SAPIENZA
UNIVERSITÀ DI ROMA

Dipartimento di Informatica

Implementazione Parallela del K-Nearest Neighbors

Progetto di Programmazione di Sistemi Embedded e Multicore

Professore:
Salvatore Pontarelli

Studenti:
Bustamante Gabriel Alonso
Lin Can
Olivieri Alessio

Università di Roma La Sapienza

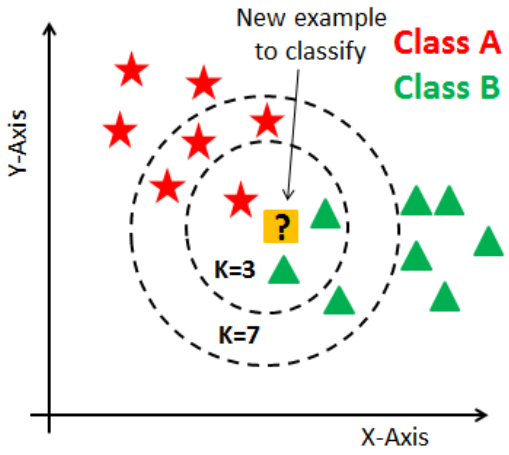
1 Introduzione

Questo progetto esplora tre implementazioni di un K-nearest neighbors in C, un'algoritmo fondamentale nell'apprendimento supervisionato non parametrico, che utilizza la distanza tra i dati per determinare la classe di appartenenza di un nuovo campione. Le implementazioni analizzate comprendono: una versione seriale tradizionale, una parallelizzata con Pthreads e una parallelizzata con CUDA.

1.1 K-nearest neighbors

Il K-nearest neighbors (K-NN) è un algoritmo di apprendimento supervisionato non parametrico utilizzato per la classificazione e la regressione. Funziona identificando i K punti di dati più vicini (i "vicini") rispetto a un nuovo campione, sulla base di una distanza predefinita, solitamente la distanza euclidea. La classe del nuovo campione viene determinata dalla maggioranza delle classi dei suoi vicini più prossimi. K-NN è semplice da implementare e non fa assunzioni particolari sui dati, rendendolo versatile per molteplici applicazioni. Tuttavia, può essere computazionalmente intensivo per grandi dataset, poiché richiede il calcolo della distanza tra ogni coppia di punti.

2 L'Algoritmo di K-nearest neighbors



2.1 Pseudocodice

Algorithm 1 Algoritmo di classificazione k-nearest neighbors

procedure KNN
  Input:
  D: un insieme di campioni di training $\{(x_1, y_1), \ldots, (x_n, y_n)\}$
  k: il numero di vicini più prossimi
  $d(x, y)$: una metrica di distanza
  x: un campione di test
  Output: La classe predetta per il campione di test x

  for ogni campione di training (x_i, y_i) in D **do**
    Calcola $d(x, x_i)$, la distanza tra x e x_i
  end for
  Sia $N \subseteq D$ l'insieme di campioni di training con le k distanze più piccole $d(x, x_i)$
  Ritorna l'etichetta maggioritaria dei campioni in N
end procedure=0

2.2 Obiettivi

Il nostro programma utilizza un dataset D composto da n punti, ognuno dei quali è definito da due coordinate (x, y). Data un'immagine I di dimensioni $H \times W$, abbiamo suddiviso I in una serie di piccoli quadrati q, ciascuno di dimensione SIDE $\times$ SIDE. Ogni quadrato q è stato classificato sul suo pixel centrale utilizzando l'algoritmo k-nearest neighbors (kNN). Infine, abbiamo colorato l'immagine in diversi colori a seconda della classe di appartenenza determinata per ciascun centro di quadrato da così ottenere l'area di appartenenza di ogni classe.

