IVF Singular Search: Agent-Based Implementation of Vector Search on GPU

Akbarbek Azamatovich Rakhmatullaev

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington
2025

Committee:
Munehiro Fukuda
Min Chen
Wooyoung Kim

Program Authorized to Offer Degree:
Computer Science and Software Engineering

University of Washington

Abstract

IVF Singular Search: Agent-Based Implementation of Vector Search on GPU

Akbarbek Azamatovich Rakhmatullaev

Chair of the Supervisory Committee:

Munehiro Fukuda

Bothell School of Science, Technology, Engineering and Mathematics

Vector search plays a crucial role in large-scale similarity search applications, with IVF (Inverted File Index) being a widely used indexing method due to its balance between accuracy and efficiency. However, traditional vector search algorithms that used IVF as an indexing method, such as IVF Flat and IVFPQ, yield results by brute force searching within each cluster/list. This paper introduces a new IVF-based vector search algorithm, called IVF Singular Search, which does the search within each cluster/list through a different arrangement of data and traversal using the Binary Search. In order to accelerate the development phase, the author used MASS CUDA to handle the searching part, which allowed to leverage the abstraction level of the code. We evaluated our IVF Singular Search, implemented for GPUs using MASS CUDA, against two other algorithms, IVF Flat and IVFPQ, demonstrating the significant speed efficiency of the approach. The findings suggest IVF Singular Search can make vector search more efficient and robust on infrastructure that requires immediate response, such as navigation systems or robots.

ACKNOWLEDGEMENTS

I would like to express my immense gratitude to my supervisor and committee chair, Professor Fukuda, for his support and encouragement throughout the research journey. His valuable feedback and high standards made finishing this thesis and obtaining a degree achievable. I am also grateful to my colleagues and friends at the Distributed Systems Lab for the introspective discussions we had and their thoughtful advice.

I would like to give special thanks to my family for the love, care, patience, and opportunity they provided me with to pursue my dreams. I am forever grateful for everything they did for me.

Finally, I would like to thank the Triune God for letting me through all the difficulties and providing me with his guidance every day.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

This chapter will discuss the main motives of this thesis as well as its objectives. It will cover current trends in the industry, the general description about the field of the research, which is vector search algorithms, proposed changes and innovations, and aimed goals.

1.1 Motivation

Artificial Intelligence (AI) is the primary topic of the computer science of the current year, as well as the past three years, since the release of publicly available AI chatbots. Hundreds and hundreds of new papers, libraries, articles and tools are being published and developed in this field constantly. One of the main components of such new AI models is the ability to access data outside of its own memory, removing the need to train and retrain on fresh data. This is achieved by the vector search. 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, data frequency, or parameter similarity, vector search algorithms can utilize the distances within the vectorized dataset to identify similarity and relationships. Apart from the AI models, vector search is primarily being used in domains like e-commerce, content discovery and recommendation systems, natural language processing (NLP), and many more.

This thesis was started as a capstone project first, and the initial idea was to try to develop a proprietary, improved vector search engine using agents. However, as more work was done on the project during the first quarter, it was discovered that no big improvements could be made with such an approach. This is when the idea of a new vector search algorithm came up, called IVF

Singular Search (IVFSS), where IVF stands for Inverted File Index [1][2], that, according to the initial calculations, would show promising results.

In order to provide the general idea, it can be said that IVFSS is similar to the IVF Flat, another vector search algorithm, which is known for its well-established efficiency in large-scale vector search. However, the main difference is that IVFSS provides only one closest vector for the given query vector (hence the word Singular is in the name of the algorithm) with better performance compared to IVF Flat, which provides k number of closest vectors in descending order. Another industry-wide IVF-based vector search algorithm is IVFPQ (IVF with Product Quantization) [2][3], which quantizes vectors in order to improve speed and memory consumption as a trade-off to precision and recall. Similarly to IVF Flat, IVFPQ also provides the number of closest vectors in descending order. Within a scope of this thesis, IVFPQ was used to serve as a third option in benchmarks, to give a clearer picture. Since IVF-based vector search algorithms are capable of showing better results when implemented on GPUs, IVFSS was also programmed using C/C++ and CUDA C++. To accelerate the implementation of the IVFSS algorithm on the GPU, MASS CUDA [4] was used to handle the search part. MASS [5][6] is a parallelizing library for Multi-Agent Spatial Simulation (hereby named MASS), and its version for CUDA is called MASS CUDA. This library was utilized because it includes entities called agents, which, by acting autonomously, help to handle subtle memory management and multi-threading, and thus provide a higher abstraction level. More detailed explanation of IVFSS's design and implementation, IVF Flat, IVFPQ, and MASS CUDA's descriptions, and all other needed information are provided in the next chapters.

1.2 Research Objectives

Since this thesis's main point is to deliver a new vector search algorithm, it has to reach several major goals in different categories. The objectives of the thesis are as follows:

1. Design and formally formulate the IVFSS algorithm

2. Implement a working IVFSS algorithm for GPUs using C/C++, CUDA C++, and MASS

CUDA, making sure the code's adaptability to different variables and datasets, as well as proper logging, deterministic results, and immunity to bugs or errors

3. Benchmark IVFSS against IVF Flat and IVFPQ for performance, demonstrating speed, memory, and accuracy of the results

4. Demonstrate the livability of IVFSS as an industry-level vector search algorithm, alongside IVF Flat and IVFPQ, and showcase MASS CUDA's ability to be used in non-agent-centric applications as a third-party library

Chapter 2

BACKGROUND

In this chapter, different core and foundational concepts that shaped the creation of the IVFSS algorithm will be discussed. This chapter will cover types of vector search algorithms, IVF and why it was chosen, MASS, MMM, and the implementation of MMM using MASS.

2.1 ANN

Generally, there are two ways one can approach vector searching: Brute-Force and "Approximate Nearest Neighbor" (ANN). Brute-force vector search simply means exhaustive search, and thus comparing a query vector to every other vector within a dataset. Although barely any big project utilizes Brute-Force vector search because of its poor performance in terms of speed and memory, it is worth noting that it provides 100% accuracy. On the other hand, ANN vector search supposes a sacrifice in terms of some accuracy for the sake of a performance boost of speed and memory. ANN search does not necessarily query every single vector in the dataset against the query vector, but rather selects the ones it counts as worthwhile checking. Also, ANN vector search includes many different algorithms, and the majority of ANN vector search algorithms are known for using indexing techniques to optimize dataset vectors' storage, improving the access to them in one way or another, e.g., speed or memory-wise [7][8][9]. Thus, such ANN vector search algorithms can be classified into four main families based on their indexing techniques: Tree-based, Hashing-based, Clustering-based, and Graph-based.

Tree-based algorithms organize data hierarchically by recursively splitting the vector space into smaller regions, similar to a decision tree. While efficient for low-dimensional data, they suffer significantly from the curse of dimensionality, where performance degrades to linear scan speeds in high-dimensional spaces. One of the popular examples of tree-based algorithms is ANNOY [10].

Hashing-based algorithms, primarily known as Locality Sensitive Hashing (LSH) [11] algorithms, transform vectors into compact binary codes where similar items have a high probability of

collision. This allows for extremely fast lookups and low memory usage, though often at the cost of lower accuracy compared to other methods.

Clustering-based algorithms, partition the dataset into groups represented by centroids. Search operations first identify the closest centroids to the query and then exhaustively search only the vectors within those specific clusters, offering a scalable balance between speed and precision. Examples of clustering-based algorithms are IVF Flat and IVFPQ, where both use IVF index to cluster data.

Graph-based algorithms, like Hierarchical Navigable Small Worlds (HNSW) [12], construct a proximity graph where vectors are nodes connected to their nearest neighbors. Searches traverse this graph greedily, hopping from node to node to find the closest match, currently providing state-of-the-art performance in high-dimensional spaces despite higher memory costs.

According to today's industry standards, tree-based and hashing-based algorithms are not particularly relevant, as they are not as efficient as clustering-based or graph-based algorithms. The main competitors in the market of vector search algorithms are IVFPQ and HNSW. The main difference between these two is that IVFPQ, as well as IVF Flat, can run effectively on a GPU, while HNSW is mainly limited to a CPU. Therefore, the reason why a clustering-based model was chosen for this thesis instead of a graph-based one is that MASS CUDA library was in a state that would allow it to implement IVF-based algorithms adequately on a GPU. Also, the author's advanced prior knowledge of IVFPQ played a big role in choosing this model.

