AGENT-BASED VECTOR SEARCH ON GPU

AKBARBEK RAKHMATULLAEV

Winter 2025 Term Report

University of Washington

Mar 25, 2025

Project Committee:
Munehiro Fukuda, Ph.D., Committee Chair
Min Chen, Ph.D., Committee Member
Wooyoung Kim, Ph.D., Committee Member

I. Introduction

Vector search is an artificial intelligence and data retrieval method that employs mathematical vectors to represent and efficiently search through complex, unstructured data. It operates by linking similar mathematical representations of data and converting queries into these same vector formats. With both queries and data represented as vectors, the search for related data involves identifying the closest matches to the query vector, a process known as nearest neighbor search. Unlike traditional search algorithms, which rely on keywords, word frequency, or word similarity, vector search utilizes the distances within the vectorized dataset to identify similarity and semantic relationships. In today's world, vector search is primarily being used in domains like e-commerce, content discovery and recommendation systems, natural language processing (NLP), and many more.

The market offers many solutions for vector search, however, according to the latest benchmarks [1][2], one of the best-performing libraries is NGT (Neighborhood Graph and Tree) [3][4]. The NGT index combines both a graph and a tree, where the graph's vertices represent searchable objects and the tree is used to subdivide the entire vector space into smaller regions. This makes NGT a top performer compared to other solutions. Although NGT performs well with reasonably large datasets, it may face scalability issues with extremely large datasets or high-dimensional data, as it can run only on one computing node/Central Processing Unit (CPU), and does not provide Graphics Processing Unit (GPU) support for Approximate Nearest Neighbor (ANN) [5].

The Multi-Agent Spatial Simulation (MASS) [6][7] library is a parallel and distributed computing framework designed for large-scale spatial and agent-based simulations. It operates primarily with two concepts: Places and Agents. Generally, Places are the spatial locations or cells that form the simulation environment and are capable of exchanging information with any other Places. Agents are active entities that can move between Places, perform actions, and interact with each other and their environment. MASS CUDA [8] is a library designed to facilitate the execution of parallel computations using mobile agents on GPUs.

When implementing vector search on a GPU, two primary algorithmic approaches are commonly used: Hierarchical Navigable Small World (HNSW)[9] and Inverted File Index with Product Quantization (IVFPQ)[10].

HNSW (Hierarchical Navigable Small World) is a graph-based approach for approximate nearest neighbor search, designed to efficiently navigate large-scale high-dimensional spaces. It constructs a multi-layered, small-world graph structure where search operations traverse the graph using greedy strategies. HNSW provides high recall rates and fast search times, making it a popular choice for large-scale vector search applications. However, the graph-based nature of HNSW requires significant memory overhead, and updates to the index can be computationally expensive. While MASS Graph — a graph-based extension of MASS — could potentially optimize HNSW with agent-based search, but its development was not mature enough for practical implementation at the time of this work. Instead, IVFPQ was chosen due to its well-established efficiency in large-scale vector search and its compatibility with MASS CUDA, allowing for immediate integration of agent-based optimizations.

IVFPQ (Inverted File System with Product Quantization) is a widely used indexing technique that balances search efficiency and memory usage. It partitions the vector space into clusters and applies product quantization to compress the data, allowing fast approximate searches while reducing storage requirements. IVFPQ is particularly effective for large-scale nearest neighbor search, as it significantly reduces computational complexity. However, its performance depends on proper tuning of parameters such as the number of clusters and quantization levels. While it provides a strong foundation for efficient search, further enhancements using agent-based approaches like MASS CUDA could improve adaptability and dynamic workload distribution across GPUs, which is a part of this project.

The idea of this project is to try to leverage the capabilities of GPUs using mobile agents by the MASS CUDA library and improve IVFPQ algorithm to perform better. A more detailed explanation of the implementation is provided in the next sections.

II. Background

The vector searching landscape is highly dynamic, with frequent advancements and discoveries. Current vector search engines, such as NGT, SPTAG [11], ANNOY [12], etc., are effective but have various limitations in terms of scalability, speed, or precision. The idea behind this project is to utilize the best state-of-the-art methods and algorithms in terms of speed and precision and scale them using mobile agents. As a result, there is an opportunity to develop an agile vector search engine.

