Department of Computer Science

Parallel Implementation of K-Nearest Neighbors

Project in Embedded and Multicore Systems Programming

Professor:
Salvatore Pontarelli

Students:
Bustamante Gabriel Alonso
Lin Can
Olivieri Alessio

University of Rome La Sapienza

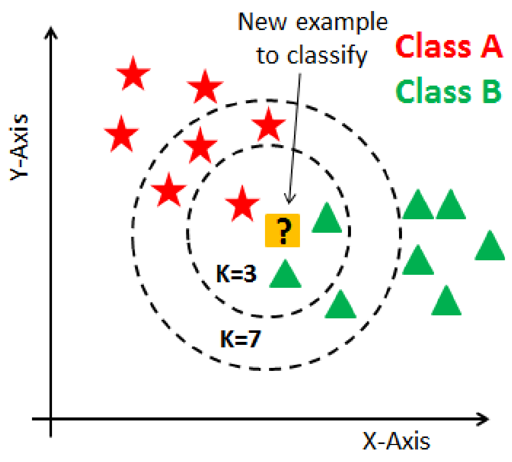
1 Introduction

This project explores three implementations of a K-nearest neighbors in C, a fundamental algorithm in non-parametric supervised learning that uses the distance between data points to determine the class of a new sample. The implementations analyzed include: a traditional serial version, a parallelized version with Pthreads, and a parallelized version with CUDA.

1.1 K-nearest neighbors

The K-nearest neighbors (K-NN) is a non-parametric supervised learning algorithm used for classification and regression. It works by identifying the K closest data points (the "neighbors") to a new sample, based on a predefined distance, usually the Euclidean distance. The class of the new sample is determined by the majority class of its nearest neighbors. K-NN is simple to implement and makes no particular assumptions about the data, making it versatile for many applications. However, it can be computationally intensive for large datasets, as it requires the calculation of the distance between each pair of points.

2 The K-nearest neighbors Algorithm



2.1 Pseudocode

Algorithm 1 K-nearest neighbors classification algorithm

```
0: procedure kNN
0:     Input:
0:     $D$: a set of training samples $\{(x_1, y_1), \ldots, (x_n, y_n)\}$
0:     $k$: the number of nearest neighbors
0:     $d(x, y)$: a distance metric
0:     $x$: a test sample
0:     Output: The predicted class for the test sample $x$
0:
0:     for each training sample $(x_i, y_i)$ in $D$ do
0:         Calculate $d(x, x_i)$, the distance between $x$ and $x_i$
0:     end for
0:     Let $N \subseteq D$ be the set of training samples with the $k$ smallest distances $d(x, x_i)$
0:     Return the majority label of the samples in $N$
0: end procedure=0
```

2.2 Objectives

Our program uses a dataset D composed of n points, each defined by two coordinates (x, y). Given an image I of dimensions $H \times W$, we divided I into a series of small squares q, each of size $\text{SIDE} \times \text{SIDE}$. Each square q was classified based on its central pixel using the k-nearest neighbors (kNN) algorithm. Finally, we colored the image in different colors depending on the class determined for each square's center, thus obtaining the area of belonging for each class.

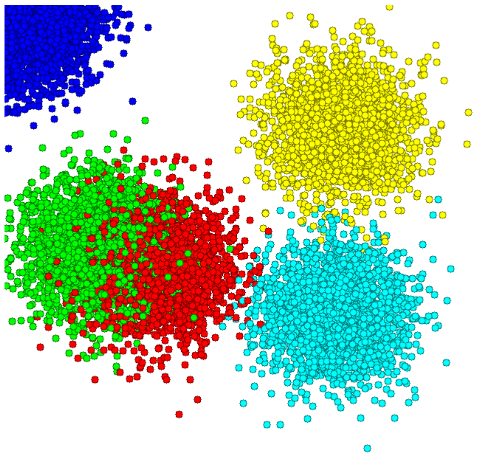


Figure 1: point placement

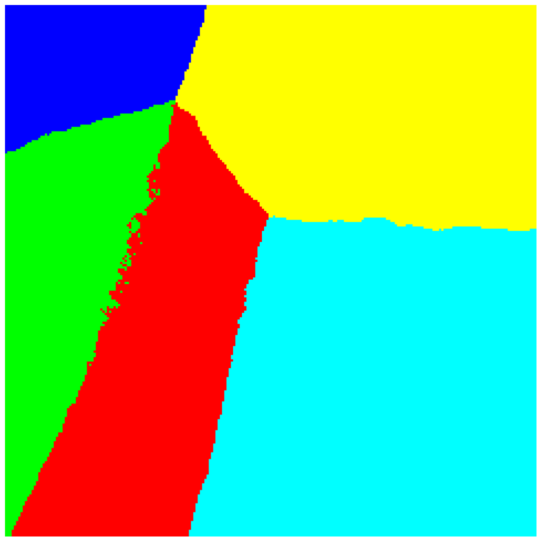


Figure 2: classification area

2.3 Different methods of distance calculation

In the k-Nearest Neighbors (kNN) algorithm, calculating the distance between points is fundamental to determining the nearest neighbors. There are various methods to compute this distance, each of which can significantly influence the model's performance depending on the type of data being processed. Below, we present the most commonly employed methods:

- **Euclidean Distance**: It is the most commonly used, defined as the square root of the sum of the squared differences of the corresponding coordinates of the points.

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

  where $\mathbf{p}$ and $\mathbf{q}$ are vectors of points in an n-dimensional space.

- **Manhattan Distance**: This metric sums the absolute differences of the coordinates.

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

  It is particularly useful in urban environments or grids where movement occurs only along orthogonal axes.

