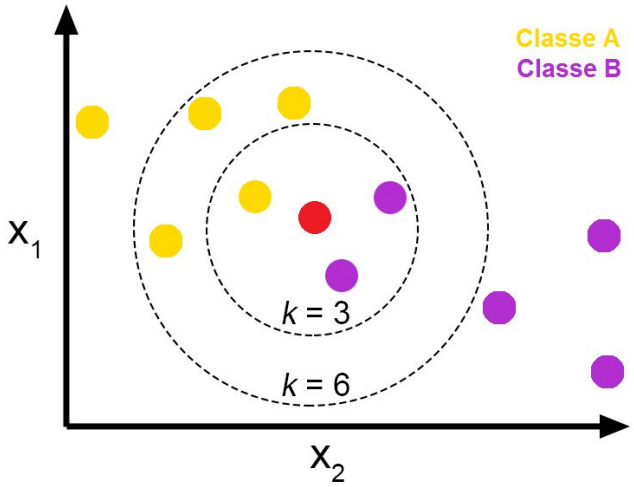
CUDA implementation of k-Nearest Neighbors

High-Performance Data and Graph Analytics Contest

D. Russica, L. Tisato
{danielekota.russica, leonardo.tisato}@mail.polimi.it

1. Introduction:

- The k-Nearest Neighbors (k-NN) algorithm is a versatile technique used for both classification and regression tasks in machine learning.
- k-NN predicts based on majority class or average of k-nearest neighbors, considering feature space proximity.



1. Introduction

- Despite its simplicity, k-NN can be computationally demanding.
- Our task was to accelerate a sequential version of the k-NN developing a parallel version of it using CUDA language without external libraries.

- Specifically, we were given a dataset of reference and query points, and our task was to compute the k-NN using cosine similarity as distance metric.

- The speedup we were able to obtain using a colab T4 GPU ranged from x23 to x172.

1.1 Our solutions’ design

- We decided to decompose the k-NN problem into 2 distinct subproblems:
  - distances computation.
  - selection of the smallest-k neighbors.
- We decided not to use unified memory.

2. Cosine similarity calculation

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 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 **A** and **B** are a query point and a reference point
- in a **n** dimensional feature space.