While existing solutions like NGT, SPTAG, ANNOY, etc. offer effective vector search, our deliverable stands out by scaling and enhancing the IVFPQ algorithm on GPU using mobile agents.

III. Challenges

This quarter, one of the biggest challenges was navigating through a vast amount of literature and research to develop an approach that could enhance the IVFPQ algorithm. This was particularly difficult because IVFPQ is already a well-established and highly optimized method, leaving little room for meaningful improvements. Another problem was configuring the machines to efficiently execute operations using CUDA. Many of the issues encountered stemmed from library dependencies and the complexities of setting up the computing environment.

IV. Goals

The primary objective this quarter was to develop a methodology, and its implementation, for effectively utilizing MASS CUDA agents within IVFPQ, which has base implementation available from cuVS (CUDA Vector Search) [13] developed by Nvidia Corporation. A significant portion of the time was dedicated to studying codebase (more than 6000 lines of code) and literature, including research papers, books, and articles, as well as experimenting with different variations of the algorithm. Eventually, I formulated an initial version of a new approach that integrates MASS CUDA and presented it to Professor Fukuda. After making some minor adjustments, we finalized the methodology. Over the past few weeks, I have been working on implementing this approach. However, due to the complexity of the code, details of which will be discussed in the next section, there is still work to be done, particularly in the implementation of agents.

V. Implementation

The core idea behind this approach is to distribute the search operation across multiple CUDA threads efficiently, leveraging MASS's ability to manage parallel computations. This approach was inspired by the work of Alex Li and Professor Fukuda[14]. Each dimension of a PQ-encoded vector is treated as a separate Place, which stores sorted values for that dimension along with their corresponding vector IDs. Agents are then dispatched to these Places to perform a binary search for the closest quantized value, ultimately contributing to an overall search result (see Figure 1). While significant progress has been made in designing the structure and implementing key components, the full implementation is still ongoing.

The implementation is structured around two custom classes: VectorDimensionPlace and QueryDimensionAgent. The VectorDimensionPlace class extends MASS's Place class and serves as the core storage unit for each dimension's sorted PQ values and associated vector IDs. It provides an efficient device-side method to find the closest match using binary search. The code is available in Listing 1. On the other hand, QueryDimensionAgent is an Agent that queries a specific Place by executing the binary search algorithm and returning the best-matching vector ID. The code is available in Listing 2. The workflow begins by initializing MASS and creating an array of Places (Listing 4, lines 75-113, also Listing 3, line 139), where each Place is responsible for one dimension of the PQ-encoded vectors. The quantized values from the dataset are sorted and stored in these Places to allow efficient lookups during the search phase. The code is available in Listing 3.

Once the Places are set up, a query vector is processed by dispatching one QueryDimensionAgent per dimension. Each Agent independently queries its assigned Place, searching for the closest quantized value to its respective component of the query vector. This results in a set of "votes," where each Agent suggests a potential matching vector ID. These votes are then aggregated to determine the final closest vector (Listing 3, lines 139-153).

While the fundamental logic and architecture are in place, the full implementation has not yet been completed due to the complexity of integrating MASS CUDA with the data structures and query workflow. One of the main challenges is to tie together kernel code available in cuVS with MASS CUDA. Current implementation is a primitive one that does not calculate how many threads, blocks, shared memory, L1 cache or global memory is required. It also lacks proper usage of LUTs (Look-up Tables). Additionally, testing and debugging parallel execution in MASS CUDA require careful verification to ensure correctness and performance gains over traditional methods. Despite these challenges, we made calculations of computational complexity of such an approach which demonstrates the feasibility of this approach and suggests that once finalized, it will provide an efficient and scalable solution for PQ-based vector search. The next steps involve refining the agent execution model, optimizing memory allocation, and validating performance improvements through extensive testing.

VI. Future Work

The objective for the upcoming quarter is to finalize Place and Agent classes, proper memory management, test and benchmark implementation, and work on a paper.

References

[1] "ANN - Benchmarks." [Online]. Available: https://github.com/erikbern/ann-benchmarks?tab=readme-ov-file

[2] M. Aumüller, E. Bernhardsson, and A. Faithfull, "ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms," 2018, *arXiv*. doi: 10.48550/ARXIV.1807.05614.