2.2 IVF

An Inverted File Index (IVF), also known as an inverted file, is a fundamental index data structure used to store and organize data for efficient retrieval. It dramatically accelerates ANN searches in high-dimensional vector datasets. In order to build this index, the entire dataset of vectors is first clustered into a predefined number of regions (defined by a variable called n_list), typically using an algorithm like K-Means Clustering [13]. Once clustering is finished, the center of each Voronoi region, a vector called a centroid, acts as a generalizing central point of all vectors in this region, and the region itself is called a "list" or "cluster". Thus, the list data structure stores all the

required information about vectors in it, such as the centroid vector, its size, ID, feature data type, and others. Overall, building an Inverted File Index results in having a multitude of such lists that partition the entire vector space into smaller sub-spaces. Below, the flow of building the IVF index is demonstrated:

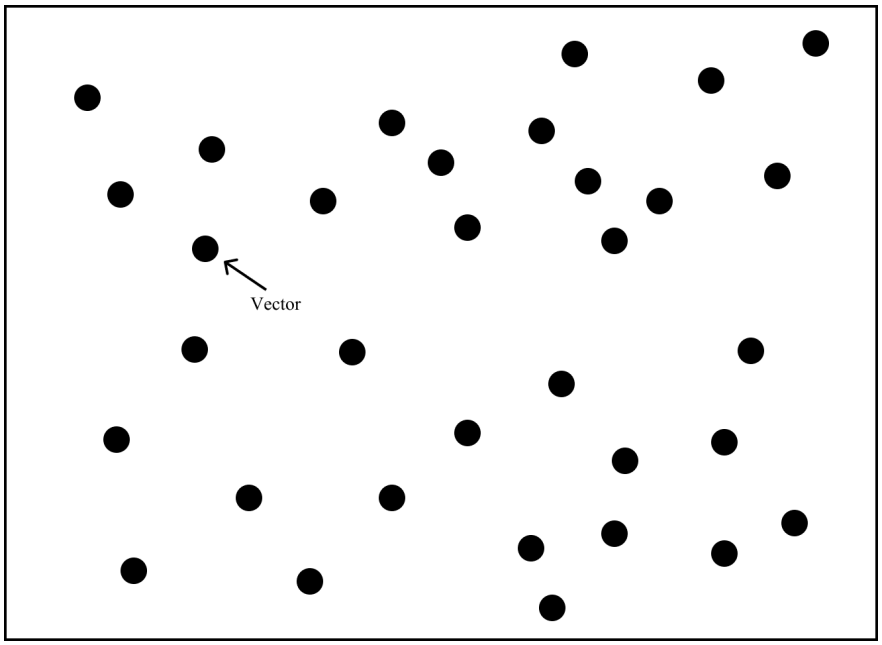

Figure 2.1: Dataset of vectors over vector space

Initially, there is a dataset of vectors, that can be depicted as vectors scattered over the vector space as shown in Fig. 2.1.

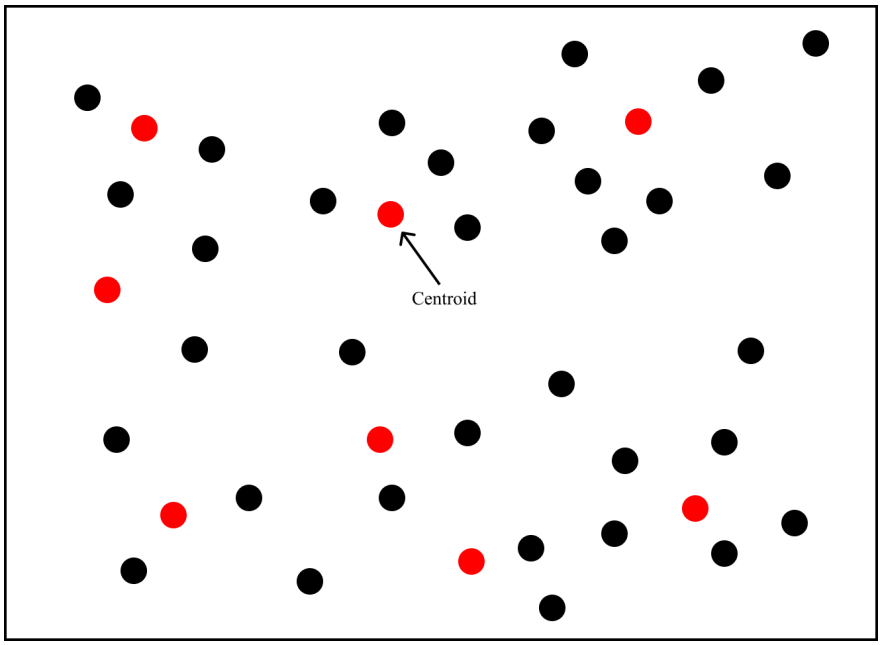

Figure 2.2: Choosing centroids to run K-Means Clustering algorithm

Next, K number of centroids is defined, to perform the K-Means Clustering as shown in Fig. 2.2.

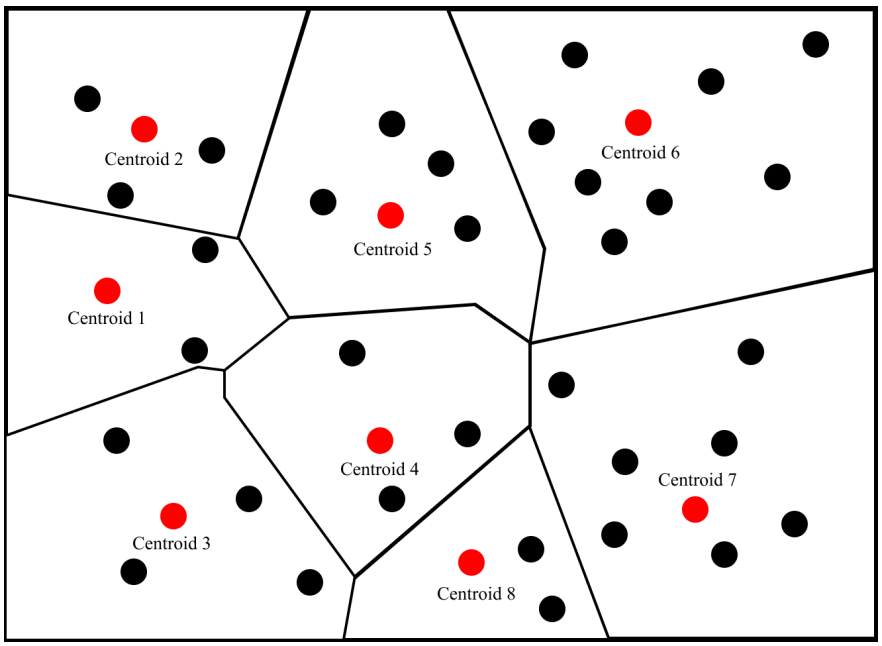

Figure 2.3: Final result of K-Means Clustering, vector space being divided into Voronoi regions

Once K-Means Clustering is finished, there will be n_list number of Voronoi regions, also known as lists, each having its own centroid, and vectors within it, as shown in Fig. 2.3.

2.3 MASS CUDA

The Multi-Agent Spatial Simulation (MASS) [5][6] 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. Fig. 2.4 depicts the general programming model of the MASS library.

