**Approximate Nearest Neighbour Search**

1. Cross-Platform Compilation

Can be compiled on Windows, Linux. Should work in MacOS too, but not tested as we don’t have a Mac machine.

1. High Performance Implementation on CPU

Parallel loop which iterates through the layers of Graph.

Enhance the efficiency of certain operations that involve searching for neighbors and modifying the graph structure.

Neighbour Search and Graph Modification: Within each layer, several tasks are performed simultaneously for different nodes in the graph. The “for” loop within the parallel section conducts tasks such as searching for neighbours, adding edges, and adjusting the graph structure.

1. High Performance Implementation on GPU

Cosine Similarity with GPU for benchmarking using CUDA.

1. Illegal Input Handling

Should be able to open the Input file.

The first line of the input file must have Dimensions, Base Vectors and Number of Query Vectors

Actual Base and Query Vectors should match the sizes defined on first line.

K should be a positive integer, and should be no greater than the total number of base vectors

1. Language Support – C++
2. Non – Trivial Optimization Techniques
3. Implemented HNSW (Hierarchical Navigable Small World), which refers to a data structure and algorithm used for approximate nearest neighbor search in high-dimensional spaces. It's a method designed to efficiently search for approximate nearest neighbors in large collections of high-dimensional data.

The following tasks were implemented –

1. When inserting an element into a structure, the process starts at the top level. It looks for the best nearby elements in that level. Then, it moves to the next level using those elements found earlier as a starting point. This keeps happening until the process is finished.
2. To find the approximate nearest neighbors in a specific layer 'l', a dynamic list called 'W' is used. This list starts by holding the initial set of closest elements known as enter points. During the search, this list is continuously updated. It works by checking the neighboring elements of the closest one that hasn't been evaluated yet in the list. This process repeats until all elements in the list have had their neighboring elements evaluated.
3. In the initial search phase, we use a simple greedy approach by setting the efficient parameter to 1. This avoids the need for introducing extra parameters. Once the search reaches a layer that's equal to or lower than 'l', the construction algorithm moves to its second phase. In this phase, two things change: 1) the efficient parameter increases from 1 to efficient construction to manage the effectiveness of the greedy search process; 2) the closest neighbors found in each layer are considered as potential connections for the inserted element.
4. KANNS is quite like the insertion algorithm when dealing with an item having a layer 'l' set to 0. The difference lies in how the closest neighbors found at the base layer are utilized: in KANNS, these neighbors, used as potential connections, are returned as the search result. The quality of this search is controlled by adjusting the 'efficient' parameter.
5. Normalizing the vectors before calculating cosine similarity. After normalizing the vectors, just computing dot product is enough, as we don’t have to worry about magnitude.

Benchmarking:

For input test – Dimensions 100, Base Vectors 10000, Query Vectors 1000

Total Euclidean time: 11.486 sec

Total HNSW time: 0.417 sec

Total cosine similarity time: 11.966 sec

Total cosine similarity with normalization time: 5.251 sec

Total cosine similarity with GPU: 0.27 seconds.

For input test – Dimensions 256, Base Vectors 10000, Query Vectors 1000

Total Euclidean time: 24.384

Total HNSW time: 1.121

Total cosine similarity time: 30.041

Total cosine similarity with normalization time: 13.495

Total cosine similarity with GPU: 0.498

For further input tests, we have skipped the Euclidean and cosine similarity CPU times, as it takes way too long to finish.

For input test – Dimensions 256, Base Vectors 50000, Query Vectors 5000

Total HNSW time: 6.457

Total cosine similarity with GPU: 4.941

For input test – Dimensions 256, Base Vectors 100000, Query Vectors 5000

Total HNSW time: 7.054

Total cosine similarity with GPU: 9.617

Observations:

For smaller dataset, it does not matter whether we choose a brute force algorithm for approximate nearest neighbour search or use a Graph Based ANN (HNSW). As the dataset increases (either the dimensions or base vectors), we can observe that there is a significant difference in computation time.

However, cosine similarity with GPU is significantly fast even for larger datasets and it takes around 100k base vectors to ensure the graph algorithm beats the GPU implementation of cosine similarity.

References

[1] Malkov, Yu A., and Dmitry A. Yashunin. "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs." *IEEE transactions on pattern analysis and machine intelligence* 42.4 (2018): 824-836.