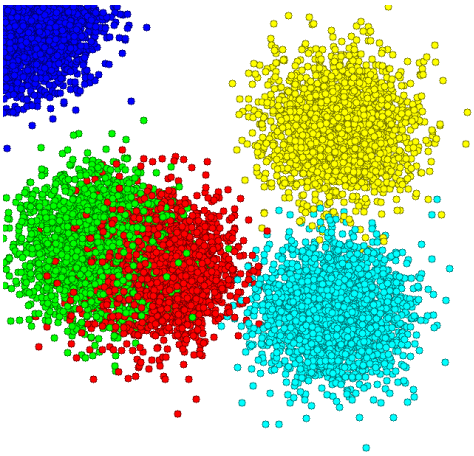


Figure 1: posizionamento punti

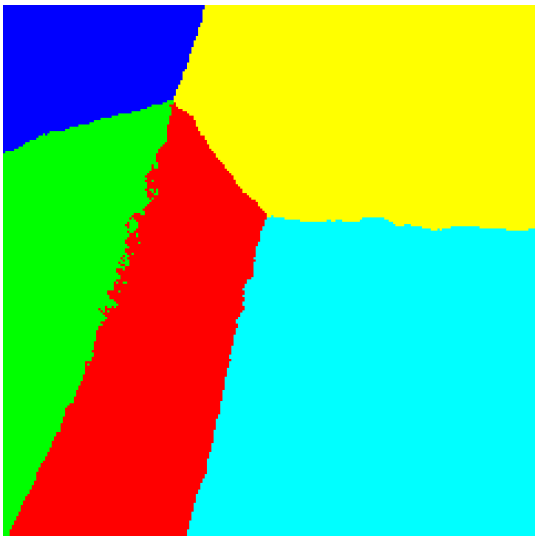


Figure 2: area di classificazione

2.3 Diversi metodi di calcolo della distanza

Nell'algoritmo k-Nearest Neighbors (kNN), il calcolo della distanza tra i punti è fondamentale per determinare i vicini più prossimi. Esistono diversi metodi per calcolare questa distanza, ognuno dei quali può influenzare significativamente le prestazioni del modello a seconda del tipo di dati trattati. Di seguito, presentiamo i metodi più comunemente impiegati:

- **Distanza Euclidea**: È la più comunemente utilizzata, definita come la radice quadrata della somma dei quadrati delle differenze delle coordinate corrispondenti dei punti.

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

  dove $\mathbf{p}$ e $\mathbf{q}$ sono vettori di punti in uno spazio n-dimensionale.

- **Distanza di Manhattan**: Questa metrica somma le differenze assolute delle coordinate.

$$d(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{n} |p_i - q_i|$$

  È particolarmente utile in ambienti urbani o in griglie dove si muove solo lungo gli assi ortogonali.

- **Distanza di Minkowski**: Generalizzazione delle distanze Euclidea e di Manhattan, è definita come:

$$d(\mathbf{p}, \mathbf{q}) = \left(\sum_{i=1}^{n} |p_i - q_i|^r \right)^{\frac{1}{r}}$$

  dove r è un parametro che determina il tipo di distanza: $r = 2$ corrisponde alla distanza Euclidea, mentre $r = 1$ corrisponde alla distanza di Manhattan.

- **Distanza di Chebyshev**: È utile quando il movimento può avvenire in qualsiasi direzione, e si definisce come la massima differenza tra le coordinate di due punti.

$$d(\mathbf{p}, \mathbf{q}) = \max_i |p_i - q_i|$$

- **Distanza di Hamming**: Misura il numero di bit che devono essere cambiati per convertire una stringa in un'altra e viene spesso usata nel contesto di dati categorici.

$$d(\mathbf{p}, \mathbf{q}) = \text{numero di posizioni in cui } p_i \neq q_i$$

La scelta della metrica di distanza dipende dalla natura dei dati e dall'obiettivo specifico dell'analisi. Nel nostro caso abbiamo scelto la distanza Euclidea

3 Implementazione in Pthreads

Nel processare un'immagine I di dimensioni H × W mediante n thread, l'immagine viene divisa in $\frac{\mathrm{H}}{n}$ segmenti verticali. Ogni thread si occupa autonomamente della classificazione del proprio segmento dell'immagine. Per ogni quadratino q, ogni thread crea altri thread per calcolare la distanza di q rispetto a ogni altro punto nell'immagine. Infine, dato l'array delle distanze $\mathbf{D}$, viene applicato un sorting parziale secondo il k-Bubble Sort.

Algorithm 2 k-Bubble Sort

Require: Array $\mathbf{D}$ di dimensione n, dove ogni elemento $\mathbf{D}[i]$ è una struttura contenente una distanza e una classe. Intero k.
Ensure: I primi k elementi di $\mathbf{D}$ sono ordinati in base alla distanza.

```
for i = 0 to k − 1 do
  for j = i + 1 to n − 1 do
    if D[i].distance > D[j].distance then
      Scambia D[i] e D[j]
    end if
  end for
end for
=0
```

Dopo la classificazione, i thread si sincronizzano tramite la funzione `join()`, garantendo l'integrità dei dati elaborati. Questa metodologia non solo accelera l'analisi delle immagini, ma ottimizza anche l'uso delle risorse hardware, essenziale in applicazioni di visione artificiale.