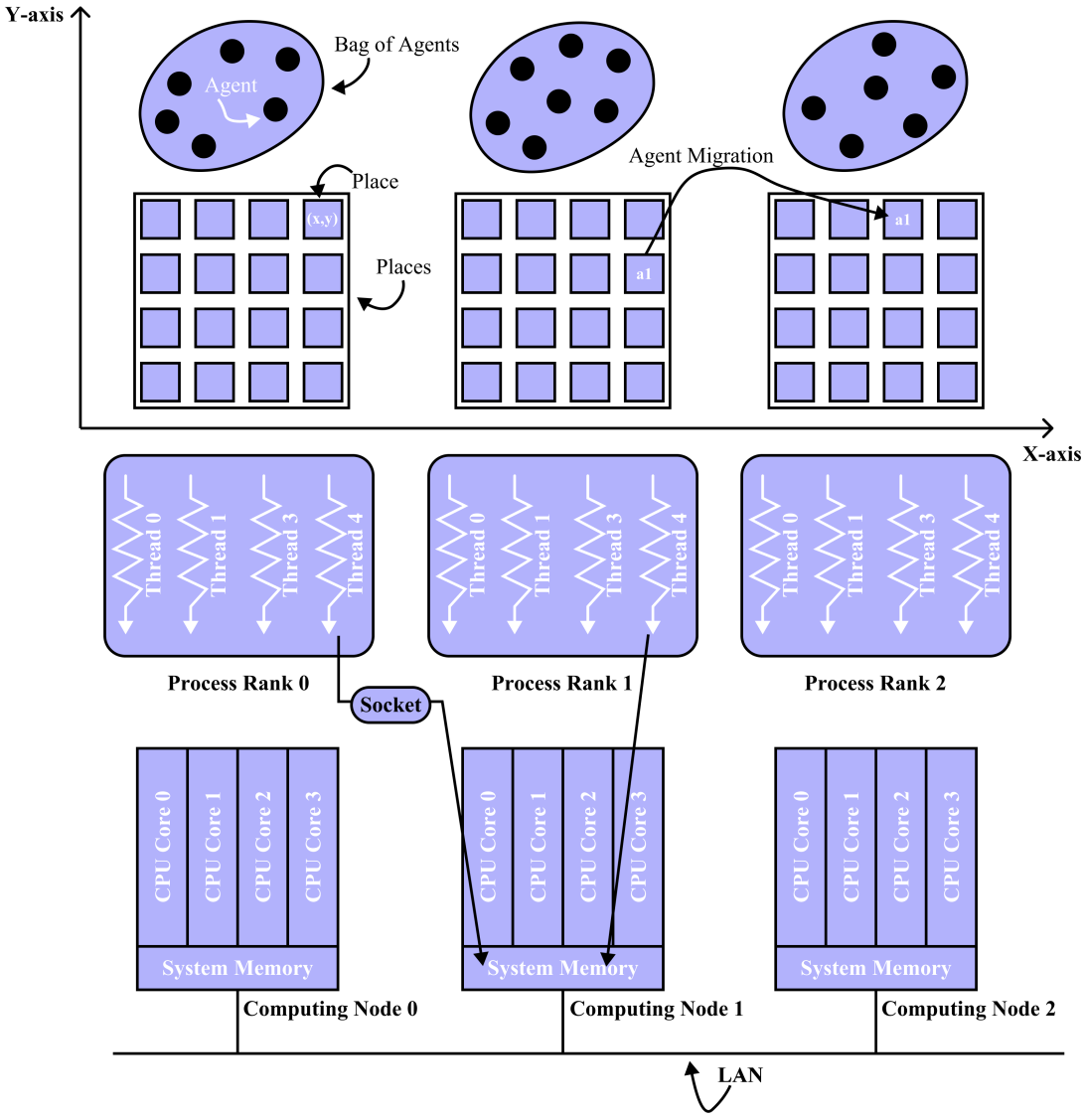

Figure 2.4: General programming model of MASS library

Essentially, MASS is used to create and instantiate a multidimensional distributed array with Places and populate them with Agents. Places can also facilitate parallel function calls and exchange data among each other, through such functions as $callAll()$ and $exchangeAll()$. On the other hand, Agents can go from Place to Place as queries, and they also have a function $callAll()$, which they use to schedule the next behaviour, and commit them using the function $manageAll()$. Execution of all MASS's built-in functions happens in parallel over clusters of computers.

MASS CUDA [4] is a library designed to facilitate the execution of parallel computations using mobile agents on GPUs.

2.4 MMM

Mathematical Model of Meaning (MMM) [14][15] is a model that provides capabilities to query and extract data items with equivalent or similar meaning or to recognize different meanings of a data item in a meta database system. The primary feature of this model is that the specific meaning of a data item can be recognized unambiguously and dynamically according to the context. The mathematical model of meaning consists of:

1. Creation of Metadata Space (MDS)

2. Data Retrieval

The creation of Metadata Space in MMM starts by getting a dataset of m data items, each having n features, and building m by n data matrix A, where $i-th$ row is data item d_i. This means that data item d_i is characterized by n features $(f_1, f_2, f_3, \ldots, f_n)$, where $i = 1, 2, \ldots, m$. This data matrix A is illustrated in Fig. 2.5.

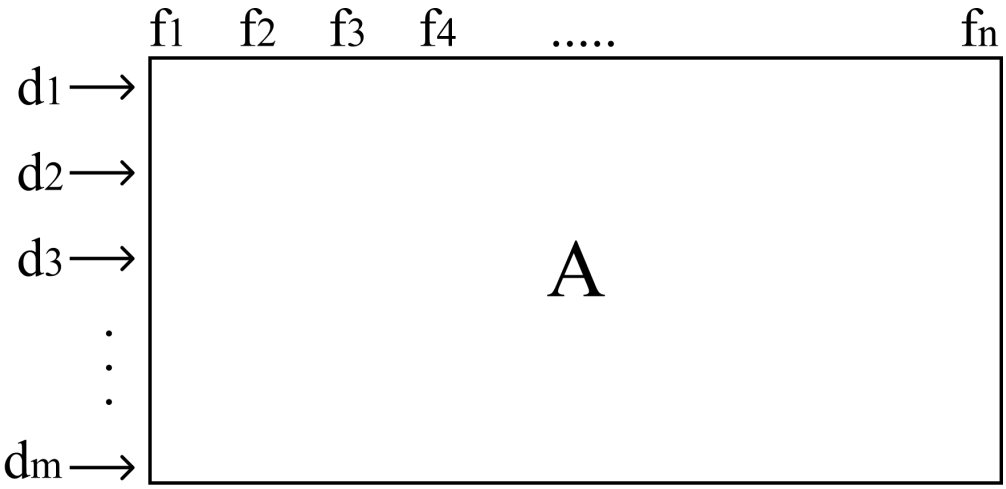

Figure 2.5: Representation of data matrix $A[m][n]$

To proceed, MMM needs to define Metadata Space MDS (also called Image Space I sometimes). To do that, it constructs a correlation matrix with respect to the features. This is done by getting the transpose matrix A^T of matrix A, and multiplying them to each other, which results in $A^T A$. Next, the model performs the eigenvalue decomposition of the correlation matrix and normalizes the eigenvectors:

$$A^T A = Q \begin{pmatrix} \lambda_1 & & & \\ & \cdots & & \\ & & \lambda_v & \\ & & & 0 \end{pmatrix} Q^T, \ where \ 0 \leq v \leq n.$$

The orthogonal matrix Q is defined by $Q = (q_1, q_2, q_3, ..., q_n)^T$, where q_i is normalized eigenvectors of $A^T A$. After that, it defines Metadata Space MDS as the span of the eigenvectors which correspond to nonzero eigenvalues, and such eigenvectors are called semantic elements. Then, MMM defines a set of semantic projection Π_v, where it considers the set of all projections from Metadata Space MDS to the invariant subspaces, also known as eigen spaces. Last but not least, MMM constructs the semantic operator S_p. This operator S_p is needed to determine the semantic projection according to the context, when such is provided.

As for the querying part, MMM needs to define data item set W, meaning set of available items to query against, and context. Then, once the query data item is provided, the model starts

computing the distance between this query data item and each data item in set W, using special metric ρ. Generally, this metric ρ is the summation of L results of multiplication between specific weight, that depends on the context, and the Euclidean distance between dimensions of the query and dataset data items, where number L also depends on the context. The correct answer is the one that corresponds to the needed metric value (closest or farthest data item). Note that both query data items and every data item in set W should be proper coordinates in MDS. The Euclidean distance calculation part of the metric calculation is illustrated in Fig. 2.6.

