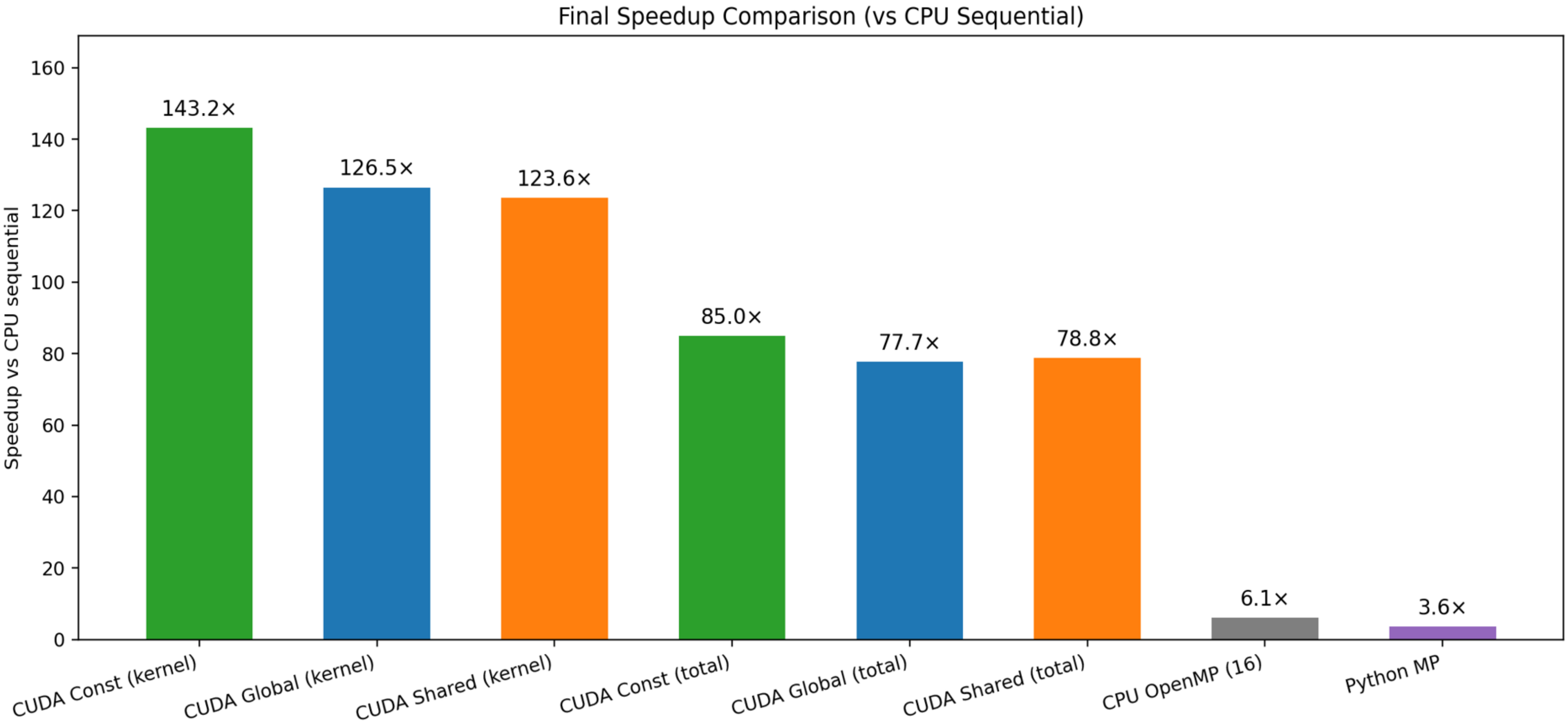
Overall Performance Comparison (Execution Time)

- **CUDA achieves orders-of-magnitude lower execution time** than CPU and Python
- **OpenMP improves over sequential**, but is limited by CPU parallelism
- **Python multiprocessing reduces time vs sequential, but overhead dominates**



ECG5000, M = 140, end-to-end execution time

Final Speedup Comparison



Conclusions

- **Parallelization Strategy:** *Outer-loop dominates*
  Parallelizing the window start positions (outer loop) exposes massive data-parallelism and scales well on both CPU (OpenMP) and GPU (CUDA).
  The inner loop remains sequential inside each thread and does not limit scalability.

- **Memory Placement (GPU): Constant > Global ≈ Shared**
  Placing the small, read-only pattern **Q** in **constant memory** yields the best performance due to caching and broadcast.

  Shared memory does not provide benefits in this scenario because loading and synchronization overhead outweigh potential gains.

- **System & Overhead Limits: Transfers and reduction dominate end-to-end time**
  Performance plateaus beyond 8–16 threads due to bandwidth saturation and efficiency drops monotonically, consistent with Amdahl's Law.