[3] M. Iwasaki, "Proximity search in metric spaces using approximate k nearest neighbor graph,"

[4] *NGT*. Yahoo! Japan. [Online]. Available: https://github.com/yahoojapan/NGT

[5] A. Andoni, P. Indyk, and I. Razenshteyn, "Approximate Nearest Neighbor Search in High Dimensions," 2018, *arXiv*. doi: 10.48550/ARXIV.1806.09823.

[6] J. Emau, T. Chuang, and M. Fukuda, "A multi-process library for multi-agent and spatial simulation," in *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada: IEEE, Aug. 2011, pp. 369–375. doi: 10.1109/PACRIM.2011.6032921.

[7] *MASS*. University of Washington. [Online]. Available: https://depts.washington.edu/dslab/MASS/

[8] L. Kosiachenko, N. Hart, and M. Fukuda, "MASS CUDA: A General GPU Parallelization Framework for Agent-Based Models," in *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection*, vol. 11523, Y. Demazeau, E. Matson, J. M. Corchado, and F. De La Prieta, Eds., in Lecture Notes in Computer Science, vol. 11523. , Cham: Springer International Publishing, 2019, pp. 139–152. doi: 10.1007/978-3-030-24209-1_12.

[9] Yu. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," 2016, arXiv. doi: 10.48550/ARXIV.1603.09320.

[10] H. Jégou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," IEEE Trans. Pattern Anal. Mach. Intell., vol. 33, no. 1, pp. 117–128, Jan. 2011, doi: 10.1109/TPAMI.2010.57.

[11] *SPTAG*. Microsoft. [Online]. Available: https://github.com/microsoft/SPTAG

[12] E. Bernhardsson, *ANNOY*. [Online]. Available: https://github.com/spotify/annoy

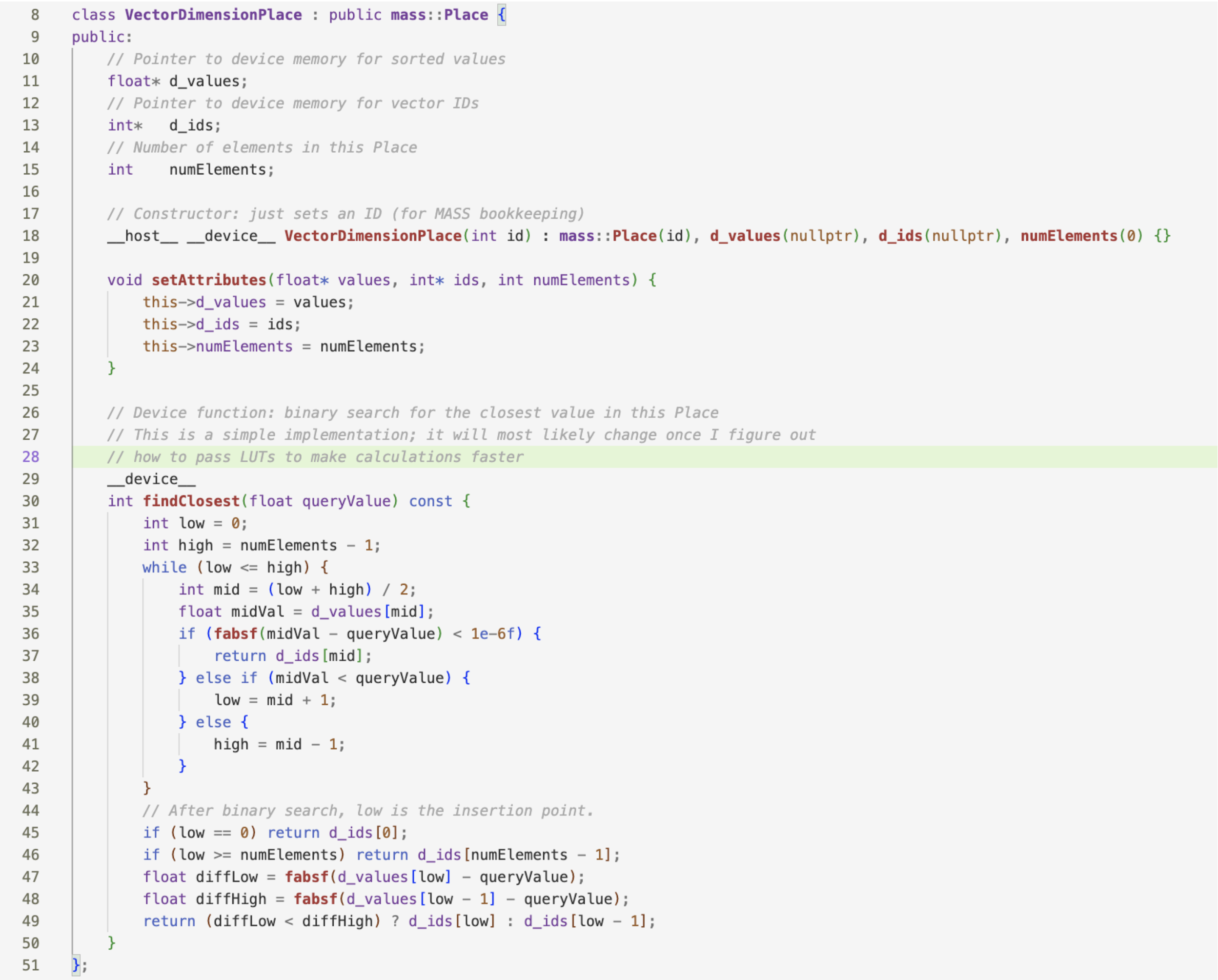
[13] *cuVS*. [Online]. Available: https://github.com/rapidsai/cuvs