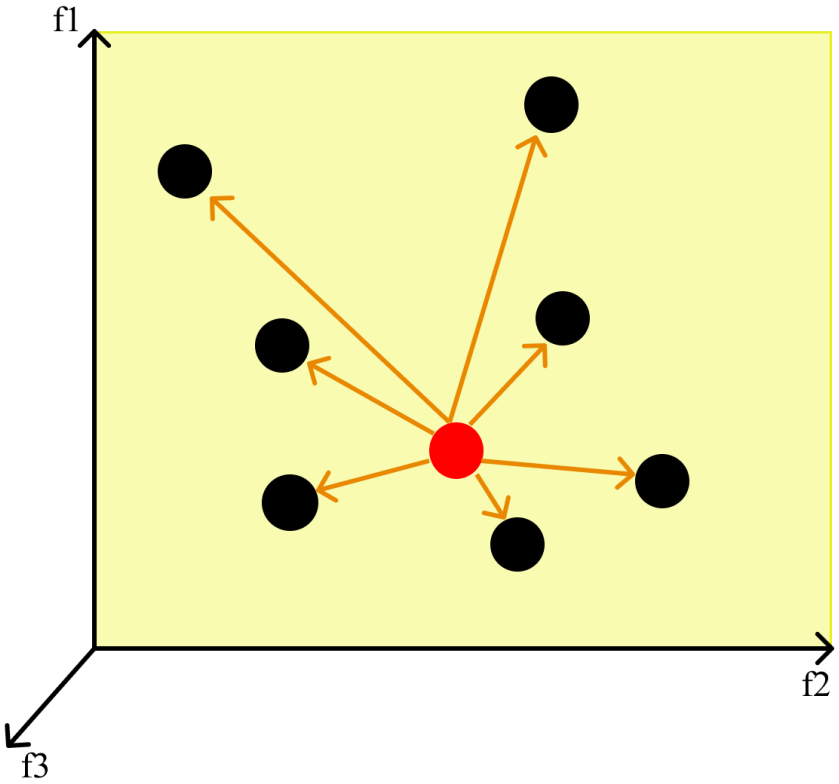

Figure 2.6: Exhaustive Euclidean distance calculation between query data item (marked red) and data items in dataset W (marked black) within a specific dimension/feature, which is denoted as a yellow plane

The different way of illustrating it would be through the following Fig. 2.7, where value at dimension/feature $f2$ of the query data item (marked red) gets compared by the Euclidean distance to all other values of other data items (marked black):

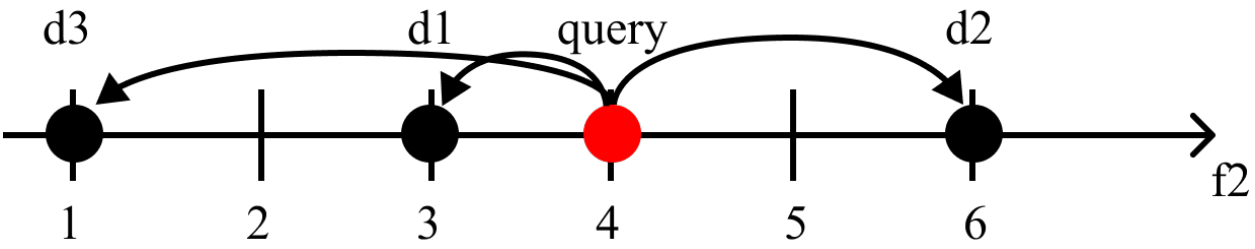

Figure 2.7: Exhaustive Euclidean distance computation between the feature value of query data item (marked red) and feature values of other data items within dataset (marked black)

2.5 Agent-Based Parallelization of MMM

In their paper, called "Agent-Based Parallelization of Multi-Dimensional Semantic Database Model" [16], Alex Li and Dr. Fukuda discuss possibilities of using MASS to implement both parts of MMM, semantic space construction and data retrieval, in order to improve performance. They argue that creating a semantic space and data retrieval are computationally demanding; thus, they propose constructing a space over a cluster system, and have multiple agents explore for a given query data item and its surrounding data items. According to the benchmarks they conducted, compared to a sequential MMM implementation, MASS-based parallelization that they implemented resulted in 22 times speedup while constructing a semantic space, and also reduced the time required for database queries by 23.7%. Although MASS can significantly improve semantic space construction speed, as their work shows, the interest within the scope of this thesis lies in the details of their implementation of data retrieval within MMM using MASS.

As they explain in their work, once the data retrieval starts and the query data item is given, MASS-based version creates agents and propagates them from the location of a query data item over a given semantic space, unlike MMM's exhaustive Euclidean distance calculation between dimensions of the query data item and each data item in dataset (as was shown in Fig. 2.7). Therefore, in the MASS-based version, higher density of data items in space allows agents to collide with any other data items quicker. This can generally be illustrated in Fig. 2.8:

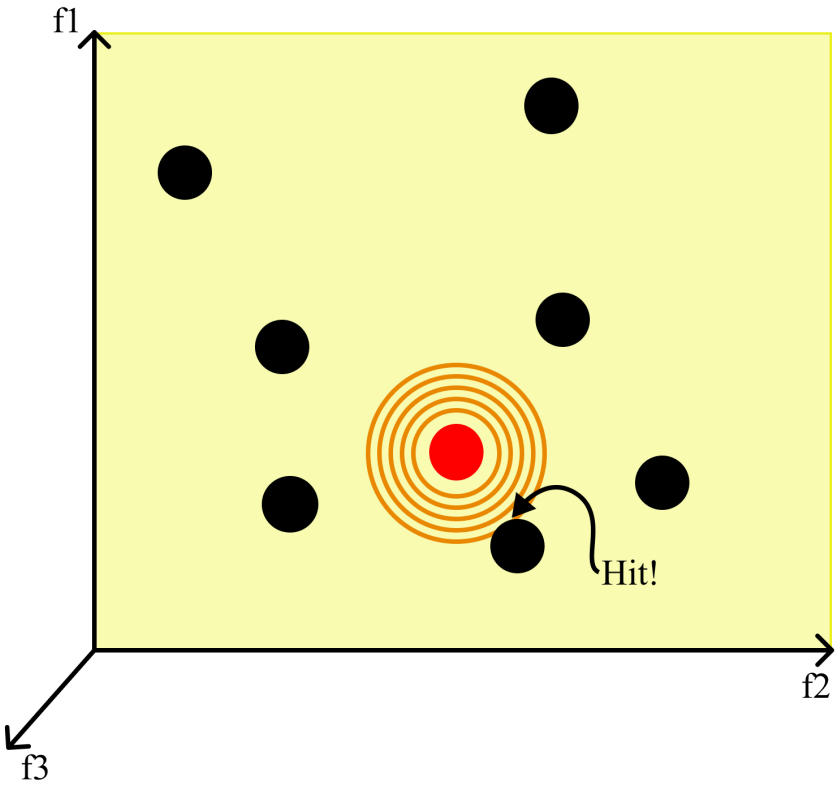

Figure 2.8: MASS-based propagation search in the semantic space

The details of such a search are provided below, step by step:

Step 1. Since an agent needs to propagate in a high-dimensional space, making discrete scales per feature is needed; otherwise, the agent will need to propagate infinitely, as the data item's value can be any number. Providing such discrete scales shrinks down possible migrations and calculations the agent has to do to the number of scales per feature. Therefore, $places[s][f]$ where $f = \#feature$ and $s = \#scales\ per\ feature$ are created.

Step 2. Create and populate agents, each agent starting from $places[i][j]$, where a query data item has feature j with its scale value at $places[i]$. It is not required to spawn an agent per feature, as each agent can cover a range of features k through l where $0 \leq k < l < f$ in order to save up some resources. Note that if one agent covers a specific range of features,

another agent does not cover the same range, so they do not collide.

Step 3. Now, agents move vertically from one scale to another. At each scale, as mentioned in the previous step, agents can cover a range of features, so in one migration, they process several features. The moment an agent encounters a new data item and its value at that cell, the agent calculates the Euclidean distance from this data item to the query data item at this feature. Fig. 2.9, depicts migration of five agents, denoted with purple arrows, along the scale axis, and each agent covering feature where it was deployed and one more feature next to it. Here, red star sign represents some $d_i - th$ value at that feature, and green caret sign represent another $d_k - th$ value at that feature.

Step 4. Agent propagation halts when all agents complete visiting all places or find a given number of top data items.