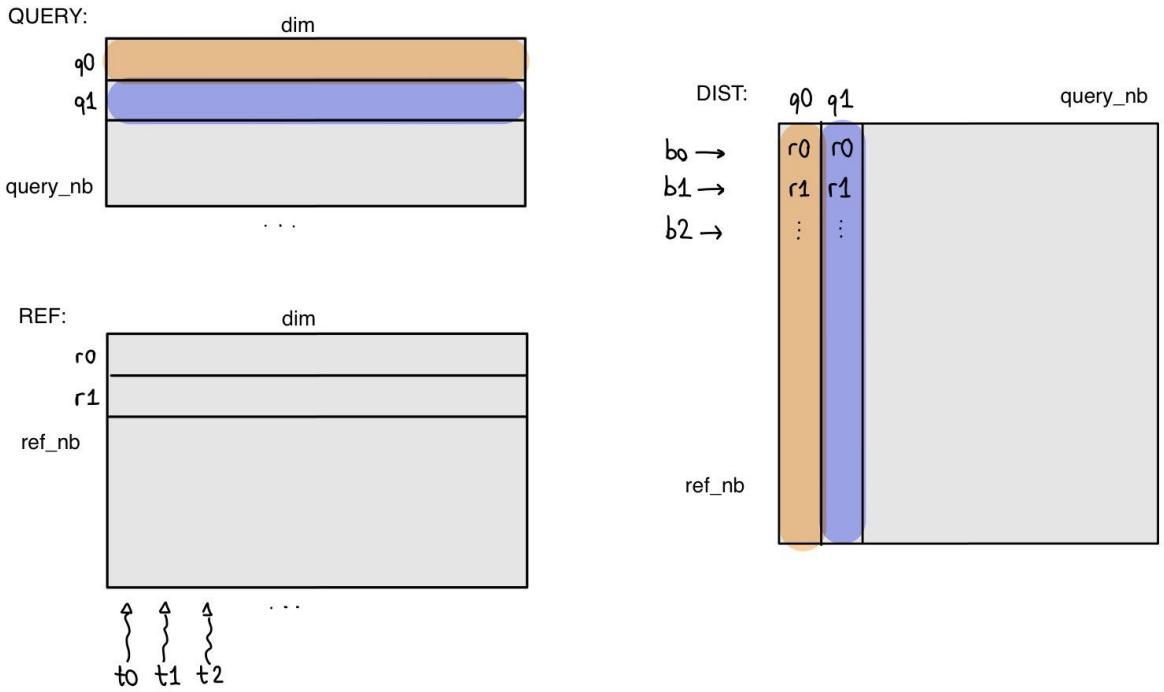
2. Cosine similarity calculation

ref_nb: # of reference points
query_nb: # of query points
dim: dimension of feature space

- Given a matrix to store:
  - the reference points (size: dim x ref_nb)
  - the query points (size: dim x query_nb).
  - and the calculated distances (size: ref_nb x query_nb)

- Our idea was to create '*ref nb*' blocks with '*dim*' threads and calculate the cosine similarity using reduction and repeat this process '*query nb*' times.
- The shared memory consisted in 3 array each to store:
  - $(A_i \cdot B_i)$, A_i^2, B_i^2 for each dimension

2.1 Representation of our method for cosine similarity calculation



2.2 Optimizations for our cosine similarity kernel

- Our cosine similarity kernel performs the first reduction step when loading the shared memory in order to:
  - decrease idle threads.
  - “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