[14] A. Li and M. Fukuda, "Agent-Based Parallelization of a Multi-Dimensional Semantic Database Model," presented at the IEEE 24th Int'l Conf. on Information Reuse and Integration for Data Science, Aug. 2023, pp. 64–69.

Appendix

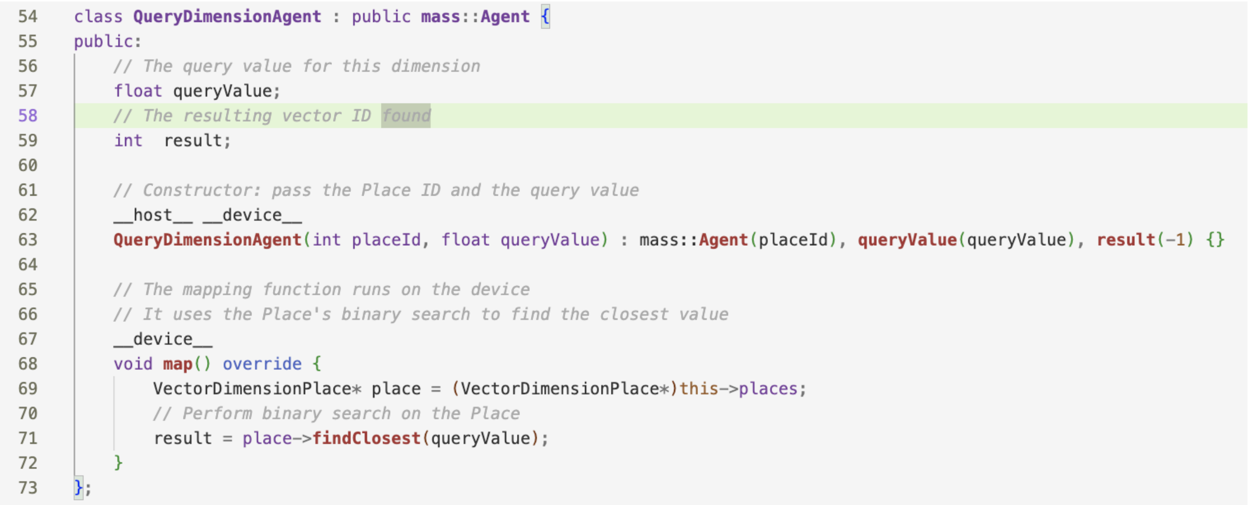
A. Code

Listing 1: Place class



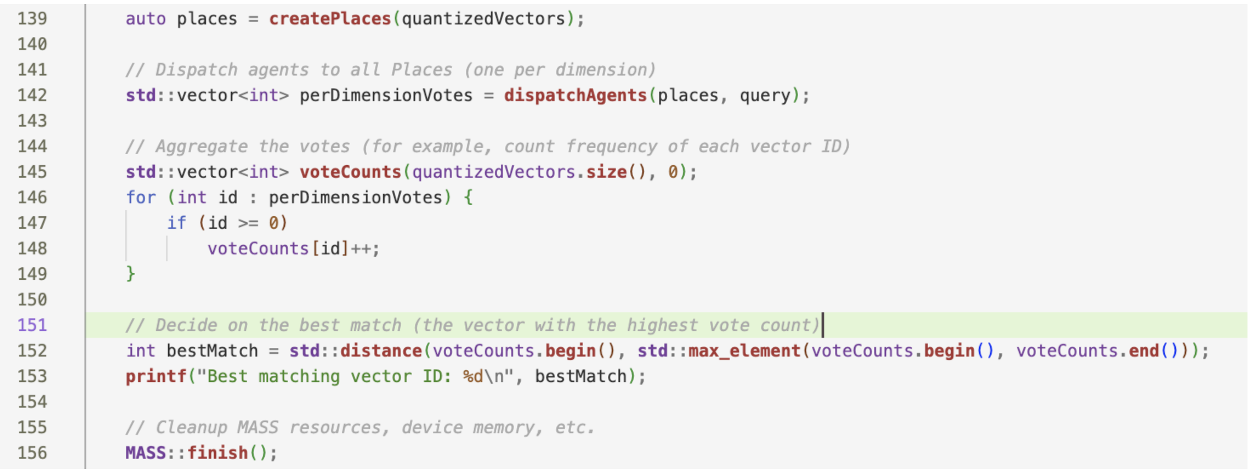
```
class VectorDimensionPlace : public mass::Place {
public:
    // Pointer to device memory for sorted values
    float* d_values;
    // Pointer to device memory for vector IDs
    int*   d_ids;
    // Number of elements in this Place
    int    numElements;

    // Constructor: just sets an ID (for MASS bookkeeping)
    __host__ __device__ VectorDimensionPlace(int id) : mass::Place(id), d_values(nullptr), d_ids(nullptr), numElements(0) {}

    void setAttributes(float* values, int* ids, int numElements) {
        this->d_values = values;
        this->d_ids = ids;
        this->numElements = numElements;
    }

    // Device function: binary search for the closest value in this Place
    // This is a simple implementation; it will most likely change once I figure out
    // how to pass LUTs to make calculations faster
    __device__
    int findClosest(float queryValue) const {
        int low = 0;
        int high = numElements - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            float midVal = d_values[mid];
            if (fabsf(midVal - queryValue) < 1e-6f) {
                return d_ids[mid];
            } else if (midVal < queryValue) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        // After binary search, low is the insertion point.
        if (low == 0) return d_ids[0];
        if (low >= numElements) return d_ids[numElements - 1];
        float diffLow = fabsf(d_values[low] - queryValue);
        float diffHigh = fabsf(d_values[low - 1] - queryValue);
        return (diffLow < diffHigh) ? d_ids[low] : d_ids[low - 1];
    }
};
```