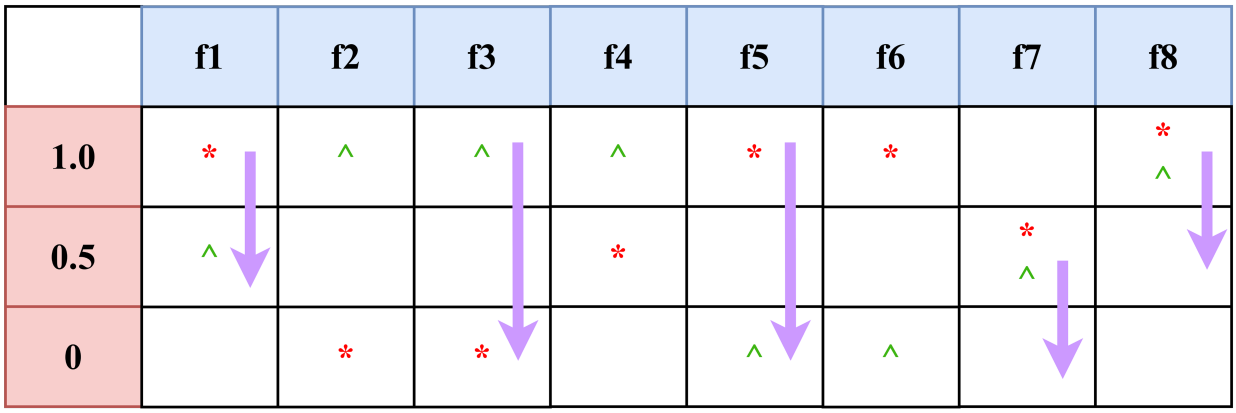

Figure 2.9: MASS-based propagation search along the scale axis, where each arrow represents agent migration, and red star sign and green caret sign represent values of two random data items' values at that feature respectively

The different way of illustrating it would be through the following Fig. 2.10, where scales of some feature $f2$ are represented as points on axis, query value at that feature is marked as red dot, values of other data items are marked as black dots, and red dotted line represents stage/scale at which agent is checking collision at each propagation phase. Once the collision happens, the Euclidean distance is calculated:

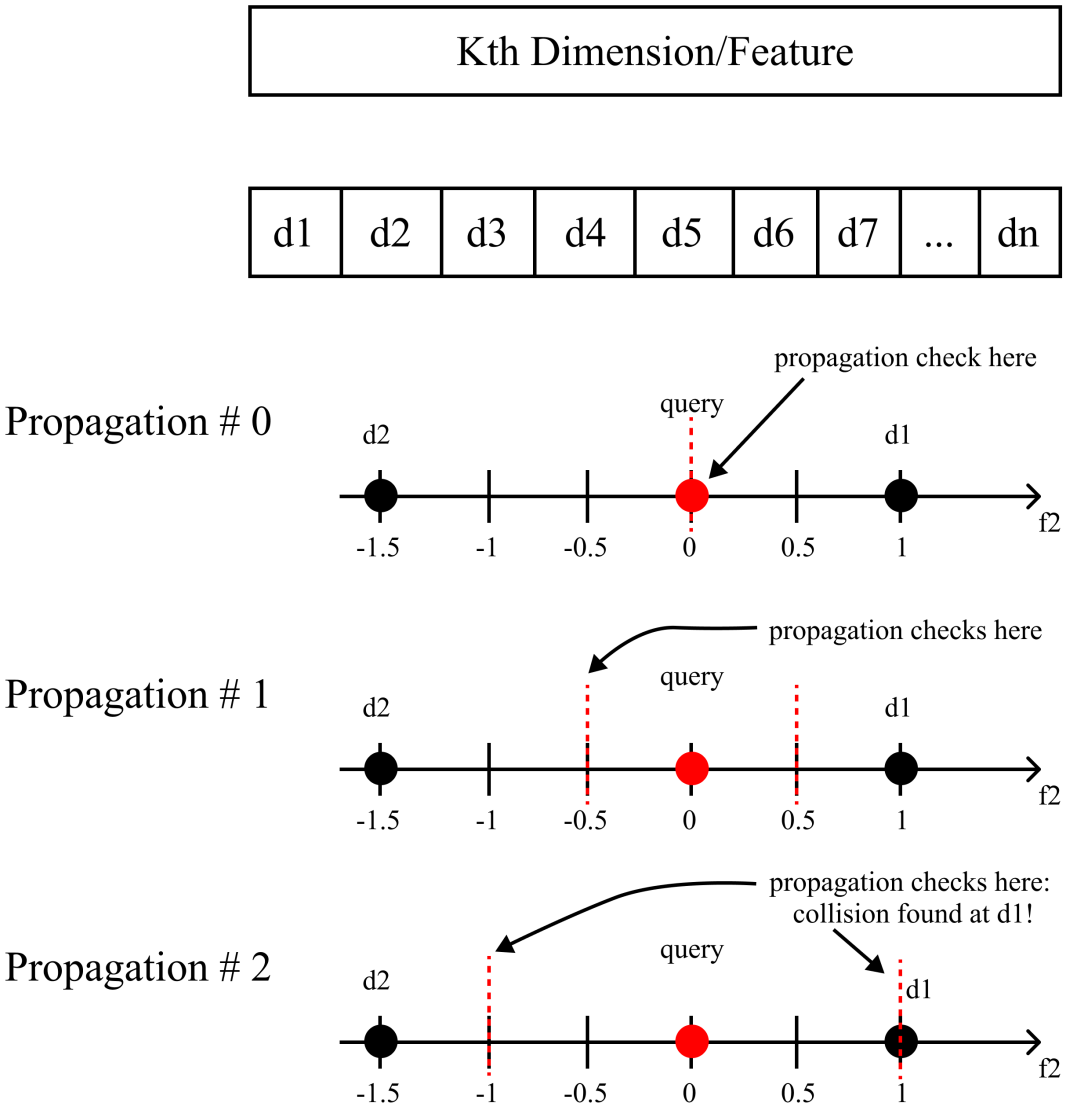

Figure 2.10: Three phases of agent propagation of MASS-based propagation search along the scale axis of some feature $f2$, where red and blacks dots represent values of query data item and dataset data items respectively, and red dotted line represents agent's collision check

Note that in the figure above, there is n number of data items in the given $K-th$ dimension/feature, but only two of them are shown as an example.

Chapter 3

RELATED WORK

In this chapter, the main competitor IVF-based vector search algorithms, IVF Flat and IVFPQ, will be discussed. The descriptive explanation of each will be given, along with a mathematical formulation of the step-by-step execution flow.

3.1 IVF Flat

In the last chapter, IVF index was discussed as a way to optimize dataset vector storage. However, the IVF index itself is simply an index that is not supposed to search for a part on its own. Thus, the naturally deduced vector searching algorithm that uses IVF index alone would be IVF Flat.

3.1.1 General Description

IVF Flat (also referred as IVFF) is a vector search algorithm that uses the IVF index as a way to optimize the storage of vectors given in the dataset, and implements two-phase searching to find the nearest vector for the given query vector:

Phase 1. In the first phase, when a new query vector arrives, the system does not compare it against the entire dataset. Instead, it first compares the query only against the limited number of centroids to find the bin, or bins, that are closest. This selection is controlled by a parameter, n_probe, which determines how many lists to search. This first approximation step drastically reduces the search space from potentially billions of vectors down to a small fraction, providing a massive speed advantage. As an example, three figures are provided below. Fig. 3.1, illustrates the query value, marked as a star, getting placed in the vector space. Fig. 3.2, shows how the query vector computes the Euclidean distance to every centroid, in order to select the n_probe closest ones. Fig. 3.3 represents selecting n_probe closest centroids, and thus regions to query, to the given query vector:

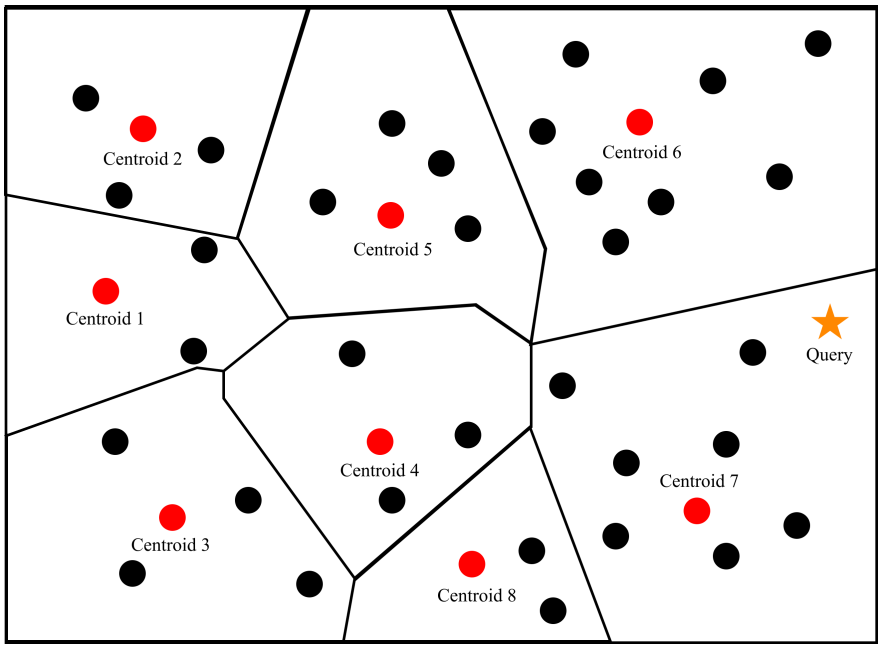

Figure 3.1: Receiving query vector as a point in the vector space, where dataset vectors are shown as black dots, and the query vector as golden start

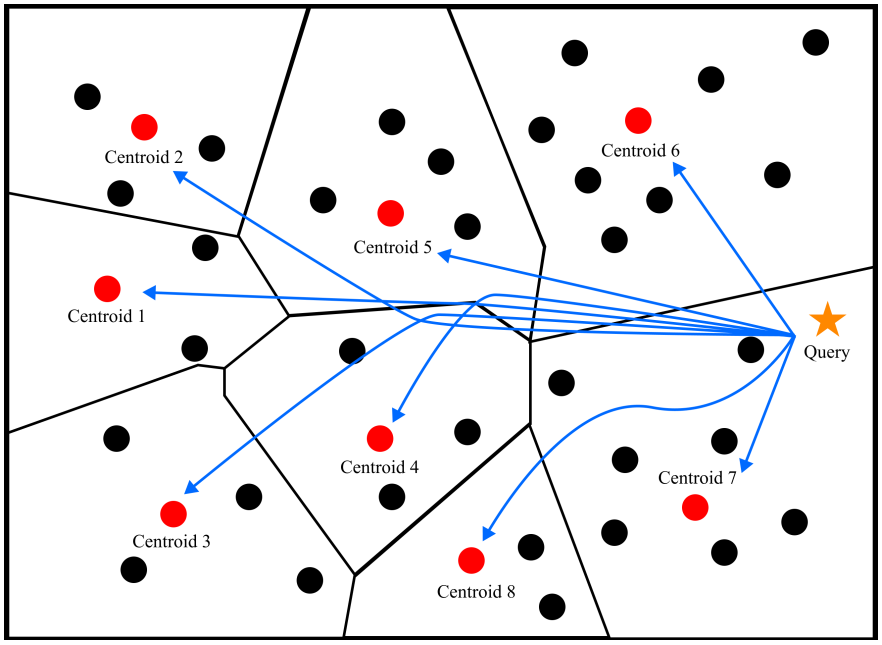

Figure 3.2: Computing the Euclidean distance from the query vector to every centroid to determine the n_probe closest ones

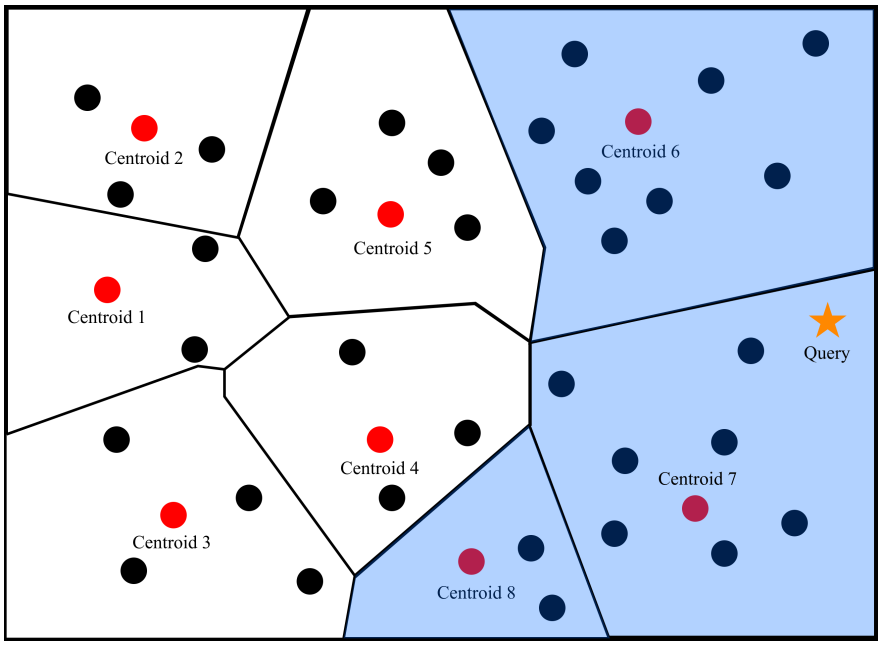

Figure 3.3: Selected n_probe closest clusters, in this case $n\_probe = 3$

Phase 2. In the second phase, the search is exhaustively performed only against the vectors within those selected lists. Fig. 3.4 illustrates the example of searching within one of the selected clusters/regions:

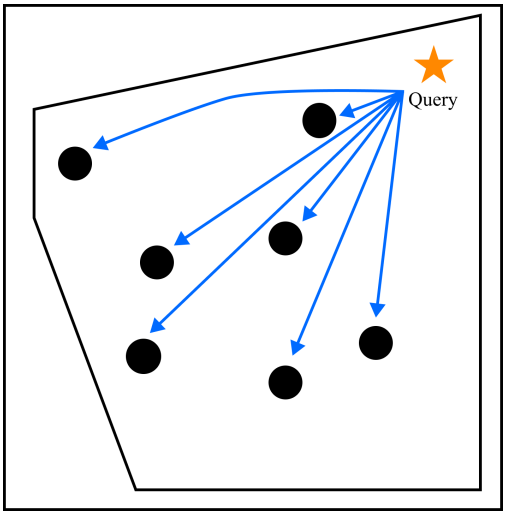

Figure 3.4: Exhaustive search within one of the selected clusters in IVF Flat

IVF Flat is the least complicated algorithm among other IVF-based vector search algorithms, as it simply operates with all the raw original vectors, and performs exhaustive search among the

selected sub-spaces.

3.1.2 Formulation

In turn, the mathematical formulation of step-by-step execution flow of the entire IVF Flat is as follows:

Notation	Meaning
N	Number of vectors
D	Number of dimensions/features in each vector
$X = \{x_i\}_{i=1}^{N},\ x_i \in \mathbb{R}^D$	Dataset
K	Number of coarse clusters/lists
$C = \{\mathrm{c_j}\}_{j=1}^{K},\ c_j \in \mathbb{R}^D$	Centroids/Coarse codebook
$a(i) = arg\min_{j\in\{1,2,\ldots,K\}} \left\|x_i - c_j\right\|^2, \quad a(i) = \{1, 2, \ldots, K\}$	Assignment function
$L_j = \{\, i \mid a(i) = j \,\}$	Inverted list
$q \in \mathbb{R}^D$	Query vector
P	Probe count (n_probe)
k	Number of nearest neighbors to return
$d^2(u, v) = \|u - v\|^2 = \|u\|^2 + \|v\|^2 - 2\,u \cdot v$	Squared Euclidean distance

Table 3.1: IVF Flat Notation Table

Step 1. Building IVF index. This involves using K-Means Clustering algorithm to divide the entire vector space into Voronoi regions, and return a set of centroids, also known as coarse codebook, such that within-cluster sum of squares is minimized.

