Optimised K-Nearest-Neighbors for high dimensional vectors

Weaviate challenge

Petridis Konstantinos

June 7, 2023

0.1 Introduction

There are three implementations discussed, the **naive**, **parallel** using go-routines, and **indexed-parallel** approach for calculating the k-nearest-vectors of a given query, from a pool of high dimensional vectors. The implementations are tested for varying values of n and d, where n is the number of vectors in the pool and d the number of dimensions of each vector.

0.2 Naive/Brute Force k-NN

The naive algorithm for computing k-NN goes as follows: given a query vector v, compute the distance of v from all the other vectors and store them in an array. Finally, sort the array and return the k first elements. Time complexity is $O(nd + n \log n)$ or $O(n(d + \log n))$. For this approach, no optimisations are applied yet.

0.3 Parallel k-NN

For this approach, we leverage the integrated concurrency model provided by Golang, to boost the performance. We notice that in the previous approach, the computation of every distance is independent of the computation of the rest. Therefore we can scatter the computation across multiple Goroutines. The question becomes how many? Well, a first thought would be to spawn one routine per iteration, thus every distance is computed by a single routine. Profiling however indicates that the creation and scheduling of Goroutines takes so much time, that the introduced overhead leads to negative speedup. The followed approach spawns as many goroutines as available CPUs to optimise usage of resources. Each Goroutine updates a part of the total array. There are no data races since there are no overlapping portions of the array that are assigned to different Goroutines. This way we avoid the use of mutexes, which would noticeably harm the performance.

0.4 Indexed-Parallel k-NN

In this approach we perform a pre-process to build an index that will facilitate faster k-NN search. After that, we identify what can be parallelized in the searching process and we implement it using Goroutines.

0.4.1 Index using Vantage Point Tree

The selected index is the **Vantage Point Tree**. It is preferred over **KD Trees** since it has been both theoretically and practically proved that it tends to outperform them in higher dimensions. The space is divided into hyper shperes based on distances between nodes. The division is also optimised to pick the best point to split the space in. The complexity to build this index is $O(n \log n)$.

0.4.2 Search on Vantage Point Tree

The search is implemented in an iterative approach, which both saves stack space and also makes it feasible to parallelize. A queue is utilised to insert the VP-Tree nodes that should be explored in next

iterations, similarly to an iterative Breadth-First-Search. Based on the number of pending nodes in the queue, we decide how many Goroutines to spawn. Up to a certain threshold, we spawn as many Goroutines as there are available nodes, to perform the search in multiple hyper spheres simultaneously. Additionally, we designate a max number of Goroutines that when exceeded, we don't spawn more of them, because the added overhead beats the benefits of the parallelization after that point.

0.5 Other Optimisations

This section presents all optimizations that were applied to the implementation.

- **Store vectors in Row-Major order**: vectors are not stored in a 2D array format, as someone may expect, but they are rather stored in a linear fashion, just like a 1D array. To access it properly we have to use correct indexing. The purpose is: values in 1D arrays are stored sequentially in memory in **Row-Major** order, allowing for efficient caching and locality. Different rows of a 2D matrix on the other hand, are not stored in the same memory block, incurring additional index calculations to determine the location of the elements in memory. Finally, storing elements in **Row-Major** order will make it easy for the compiler to identify places were **SIMD** can be applied, and it will leverage SIMD instructions to boost performance even more. This is mostly the case when looping over elements that are stored adjacently in memory.
- **Use sufficient types to minimise memory usage**: The initial implementation used **int** type. It required more than 35GB of RAM to build the index, when dealing with 1.000.000 vectors of 256 dimensions. Changing that to use int8, combined with using pointers to avoid copying objects in each function call, dropped the RAM usage to only 7GB RAM for the same number of parameters. It should be underlined at this point that this amount of memory is only used to build the index, not to perform the k-NN search.
- **Use of pointers to avoid copying of objects** in each function call, as mentioned above.

0.6 Results

Both the optimised approaches outperform the naive approach in the majority of cases. The most consistent approach is the **Parallel k-NN**, which constantly achieves at minimum **2X speedup** compared to the naive k-NN, for all values of n and d. The **Indexed-Parallel k-NN** performance depends on the dataset size and characteristics, like data distribution in space. For example if the points are very densely located next to one another, most VPTree iterations will search both inside and outside hyper spheres of a given vantage point, thus search the whole space. However testing proves that **VPTree k-NN** search achieves up to **16X speedup** compared to naive approach as the dataset increases.

0.7 Code

The code can be found on GitHub in this link. The parameters n and d (including others) can be modified in the **Constants.go** file under the **Constants** directory. The main file which runs all the methods is inside the **src** directory.