Time Series Pattern Recognition

Edoardo Sarri

Parallel Computing

Febbraio 2026

Introduction

Obiettivo

Implementazione in C++ di un algoritmo per il riconoscimento di pattern in serie temporali multivariate.

- **Input:** Serie temporale T di lunghezza N.
- **Query:** Pattern Q di lunghezza M ($M \ll N$).
- **Output:** Indice i che minimizza la distanza tra Q e la finestra temporale in T.

Versioni

Sono state realizzate due implementazioni:

1. **Sequenziale:** CPU based.
2. **Parallela:** GPU based (CUDA).

Introduction

Approccio

Utilizzo di una **Sliding Window**: per ogni istante temporale i si calcola la distanza tra la query e la porzione di input corrispondente.

Metrica: SAD

Sum of Absolute Differences (SAD). Per dati multivariati con D dimensioni:

$$D(i) = \sum_{m=0}^{M-1} \sum_{k=0}^{D-1} |Q[m,k] - T[i+m,k]|$$

L'obiettivo è trovare $i_{min} = \arg\min_i D(i)$.

Introduction

Dataset

Human Activity Recognition (HAR) dataset.

- **Dimensioni (D):** 6 (3 acc + 3 gyro).
- **Frequenza:** 50Hz.
- **Preprocessing:** Concatenazione di training e test set + Data Augmentation.

Input e Query

- **Input:** Stream continuo generato concatenando i campioni e applicando un moltiplicatore con rumore uniforme.
- **Query:** Sottosequenza estratta casualmente dall'input con aggiunta di rumore Gaussiano ($\mu = 0, \sigma = 0.01$).

Sequential

Data Loading

Il file di input può raggiungere grandi dimensioni (≈ 3.5 GB).

- **Problema:** Overhead `std::ifstream` (copie multiple, syscall).
- **Soluzione:** Memory Mapped I/O (`mmap`).
- **Ottimizzazione:** `madvise` con flag `MADV_SEQUENTIAL` per indicare l'accesso lineare al kernel.
- **Risultato:** Parsing ridotto da $\approx 47s$ a $\approx 13s$.

Sequential

Memory Layout - Array of Structures (AoS)

Struttura contigua per ogni timestamp: $[x_{1,1}, \ldots, x_{1,D}, x_{2,1}, \ldots]$.

- **Vantaggio:** Località spaziale ottimizzata per la sliding window.
- **Vectorization:** Uso di `-march=native` e keyword `restrict` per permettere al compilatore l'auto-vettorizzazione SIMD.

Padding

Allineamento dei dati alla cache line (64 Byte).

- Aggiunta di dimensioni fittizie per portare la dimensione del singolo timestamp a divisori di 64B.
- *Nota:* L'analisi sperimentale ha mostrato che l'overhead di

Sequential

Algoritmo SAD

Brute Force Sliding Window.

$$i^* = \arg \min_{i \in [0, N-M]} D(i)$$

Early Abandoning

Interruzione del calcolo della distanza se la somma parziale supera il minimo globale corrente.

- **Speedup:** $\approx 2.87\times$ se attivato.
- **Baseline:** Disattivato per confronto equo ("apple-to-apple") con la versione GPU (dove la divergenza dei thread lo rende inefficace).

Parallel

Memory Layout - Structure of Arrays (SoA)

Struttura trasposta per l'accesso GPU:
$[x_{1,1}, x_{2,1}, \ldots, x_{N,1}, x_{1,2}, \ldots]$.

- **Problema AoS su GPU:** Accessi non coalesced all'interno del warp.
- **Soluzione SoA:** Ogni thread accede a dati contigui in memoria per la stessa dimensione, massimizzando il throughput della memoria globale.

Parallel

Memory Hierarchy

Strategia per minimizzare la latenza della Global Memory:

- **Global Memory:** Dataset completo (T).
- **Constant Memory:** Query (Q).
  - Accesso broadcast a tutti i thread del warp.
  - Cache dedicata (64KB), sufficiente per query tipiche.
- **Shared Memory:** Tiling dell'input e riduzione parziale.
- **Registers:** Accumulatori locali per il calcolo SAD.

Parallel

Tiling & Reduction

1. **Loading:** I thread del blocco collaborano per caricare una "tile" di input dalla Global alla Shared Memory.
2. **Compute:** Calcolo SAD locale usando i dati in Shared Memory (riduzione accessi Globali di fattore $\approx M$).
3. **Block Reduction:** Riduzione parallela intra-blocco per trovare il minimo locale.
   - Riduce drasticamente le scritture in Global Memory.
   - Complessità $\mathcal{O}(\log(\text{block\_dim}))$.

Parallel

Grid Configuration

Mapping 1-to-1: 1 Thread = 1 Timestamp (sliding window start).

- **Block Size:** 256 thread (default).
- **Grid Size:** $\lceil N/256 \rceil$.
- Scelta bilanciata per occupazione SM ed efficienza della riduzione.

Analysis

Sequential Analysis

- **Tempo totale ($Q = 64$):** $\approx 38.8s$.
- **Early Abandoning:** Speedup $\approx 2.87\times$ (da $25.35s$ a $8.84s$ solo matching).
- **Padding:** Non efficace in questo contesto (overhead dati inutili).

Analysis

Parallel Analysis

- **Tempo totale ($Q = 64$):** $\approx 14.1s$.
- **Bottleneck:** Il caricamento dati domina l'esecuzione ($\approx 14.3s$ di loading vs $0.05s$ di kernel).
- **Scalabilità:** Il tempo di matching è quasi irrilevante rispetto all'I/O. Aumentare Q ha impatto minimo sul totale.

Analysis

Block Size Analysis

Valutazione del tempo di kernel al variare della dimensione del blocco:

- **32 threads:** $0.061s$.
- **256 threads (scelta):** $0.055s$ (ottimo locale).
- **1024 threads:** $0.067s$ (overhead sincronizzazione barriera).

Analysis

Conclusioni

- La versione GPU offre un kernel estremamente performante, rendendo il problema **I/O bound**.
- L'approccio `mmap` è essenziale per gestire dataset di queste dimensioni.
- L'ottimizzazione dell'uso della memoria (Shared + Constant) è stata la chiave per le prestazioni GPU.

Time Series Pattern Recognition

Edoardo Sarri

Grazie