$\min_{C,a} J(C, a) = \sum_{i=1}^{N} \|x_i - c_{a(i)}\|^2$, where alternating formulas are used:

Assignment: $a(i) = \arg\min_{j\in\{1,2,\ldots,K\}} \|x_i - c_j\|^2$

Centroid update: $c_j = \frac{1}{|L_j|} \sum_{i\in L_j} x_i,\ for\ each\ j$

Step 2. Search - selecting centroids. Once the query vector is provided, compute centroid distances to identify centroids closest to it. Thus, for all $j = 1, 2, \ldots K$ compute:

$\delta_j = \|q - c_j\|^2 = \|q\|^2 + \|c_j\|^2 - 2\,q \cdot c_j$

Then, select the P centroid indices with smallest distances to the query vector:

$S = \text{topP}\left(\{\,\delta_j\,\}_{j=1}^{K}\right)$

Step 3. Search - scan probed clusters/lists. Now, when closest centroids are identified, meaning lists, the query vector needs to find distance from itself to every other vector in this list and then select closest vectors. Therefore, for each computed centroid $j \in S$ and each vector index $i \in L_j$:

$d^2(q, x_i) = \|q - x_i\|^2$

Meanwhile, a global top-k structure is maintained to keep k shortest distances encountered:

$\left\{d^2(q, x_i) \mid i \in \bigcup_{j \in S} L_j\right\}$

The core idea of IVF Flat is to trade perfect accuracy for immense speed by intelligently partitioning the data and only searching the most promising partitions.

3.2 IVFPQ

Although IVF Flat is highly effective, it incurs relatively high memory consumption. To address this limitation, Inverted File with Product Quantization (IVFPQ) was introduced to reduce memory usage while further improving query efficiency.

3.2.1 General Description

The IVFPQ is an efficient ANN vector search algorithm that works the best in huge, high-dimensional datasets by combining IVF indexing with Product Quantization (PQ), which is applied to every vector. The PQ compression allows for speeding up searches by essentially splitting each vector into subvectors and quantizing them, and reducing data size. Thus, IVFPQ still partitions the vector space into K lists/clusters like IVF Flat, but instead of storing vectors in raw form, it stores compact PQ codes of the residuals inside each list. This can be seen in Fig. 3.5 below, where instead of storing the raw form of the vectors, IVFPQ applies the product quantization to store compact PQ codes of the residuals, where pq_dim denotes dimensionality of the quantized vector:

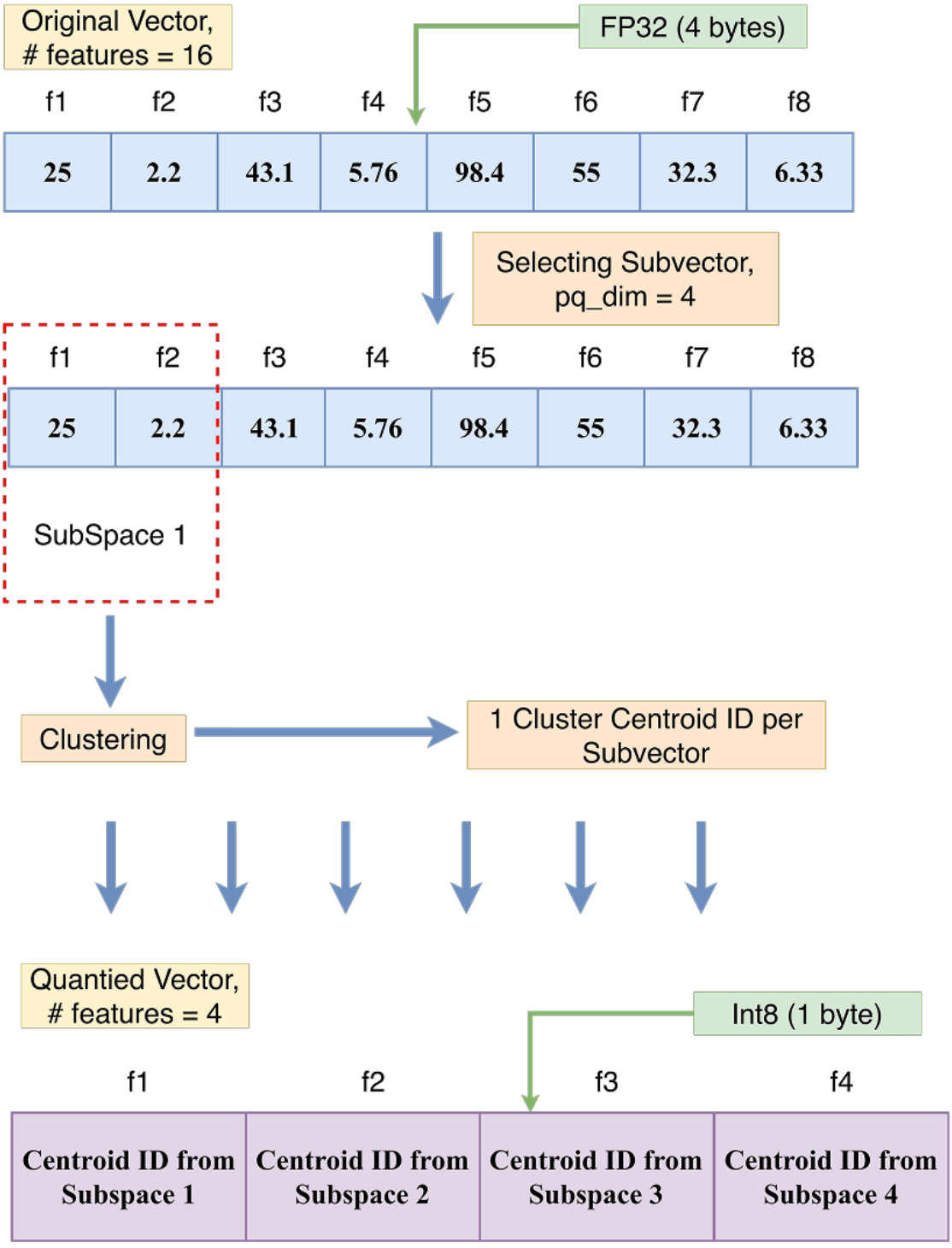

Figure 3.5: General view of applying Product Quantization

In the figure above, Fig. 3.5, each feature value was a floating type FP32, which takes 4 bytes of memory. After the product quantization, the # $features/pq\_dim$ number of feature values is

replaced with one centroid ID, which has Int8 type, and occupies just 1 byte. This allows IVFPQ to achieve significant memory reduction (up to a tenfold decrease) and accelerate search speed, with the compromise of some loss in accuracy.

3.2.2 Formulation

The mathematical formulation of the step-by-step execution of the entire IVFPQ is as follows:

Notation	Meaning
N	Number of vectors
D	Number of dimensions/features in each vector
$X = \{x_i\}_{i=1}^{N},\ x_i \in \mathbb{R}^D$	Dataset
K	Number of coarse clusters/lists
$C = \{\mathrm{c_j}\}_{j=1}^{K},\ c_j \in \mathbb{R}^D$	Centroids/Coarse codebook
$a(i) = arg\min_{j \in \{1,2,\ldots,K\}} \left\|x_i - c_j\right\|^2, \quad a(i) = \{1, 2, \ldots, K\}$	Assignment function
$L_j = \{\, i \mid a(i) = j \,\}$	Inverted list
M	Number of PQ subspaces/blocks (pq_dim)
$d = D/M$	Dimension of each subvector
K_s	Number of codewords per subspace
$r_i = x_i - c_{a(i)}$	Residual of x_i relative to its centroid
$\mathrm{r}_i^m,\ m = 1, 2, \ldots, K$	Residual of r_i's $m - th$ subvector
$Q_m = \{q_{m,1}, q_{m,2}, \ldots, q_{m,K_s}\}$	Codebooks of m's subspace
$b_{i,m} \in \{1, 2, \ldots, K_s\}$	PQ code index for r_i^m, meaning codeword it uses
$q \in \mathbb{R}^D$	Query vector
$r_{q,j} = q - c_j$	Residual of query vector q relative to centroid c_j
P	Probe count (n_probe)
k	Number of nearest neighbors to return
$\mathrm{LUT}_{j,m}[k]$	Lookup table (LUT) value, meaning distance between $r_{q,j}^m$ and codeword $q_{m,k}$
$\widetilde{d^2}(q, x_i)$	Approximate distance computed using Asymmetric Distance Computation (ADC), which is sum of LUT entries for code indices

Table 3.2: IVFPQ Notation Table

Step 1. Building IVF index. This involves using K-Means Clustering algorithm to divide the entire vector space into Voronoi regions, and return a set of centroids, also known as coarse codebook, such that within-cluster sum of squares is minimized.