Listing 2: Agent class



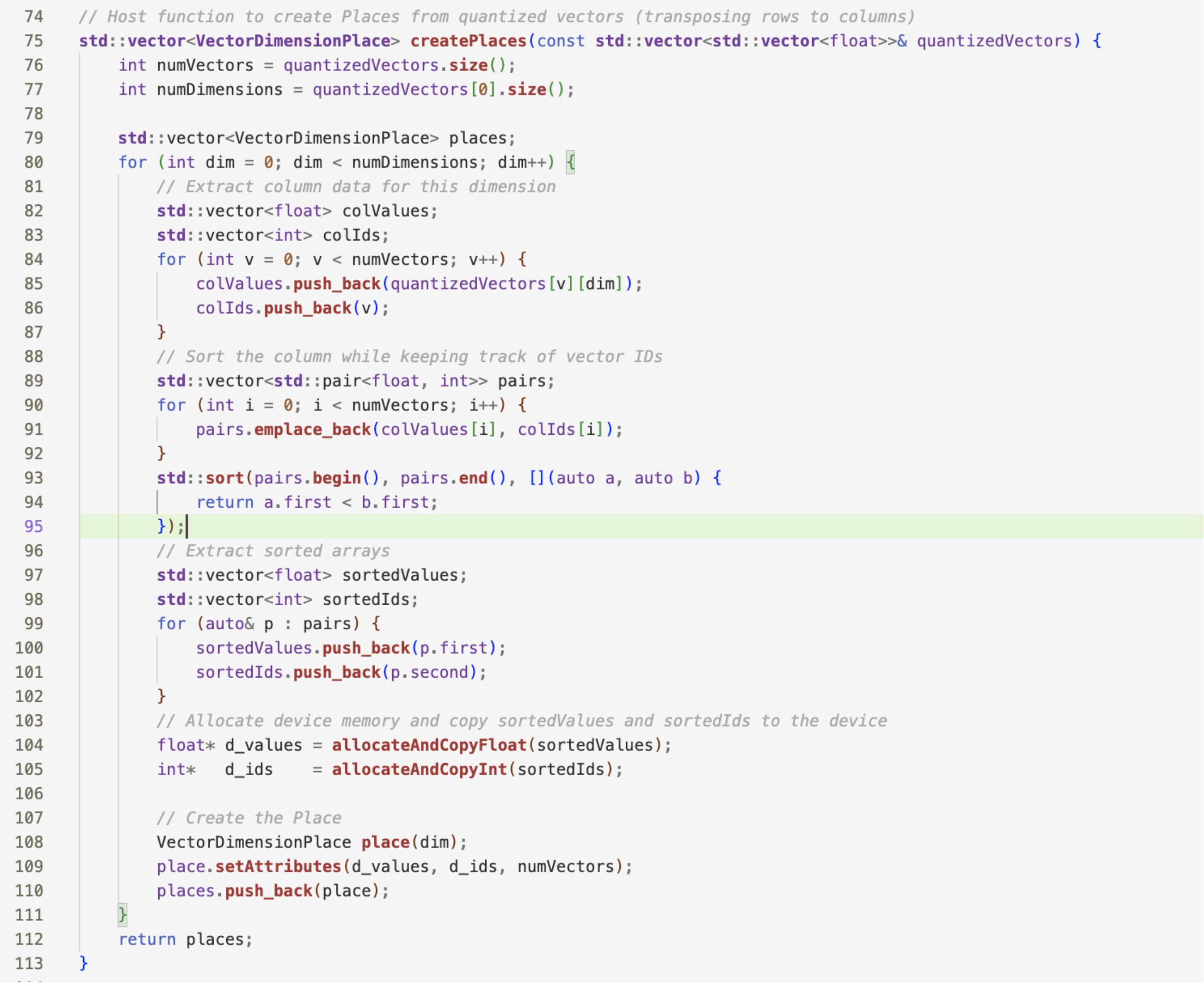
```cpp
class QueryDimensionAgent : public mass::Agent {
public:
    // The query value for this dimension
    float queryValue;
    // The resulting vector ID found
    int  result;

    // Constructor: pass the Place ID and the query value
    __host__ __device__
    QueryDimensionAgent(int placeId, float queryValue) : mass::Agent(placeId), queryValue(queryValue), result(-1) {}

    // The mapping function runs on the device
    // It uses the Place's binary search to find the closest value
    __device__
    void map() override {
        VectorDimensionPlace* place = (VectorDimensionPlace*)this->places;
        // Perform binary search on the Place
        result = place->findClosest(queryValue);
    }
};
```

Listing 3: General part



```cpp
    auto places = createPlaces(quantizedVectors);

    // Dispatch agents to all Places (one per dimension)
    std::vector<int> perDimensionVotes = dispatchAgents(places, query);

    // Aggregate the votes (for example, count frequency of each vector ID)
    std::vector<int> voteCounts(quantizedVectors.size(), 0);
    for (int id : perDimensionVotes) {
        if (id >= 0)
            voteCounts[id]++;
    }

    // Decide on the best match (the vector with the highest vote count)
    int bestMatch = std::distance(voteCounts.begin(), std::max_element(voteCounts.begin(), voteCounts.end()));
    printf("Best matching vector ID: %d\n", bestMatch);

    // Cleanup MASS resources, device memory, etc.
    MASS::finish();
```