4 Implementazione in Cuda

Nel processare un'immagine I di dimensioni $H \times W$, definiamo un *tile* di grandezza T. Per gestire efficacemente il carico di lavoro, il dominio dell'immagine viene suddiviso in blocchi CUDA. Il numero di blocchi CUDA, nb, è calcolato come segue:

$$\text{nb} = \left(\left\lceil \frac{H}{T} \right\rceil, \left\lceil \frac{W}{T} \right\rceil \right),$$

dove $\lceil \cdot \rceil$ denota l'arrotondamento all'intero superiore, assicurando che l'intera immagine sia coperta anche se le dimensioni non sono multipli esatti di T.

Ciascun blocco CUDA è suddiviso in thread, ognuno dei quali processa un *sub-tile* di dimensioni definite da una variabile chiamata SIDE. Questa variabile rappresenta la dimensione del quadrato di pixel che ciascun thread elaborerà individualmente all'interno del blocco.

L'indice del pixel (x, y) che ogni thread deve processare è calcolato con le seguenti formule:

int tx = threadIdx.x;
int ty = threadIdx.y;
int bx = blockIdx.x;
int by = blockIdx.y;
int $x = bx \times T + tx \times$ SIDE;
int $y = by \times T + ty \times$ SIDE;

Per massimizzare l'utilizzo della GPU, abbiamo implementato un algoritmo personalizzato basato sull'insertion sort per trovare le distanze minori in modo efficiente. Questo algoritmo mantiene ordinato un array di distanze vicine, aggiornandolo e riordinandolo ogni volta che viene trovata una distanza minore.

Algorithm 3 Insertion sort per CUDA

Require: Array $\mathbf{P}$ of num_points points, new point $\mathbf{p}$, integer k
Ensure: The labels of the k nearest neighbors to $\mathbf{p}$
Initialize arrays $\mathbf{D}$ and $\mathbf{L}$ of size k
for $i = 0$ to $k - 1$ **do**
  $\mathbf{D}[i] \leftarrow$ euclidean_distance($\mathbf{P}[i], \mathbf{p}$)
  $\mathbf{L}[i] \leftarrow \mathbf{P}[i].label$
end for
insertion_sort($\mathbf{D}$, $\mathbf{L}$, k)
for $i = k$ to $num\_points - 1$ **do**
  $dist \leftarrow$ euclidean_distance($\mathbf{P}[i], \mathbf{p}$)
  if $dist < \mathbf{D}[k-1]$ **then**
    $\mathbf{D}[k-1] \leftarrow dist$
    $\mathbf{L}[k-1] \leftarrow \mathbf{P}[i].label$
    insertion_sort($\mathbf{D}$, $\mathbf{L}$, k)
  end if
end for
=0

5 Performance

In questa sezione, presentiamo i risultati ottenuti dai nostri programmi utilizzando dataset di diverse dimensioni. I test sono stati eseguiti su un computer dotato di CPU Intel i5-13400F e GPU NVIDIA RTX 3080 con 10GB di memoria. Abbiamo calcolato lo speedup, definito come il rapporto tra il tempo di esecuzione della versione seriale e il tempo di esecuzione della versione parallela di ciascun programma. La formula per lo speedup è data da:

$$\text{Speedup} = \frac{T_s}{T_p} \tag{1}$$

dove T_s rappresenta il tempo di esecuzione della versione seriale e T_p rappresenta il tempo di esecuzione della versione parallela. Lo speedup fornisce una misura dell'efficienza delle implementazioni parallele rispetto alla loro controparte seriale.

Numero di punti	**Seriale**	**Pthread con 8 thread**	**Speedup Pthread**	**Cuda**	**Speedup Cuda**
10k	18.76s	6.18s	3	0.44s	42
50k	1min30s	12s	7.5	1.6s	56
100k	3min3s	22,25s	8,22	3s	61

Table 1: Performance Comparison

6 Conclusione

L'implementazione con Pthreads ha mostrato un notevole speedup, specialmente con l'aumento del numero di punti nel dataset. Tuttavia, è la versione con CUDA che ha offerto i risultati più impressionanti, grazie alla sua capacità di sfruttare al massimo le risorse della GPU per parallelizzare i calcoli. Con un speedup fino a 61 volte superiore rispetto alla versione seriale, l'implementazione CUDA ha reso possibile la classificazione di dataset molto grandi in tempi ridotti.

Questi risultati evidenziano l'importanza della parallelizzazione nell'ottimizzazione degli algoritmi di machine learning, in particolare quando si lavora con dataset di grandi dimensioni. L'uso di tecniche di parallelizzazione non solo accelera i tempi di elaborazione, ma rende anche possibile l'analisi di dati che sarebbero altrimenti impraticabili con approcci seriali.