CUDA implementation of k-Nearest Neighbors
High performance data and graph analytics

D. Russica, L. Tisato
Dipartimento di Elettronica, Informatica e Bioingegneria
Politecnico di Milano, Milan, Italy
{danielekota.russica, leonardo.tisato}@mail.polimi.it

December 26, 2023

Abstract

The k-Nearest Neighbors (k-NN) algorithm is a versatile and intuitive technique used for both classification and regression tasks in machine learning. By considering the proximity of data points in feature space, k-NN makes predictions based on the majority class or average value of the k-nearest neighbors. Despite its simplicity, k-NN can be computationally demanding, especially with large datasets or high-dimensional feature spaces. Our task was to accelerate a sequential version of the k-NN developing a parallel version of it using CUDA language without external libraries. The speedup we were able to obtain using a colab T4 GPU ranged from x23 to x172.

1 Introduction

In this challenge, our objective involved computing the k-nearest neighbors (k-NN) for each query within a dataset of reference points, employing the cosine similarity as the designated metric. Our approach to address this task comprised the decomposition of the k-NN problem into two distinct subproblems: the computation of distances and the subsequent selection of the smallest-k neighbors.

Throughout the course of this project, we decided not to use unified memory for device memory management.

2 Cosine similarity calculation

This subsection takes the reference points matrix and the query points matrix as input and provides the distances matrix.

The cosine similarity is a metric used to measure the similarity between two non-zero vectors in an inner product space. It is often used to measure document similarity in text analysis and it can be calculated as:

$$CosineSimilarity(A, B) = \frac{< \mathbf{A}, \mathbf{B} >}{\|\mathbf{A}\| \cdot \|\mathbf{B}\|} = \frac{\sum_i^n (A_i \cdot B_i)}{\sqrt{\sum_i^n A_i^2} \cdot \sqrt{\sum_i^n B_i^2}}$$

Where $\mathbf{A}$ and $\mathbf{B}$ are a query point and a reference point in a n dimensional feature space.

Given a matrix to store the reference points (size: $dim\ x\ ref\_nb$) and a matrix to store the query points (size: $dim\ x\ query\_nb$). Where ref_nb and $query\_nb$ are respectively the number of reference points and query points and dim is the dimension of the space.

Our idea was to create ref_nb blocks with dim threads, calculate the cosine similarity using reduction and repeat this process $query\_nb$ times. Therefore each thread would handle a single dimension and each block would handle a reference point.

Listing 1: Cosine similarity kernel

```
for query in query_nb sequentially:
        for ref in ref_nb in parallel:
                calculate_cosine_similarity(ref, query)
```

And the results were stored in a distances matrix on the device (size: $ref\_nb \; x \; query\_nb$).

Our cosine similarity kernel performs the first reduction step when loading the shared memory, in order to decrease idle threads and "fit" more than 1024 dimensions in our block.
Furthermore, to handle cases where the feature space dimension is not a power of two, we decided to pad (if needed) the shared memory array with zeros, up to the closest power of two.

3 K selection

This subsection addresses the selection of the k smallest elements for each query from the distances matrix.

We identified three primary methods for selecting the k-nearest neighbors as: complete sorting, partial sorting, and selection. Due to the relatively small size of the k values compared to the number of reference points, we excluded the complete sorting option.

3.1 Selection approach

In one of our solutions, we iteratively selected the k smallest values from the array of distances for each query. To achieve this, we devised a two-step reduction kernel process. The first kernel extracted a set of candidates, each representing the smallest distance within a subset of the array, while the second kernel determined the absolute minimum value among these candidates. The selected minimum value was then inserted into the output array, and a "1" was marked in its corresponding position in the distance matrix to prevent redundant selections.

We divided this operation into two kernels since each array had more than the maximum number of thread per block.

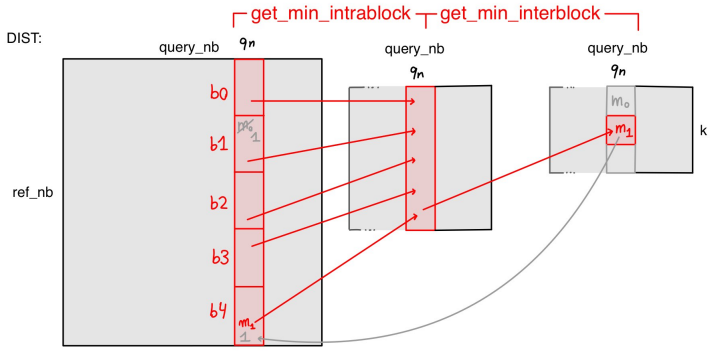

Figure 1: Schematic diagram of the k selection approach

3.2 Partial sorting approach

Alternatively, we implemented a partial sorting approach. In this case, each thread in our kernel performed an insertion sort on the columns of the distance matrix, limited to the smallest-k elements.