Listing 4: Create Places



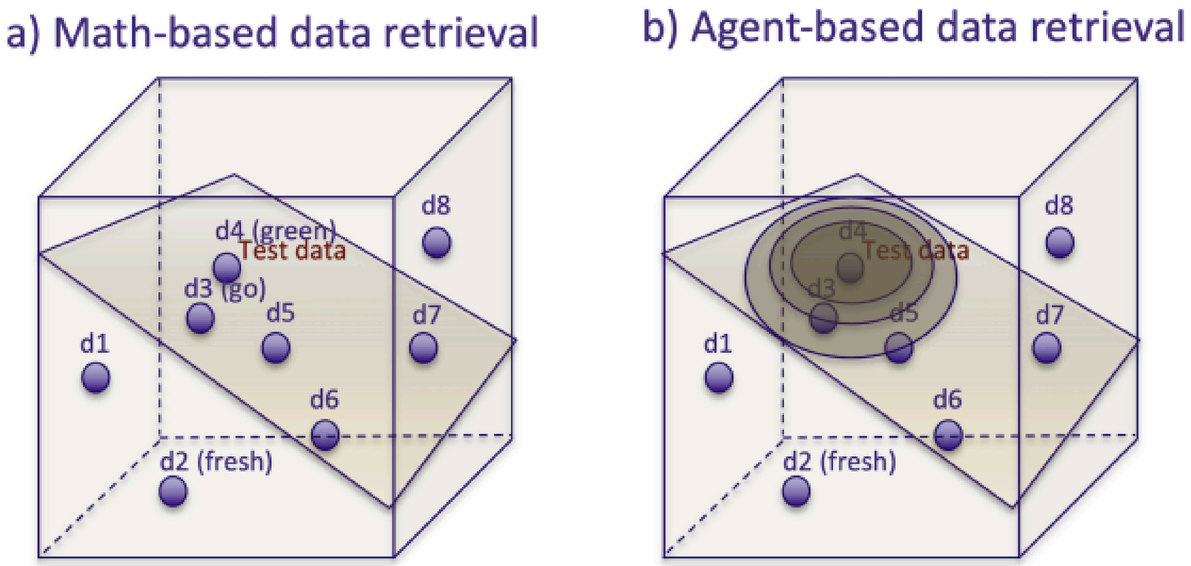
```cpp
// Host function to create Places from quantized vectors (transposing rows to columns)
std::vector<VectorDimensionPlace> createPlaces(const std::vector<std::vector<float>>& quantizedVectors) {
    int numVectors = quantizedVectors.size();
    int numDimensions = quantizedVectors[0].size();

    std::vector<VectorDimensionPlace> places;
    for (int dim = 0; dim < numDimensions; dim++) {
        // Extract column data for this dimension
        std::vector<float> colValues;
        std::vector<int> colIds;
        for (int v = 0; v < numVectors; v++) {
            colValues.push_back(quantizedVectors[v][dim]);
            colIds.push_back(v);
        }
        // Sort the column while keeping track of vector IDs
        std::vector<std::pair<float, int>> pairs;
        for (int i = 0; i < numVectors; i++) {
            pairs.emplace_back(colValues[i], colIds[i]);
        }
        std::sort(pairs.begin(), pairs.end(), [](auto a, auto b) {
            return a.first < b.first;
        });
        // Extract sorted arrays
        std::vector<float> sortedValues;
        std::vector<int> sortedIds;
        for (auto& p : pairs) {
            sortedValues.push_back(p.first);
            sortedIds.push_back(p.second);
        }
        // Allocate device memory and copy sortedValues and sortedIds to the device
        float* d_values = allocateAndCopyFloat(sortedValues);
        int*   d_ids    = allocateAndCopyInt(sortedIds);

        // Create the Place
        VectorDimensionPlace place(dim);
        place.setAttributes(d_values, d_ids, numVectors);
        places.push_back(place);
    }
    return places;
}
```

B. How To Run a Program

To run the Feature Extraction program, Python 3.9 or higher, along with the corresponding version of PyTorch, is required. For the IVF PQ program, the latest version of cuVS must be installed, ensuring that all associated prerequisites, including compatibility with the appropriate version of the CUDA Toolkit, are met. Additionally, the libnpy library is necessary to convert the NumPy arrays generated by the Feature Extraction program into a C++-compatible vector data structure.

C. Figures

Figure 1:



a) Math-based data retrieval

Sort: d4, d3, d5, d6, d7.

b) Agent-based data retrieval

Use agent propagation and collision.