- **Minkowski Distance**: Generalization of Euclidean and Manhattan distances, defined as:

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

  where r is a parameter that determines the type of distance: $r = 2$ corresponds to the Euclidean distance, while $r = 1$ corresponds to the Manhattan distance.

- **Chebyshev Distance**: Useful when movement can occur in any direction, defined as the maximum difference between the coordinates of two points.
$$d(\mathbf{p}, \mathbf{q}) = \max_i |p_i - q_i|$$
- **Hamming Distance**: Measures the number of bits that must be changed to convert one string into another and is often used in the context of categorical data.
$$d(\mathbf{p}, \mathbf{q}) = \text{number of positions where } p_i \neq q_i$$

The choice of distance metric depends on the nature of the data and the specific analysis objective. In our case, we chose the Euclidean distance.

3 Implementation in Pthreads

When processing an image I of dimensions $H \times W$ using n threads, the image is divided into $\frac{\mathrm{H}}{n}$ vertical segments. Each thread independently handles the classification of its image segment. For each small square q, each thread creates other threads to calculate the distance of q from every other point in the image. Finally, given the distance array $\mathbf{D}$, a partial sorting according to the k-Bubble Sort is applied.

Algorithm 2 k-Bubble Sort

```
Require: Array $\mathbf{D}$ of size $n$, where each element $\mathbf{D}[i]$ is a structure containing a distance and a class.
    Integer $k$.
Ensure: The first $k$ elements of $\mathbf{D}$ are sorted by distance.
1: for $i = 0$ to $k - 1$ do
2:   for $j = i + 1$ to $n - 1$ do
3:     if $\mathbf{D}[i]$.distance > $\mathbf{D}[j]$.distance then
4:       Swap $\mathbf{D}[i]$ and $\mathbf{D}[j]$
5:     end if
6:   end for
7: end for
   =0
```

After classification, the threads synchronize via the `join()` function, ensuring the integrity of the processed data. This methodology not only speeds up image analysis but also optimizes hardware resource usage, essential in computer vision applications.

4 Implementation in CUDA

When processing an image I of dimensions $H \times W$, we define a tile size T. To effectively manage the workload, the image domain is divided into CUDA blocks. The number of CUDA blocks, nb, is calculated as follows:
$$\text{nb} = \left(\left\lceil \frac{H}{T} \right\rceil, \left\lceil \frac{W}{T} \right\rceil \right),$$
where $\lceil \cdot \rceil$ denotes the ceiling function, ensuring full coverage of the image even if the dimensions are not exact multiples of T.

Each CUDA block is divided into threads, each of which processes a sub-tile of dimensions defined by a variable called SIDE. This variable represents the size of the square of pixels each thread will process individually within the block.

The index of the pixel (x, y) that each thread must process is calculated with the following formulas:

$$\begin{aligned}
\text{int } tx &= \text{threadIdx.x;} \\
\text{int } ty &= \text{threadIdx.y;} \\
\text{int } bx &= \text{blockIdx.x;} \\
\text{int } by &= \text{blockIdx.y;} \\
\text{int } x &= bx \times T + tx \times \text{SIDE;} \\
\text{int } y &= by \times T + ty \times \text{SIDE;}
\end{aligned}$$

To maximize GPU utilization, we implemented a custom algorithm based on insertion sort to efficiently find the nearest distances. This algorithm maintains an ordered array of nearby distances, updating and reordering it whenever a shorter distance is found.

Algorithm 3 Insertion sort for CUDA

```
Require: Array $\mathbf{P}$ of $num\_points$ points, new point $\mathbf{p}$, integer $k$
Ensure: The labels of the $k$ nearest neighbors to $\mathbf{p}$
 1: Initialize arrays $\mathbf{D}$ and $\mathbf{L}$ of size $k$
 2: for $i = 0$ to $k - 1$ do
 3:     $\mathbf{D}[i] \leftarrow$ euclidean_distance($\mathbf{P}[i], \mathbf{p}$)
 4:     $\mathbf{L}[i] \leftarrow \mathbf{P}[i].label$
 5: end for
 6: insertion_sort($\mathbf{D}$, $\mathbf{L}$, $k$)
 7: for $i = k$ to $num\_points - 1$ do
 8:     $dist \leftarrow$ euclidean_distance($\mathbf{P}[i], \mathbf{p}$)
 9:     if $dist < \mathbf{D}[k-1]$ then
10:         $\mathbf{D}[k-1] \leftarrow dist$
11:         $\mathbf{L}[k-1] \leftarrow \mathbf{P}[i].label$
12:         insertion_sort($\mathbf{D}$, $\mathbf{L}$, $k$)
13:     end if
14: end for
    =0
```

5 Performance

In this section, we present the results obtained from our programs using datasets of different sizes. Tests were conducted on a computer equipped with an Intel i5-13400F CPU and an NVIDIA RTX 3080 GPU with 10GB of memory. We calculated the speedup, defined as the ratio of the execution time of the serial version to the execution time of the parallel version of each program. The formula for speedup is given by:

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

where T_s represents the execution time of the serial version and T_p represents the execution time of the parallel version. The speedup provides a measure of the efficiency of the parallel implementations compared to their serial counterpart.

Number of points	Serial	Pthread 8 threads	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 Conclusion

The Pthreads implementation showed a significant speedup, especially with the increase in the number of points in the dataset. However, it is the CUDA implementation that offered the most impressive results, thanks to its ability to fully leverage the GPU resources to parallelize computations. With a speedup up to 61 times greater than the serial version, the CUDA implementation enabled the classification of very large datasets in reduced times.

These results highlight the importance of parallelization in optimizing machine learning algorithms, particularly when working with large datasets. The use of parallelization techniques not only speeds up processing times but also makes it possible to analyze data that would otherwise be impractical with serial approaches.