3.3 Other approaches

In our pursuit of optimizing the selection process, we wanted to explore alternative approaches leveraging the inherent characteristics of cosine distances, confined within the range of 0 to 1. One such method involved computing the average distance between two points in the space within our specified

parameter sets and then employ stream compaction to reduce the length of each distance array by eliminating distances that exceeded a certain threshold. To implement this, we considered modifying the data structure of the distances matrix. The key challenge lay in determining an appropriate cutoff value that would exclude as many reference points as possible while maintaining perfect accuracy in the final k-NN results. Unfortunately, we did not have the time needed to explore this solution effectively.

4 Handling large datasets

Given the subdivision of the original problem into two distinct subproblems, all our solutions necessitate a matrix of distances. These distances are computed in the first phase and subsequently serve as input for the selection part. In scenarios involving extensive datasets with a high number of reference and query points, the distance matrix may exceed the capacity of device memory.

To address this limitation and ensure the scalability of our solution, particularly in datasets with many points, we opted to partition the query points into batches. By doing so, we could efficiently compute distances and identify the k-nearest neighbors for each batch, mitigating the risk of memory constraints.

5 Results

We received four sets of parameters to test our solutions. The table below presents the details of these parameters.

	ref_nb	query_nb	dim	k
Parameters 0	4096	1024	64	16
Parameters 1	16384	4096	128	100
Parameters 2	163840	40960	128	16
Parameters 3	16384	4096	1280	16

Table 1: Given set of parameters

The initial set, Parameters 0, served as a small-scale dataset employed for programming and fine-tuning our solution. The subsequent three parameter sets were deliberately designed to test our solution under varied conditions. Parameters 1 featured a large k value, Parameters 2 involved a substantial number of reference and query points, while Parameters 3 had a sizable feature space.

The following table illustrates the time taken and corresponding speedup achieved through the implementation of our proposed methods, over a 50 iteration average for the GPU versions.

Method	Param0	Param1	Param2*	Param3
CPU	4.7857s	101.10642s	17920.13s	2795.54s
GPU - Selection approach	0.1939s	4.23653s	217.6s	18.6475s
Speedup vs CPU	x24	x23	x82	x150
GPU - Partial sorting approach	0.05845s	1.2350s	119s	16.2107s
Speedup vs CPU	x82	x81	x150	x172

Table 2: Time and speed up for the methods we implemented, against all the set of parameters

*: We estimated the running time for the second set of parameters to be over five hours for the CPU version, since the resources on colab are limited, we decided to just estimate the value both for the CPU solution as well as for the GPU solution (since the result must be compared with the CPU version's ground truth).

We anticipated that the selection approach would yield superior results, assuming that with a small value of k, it would be more efficient to directly pick the smallest elements rather than partially sorting the array. However, as the results indicate, the partial sorting approach consistently outperformed the selection approach.

To assess the behavior of our solutions under varying values of k, we employed Parameters 1 with a range of k values for testing.

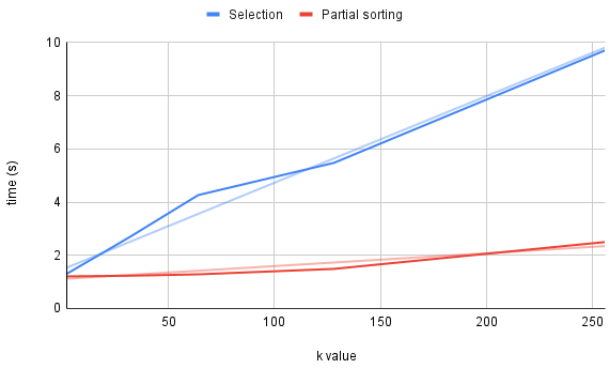

Figure 2: Time(s) vs k value for selection and partial sorting approach

As anticipated, the performance trend revealed that the larger the value of k, the more pronounced the superiority of the partial sorting solution over the selection approach. But we suspect that there might be a chance that a more efficient version of our selection approach could give better results for small k values.

6 Possible improvements

We are aware that our solutions could be improved in various ways.

First we tried to create a stream version of the second subproblem, but we did not see any improvements in the performance so we decided not to use streams. However a stream version that would divide the queries in batches and asynchronously copy to device the queries matrix, perform cosine distance calculation, select and asynchronously copy back the results for each batch independently may improve performances.

Moreover, as mentioned above, our selection approach could be optimized, for example adding the first reduction operation while loading the shared memory or trying to combine the two kernels into one, limiting the number of kernel launches.

7 Conclusion

We are thankful for the challenge as it was a great chance to develop new skills. Considering that it was our first time using CUDA we are satisfied with the results. Nonetheless, we acknowledge that there is still much to learn, and we are confident that with further exploration and refinement, our results can be significantly enhanced.