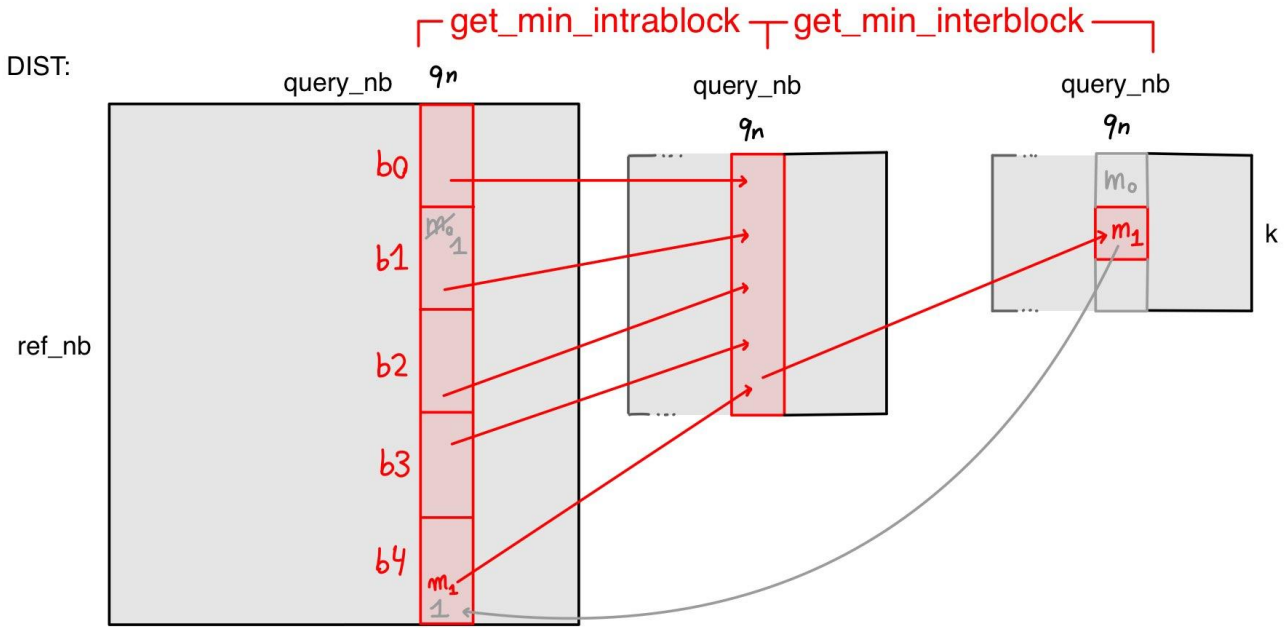
3. K selection

- This subsection addresses the transformation of the distances matrix into the final k-nearest neighbor (k-NN) distances and indexes.
- We identified three primary methods for selecting the k-nearest neighbors as:
    1. Complete sorting
    2. Selection
    3. Partial sorting
- Due to the relatively small size of the k values compared to the number of reference points, we excluded the complete sorting option.

3.2 Selection approach

- In one of our solutions, we iteratively selected the k smallest values from the array of distances for each query, on the GPU.
- 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.
  - the second kernel determined the absolute minimum value among these candidates.
- Then the minimum would be added in the output array and a “1” would be placed in the correspondent cell on the device memory to avoid redundant selections.

3.2 Representation of our selection approach



3.3 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.

3.4 Other approaches

- 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.
- Then employ stream compaction to reduce the length of each distance array by eliminating distances that exceeded a certain threshold.

3.4 Other approaches

- However this approach would have forced us to rethink our data structure from scratch.
- Furthermore finding the perfect threshold value, that exclude the most candidates as possible and keeps the accuracy at 100% is not easy, and this value would change as the starting parameters change.

- 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.
- In scenarios involving extensive datasets, the distance matrix may exceed the capacity of device memory.
- To ensure the scalability of our solution, we opted to partition the query points into batches, if the distance matrix would not fit into device memory.
- By doing so, we could compute distances and identify the k-nearest neighbors for each batch.

5. Results

- The set of parameters we received to evaluate our solution were:

	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

To test our solution

High k value

high ref_nb & query_nb

high dimensionality

5. Results

- Here are the results obtained using the selection and partial sorting approaches:

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

As k value increase, it is beneficial to sort the array instead of picking the smallest each time.

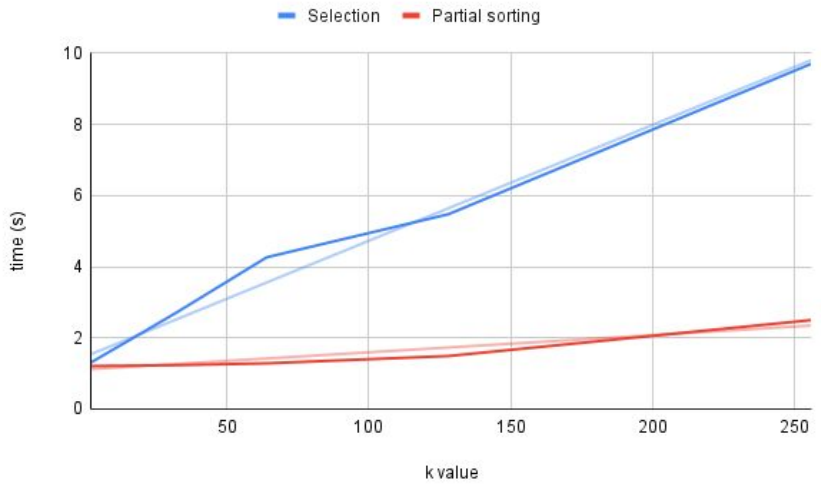
All values are estimates! GT calculation would take up to 5h, therefore result were calculated on a version of the same dataset with less queries

As the dimensionality increase (longer distance calculation time) and k decreases, the performance are almost equal

5.1 Results (selection vs partial sort approach)

- We hoped that our selection approach could outperform the partial sorting approach for small k values, however after testing our solutions against “parameters 1” with a variable number of k we found out that this did not happen.

- This could be due to the fact that our selection kernels were not optimized enough.



6.1 Possible improvements

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

Thank you for your attention

D. Russica, L. Tisato
{danielekota.russica, leonardo.tisato}@mail.polimi.it