Implementation of k-Nearest Neighbors algorithm using CUDA

author: Szymon Nachlik

FACULTY OF PHYSICS AND APPLIED COMPUTER SCIENCE
AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

Introduction

k-Nearest Neighbors (*kNN*)[1] algorithm is a popular machine learning algorithm used for classification and regression tasks. The basic idea behind *kNN* is to identify the *k* nearest data points to a given test sample and make predictions based on the majority class of the nearest neighbors. Implementing the algorithm on CPU can be time-consuming and limiting in terms of processing power, especially when dealing with large datasets. In this report, I will present the implementation of *kNN* algorithm using CUDA. The main objective of this project was to evaluate the performance and efficiency of the *kNN* algorithm using CUDA compared to its traditional CPU implementation. The report will describe the implementation details, results and comparison of the two implementations.

1) *kNN algorithm details*

Implementation

In order to measure the performance of the algorithm running on the GPU, I also implemented the CPU version. Both CPU and GPU algorithm implementations share common steps.

Theoretical description of how the implemented algorithms work:

0) loading data from csv files (assigns to a two-dimensional array)
1) calculation of the euclidean distances of the sample from all training samples
2) sorting the distances table while preserving the indices
3) selecting the k-smallest values from the sorted array
4) replacing the distance with the corresponding targets (thanks to preserving the indexes)
5) finding the mode of the resulting k-element array
6) giving predicted class based on the mode

1) Distance calculation:

CPU algorithm:

Calculating euclidean distance with CPU is iterating through all training samples. For each sample, the algorithm goes through all dimensions by calculating the distance in that dimension and adding it to the total *(nested loop).*

GPU algorithm:

In the GPU version, each training sample is assigned a separate thread. Then, similarly to the CPU version, the algorithm iterates over each dimension calculating the total distance.

Both algorithms store the results in an array. At this point, the indexing is preserved, and the index of the training sample is identical to the corresponding distance in the resulting array.

2) Sorting:

The distance array must be sorted to find the k-smallest values. The issue is that simply sorting the values is not enough - we would lose the information about the indices.

I utilized the standard library and created a map that holds information about the distance and its corresponding target. Then map is sorted.

Now, based on the hyperparameter k, we can choose the corresponding number of smallest values with the targets.

3) Finding mode:

The final step is to find the mode of the resulting array of targets. The mode is also the prediction of the algorithm and the assignment of the sample to that class.

Performance analysis

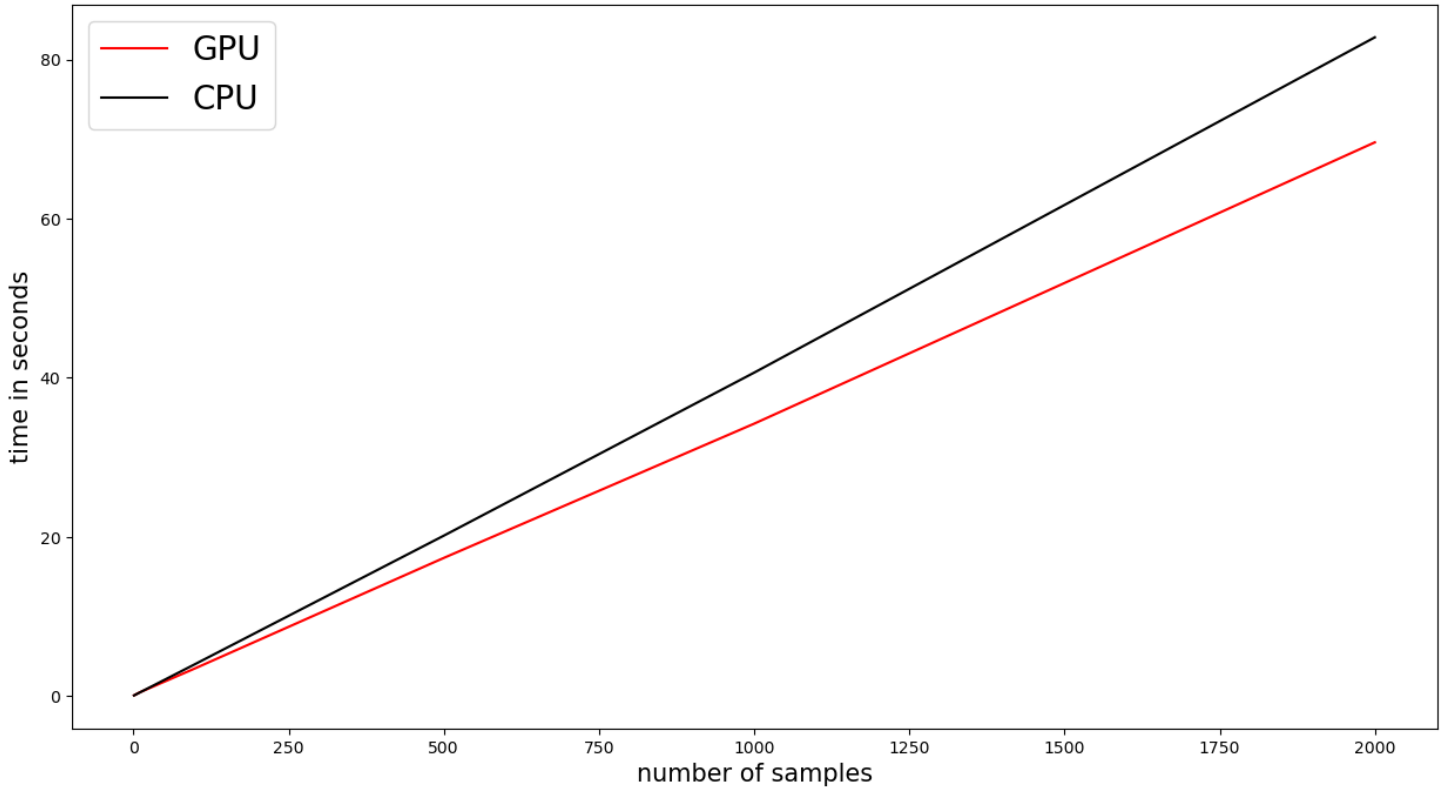
In order to examine the performance differences of the implemented *kNN* algorithms, I conducted the following comparisons:

- duration of predictions calculation depending on the number of samples
- duration of prediction calculation depending on the number of dimensions

The data used for the calculations comes from the *MNIST - Dataset of Handwritten Digits*.

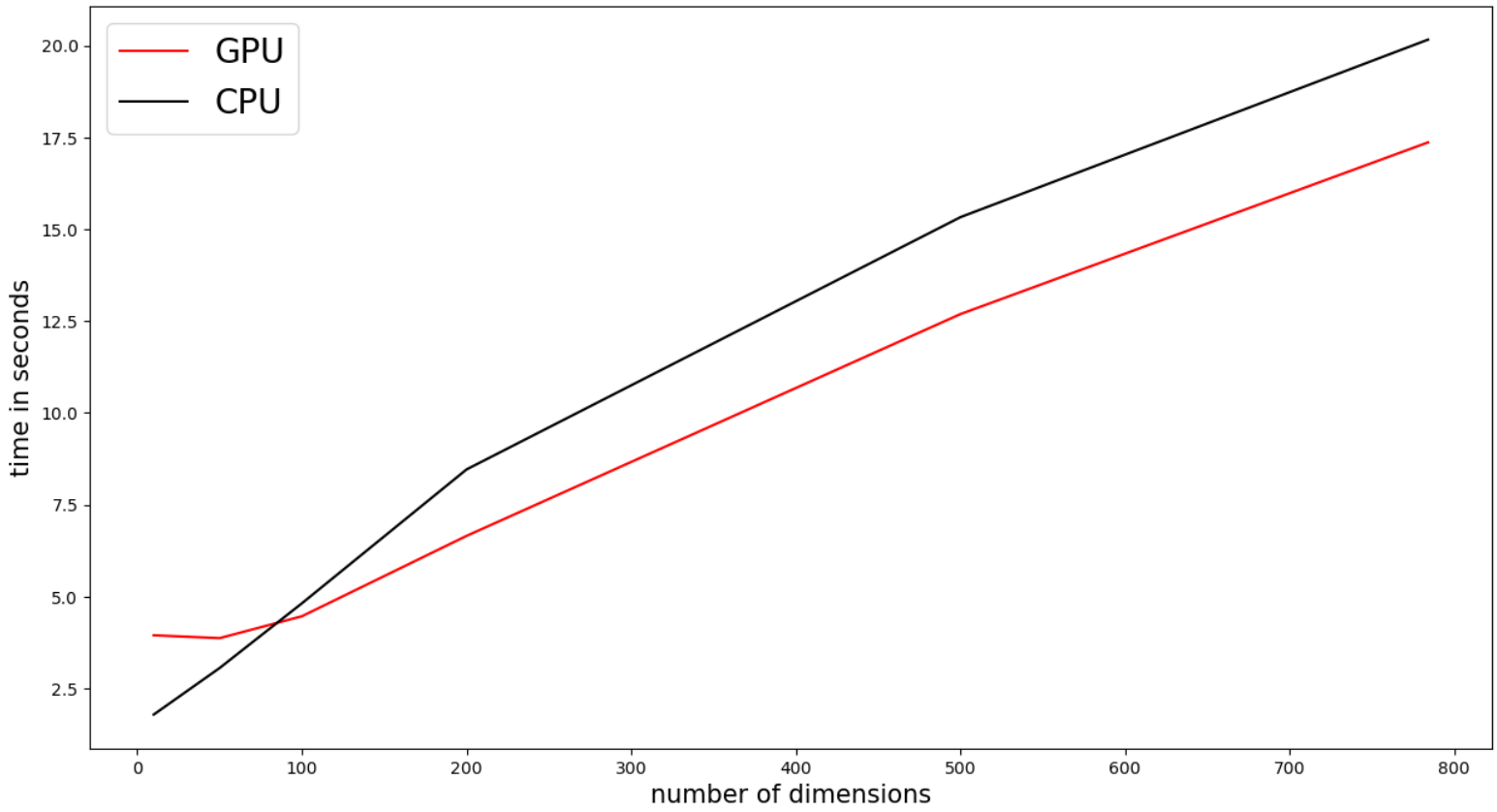
1) Duration of predictions calculation depending on the number of samples:

I carried out 6 measurements for different amounts of samples to be predicted. Calculation duration results for CPU and GPU algorithm:



2) Duration of predictions calculation depending on the number of dimensions:

I carried out 6 measurements for different amounts of dimensions in dataset for 500 samples. Maximum dimensions in dataset are 784. Calculation duration results for CPU and GPU algorithm:



Conclusions and summary

For a large number of samples, the efficiency of the *kNN* algorithm on the GPU is higher than on the CPU, but it is not a big difference, only about 15%.

For a small number of dimensions, CPU algorithm is more efficient than the GPU, but by increasing the number of dimensions, the kernel becomes much more efficient. For several thousand dimensions, this difference will be very large.

The kernel implementation consists in minimizing the time of calculating the distance of the sample from the test set. Unfortunately, sorting is done on the CPU and the performance is not as high as sorting on the GPU could be. Due to the need to synchronize the GPU with the device, the running time of *kNN* to CPU for small dimensions becomes more profitable. However, for large dimensions, the calculation of the length takes a considerable amount of time, and this is where the kernel's advantage becomes apparent.