$\min_{C,a} J(C,a) = \sum_{i=1}^{N} \|x_i - c_{a(i)}\|^2$, where alternating formulas are used:

Assignment: $a(i) = arg \min_{j \in \{1,2,\ldots,K\}} \left\|x_i - c_j\right\|^2,\ a(i) = \{1,2,\ldots,K\}$

Centroid update: $c_j = \frac{1}{|L_j|} \sum_{i \in L_j} x_i,\ for\ each\ j$

Step 2. Training PQ on residuals. For each vector compute residual to its assigned centroid:

$r_i = x_i - c_{a(i)}$

Then, split each residual into M contiguous subvectors:

$r_i = [\, r_i^1,\ r_i^2,\ \ldots,\ r_i^M \,],\ r_i^m \in \mathbb{R}^d,\ d = D/M$

Now, for each subspace $m = 1, 2, \ldots, M$, run K-Means Clustering to get sub-codebook:

$Q_m = \{q_{m,1}, q_{m,2}, \ldots, q_{m,K_s}\}$

Step 3. Encode dataset. For every vector , compute code index for each of its subspaces:

$b_{i,m} = \arg\min_{k \in \{1..K_s\}} \|r_i^m - q_{m,k}\|^2$

Once done, store the compact code $(b_{i,1}, b_{i,2}, \ldots, b_{i,M})$ inside list L_j. Also, optionally, IVFPQ allows to keep original raw form of every vector x_i, so that it uses it for refining the results after the search is done.

Step 4. Search - selecting centroids. Once the query vector is provided, compute centroid distances to identify centroids closest to it. Thus, for all $j = 1, 2, \ldots K$ compute:

$\delta_j = \|q - c_j\|^2 = \|q\|^2 + \|c_j\|^2 - 2\, q \cdot c_j$

Then, select the P centroid indices with smallest distances to the query vector:

$S = \text{topP}\left(\{\, \delta_j \,\}_{j=1}^{K}\right)$

Step 5. Search - computing query residual and LUTs. For each probed centroid $j \in S$, compute query residual and LUTs. First, compute query residual relative to centroid j:

$r_{q,j} = q - c_j,\ r_{q,j} = [\ r^1_{q,j},\ r^2_{q,j},\ \ldots,\ r^M_{q,j}\ ]$

Then, for each subspace build a LUT of distances between the query residual subvector and all sub-codewords:

$\mathrm{LUT}_{j,m}[w] = \|r^m_{q,j} - q_{m,w}\|^2,\ w = 1..K_s$

Step 6. Search - ADC for each coded vector. Now, for each coded vector, given stored code $b_{i,[1,2,\ldots,M]}$, calculate approximate squared distance:

$\widetilde{d^2}(q, x_i) = \sum_{m=1}^{M} \mathrm{LUT}_{j,m}\left(b_{i,m}\right)$

Meanwhile, a global top-k structure is maintained to keep k shortest distances encountered.

Step 7. (Optional) Refining results. In case it was chosen to keep the original raw form of vectors, they can be used to refine, or re-rank, the obtained results to increase accuracy, by recomputing exact distances for top-k candidates:

$d^2(q, x_i) = \|q - x_i\|^2$

Chapter 4

IVF SINGULAR SEARCH

This chapter discusses the new proposed algorithm, IVF Singular Search. It includes the general explanation of the algorithm, as well as the mathematical formulation.

4.1 General Description

IVF Singular Search (IVFSS), or Inverted File with Singular Search, is the ANN vector search algorithm developed during the course of this thesis work. Since generally IVF-based search algorithms work better when implemented on GPU, IVFSS was also implemented for GPUs with GPU-first in mind.

The IVFSS works similarly to IVF Flat, also splitting the vector space into Voronoi regions using the IVF index; however, it searches for the nearest neighbors within each cluster/list differently. This search part of IVFSS was inspired by the way MMM transposes the original data matrix, and then queries each row, meaning grouped features, trying to find the nearest one to the query's feature. It was also inspired by the way MASS Agents and Places were used to query MASS-based improved version of MMM in Alex Li's work, with the difference that MASS CUDA was utilized instead of the standard MASS, as a way to accelerate the implementation process.

However, unlike in the standard MMM, where the search goes exhaustively for each data item, or in MASS-based MMM, where gradual propagation is used, in IVFSS, searching supposes having sorted rows, meaning arrays of features/dimensions, and then performing the Binary Search within each feature array using the query's feature value to find the closest one. In this scenario, MASS Place represents each feature array, and Agent represents the query's feature value. Then, all the obtained indices of the closest vectors are aggregated and sorted by number of occurrences, to choose the one that was selected the most, meaning the closest one. At last, the squared distance between the original raw form of the closest vector from each list and the query vector gets computed, and the closest neighbor is identified. Note that the raw original form of dataset

vectors can be obtained by rebuilding them using information stored in the list, or just by storing the original dataset; in the standard version, the former is implemented. Therefore, due to the specifics of the searching technique, IVFSS is capable of returning only one nearest neighbor.

4.2 Formulation

The mathematical formulation of step-by-step execution of the entire standard IVFSS, without MASS, is as follows:

Notation	Meaning
N	Number of vectors
D	Number of dimensions/features in each vector
$X = \{x_i\}_{i=1}^{N},\ x_i \in \mathbb{R}^D$	Dataset
K	Number of coarse clusters/lists
$C = \{\mathrm{c_j}\}_{j=1}^{K},\ c_j \in \mathbb{R}^D$	Centroids/Coarse codebook
$a(i) = arg\min_{j \in \{1,2,\ldots,K\}} \left\|x_i - c_j\right\|^2, \quad a(i) = \{1, 2, \ldots, K\}$	Assignment function
$L_j = \{\, i \mid a(i) = j \,\}$	Inverted list
$size(L_j)$	Number of vectors in a list L_j
$Z[size(L_j)][D]$	Data inside of list L_j
$q \in \mathbb{R}^D$	Query vector
P	Probe count (n_probe)
$I_{L_j} = \{y_i \mid y_i = index(x_h)\}_{i=1}^{D},\ h \in 1, 2, \ldots, N$	Intermediary result array of some list L_j
$d^2(u, v) = \|u - v\|^2 = \|u\|^2 + \|v\|^2 - 2\,u \cdot v$	Squared Euclidean distance

Table 4.1: IVF Singular Search Notation Table

Step 1. Building IVF index. This involves using K-Means Clustering algorithm to divide the entire vector space into Voronoi regions, and return a set of centroids, also known as coarse codebook, such that within-cluster sum of squares is minimized.

$\min_{C,a} J(C, a) = \sum_{i=1}^{N} \|x_i - c_{a(i)}\|^2$, where alternating formulas are used:

Assignment: $a(i) = \arg\min_{j \in \{1,2,\ldots,K\}} \|x_i - c_j\|^2$

Centroid update: $c_j = \frac{1}{|L_j|} \sum_{i \in L_j} x_i, \ for\ each\ j$

Step 2. Transpose data inside lists. Now, each list L_j has dataset of vectors within, so it needs to be transposed before the search starts, Z^T

Step 3. Search - selecting centroids. Once the query vector is provided, compute centroid distances to identify centroids closest to it. Thus, for all $j = 1, 2, \ldots K$ compute:

$\delta_j = \|q - c_j\|^2 = \|q\|^2 + \|c_j\|^2 - 2\, q \cdot c_j$

Then, select the P centroid indices with smallest distances to the query vector:

$S = \text{topP}\left(\{\, \delta_j \,\}_{j=1}^{K}\right)$

Step 4. Search - performing the Binary Search. Once top closest P centroids have been selected, within each selected list L_j, where $j \in S$, on GPU it allocates D number of threads, so that each thread runs the Binary Search using $i - th$ feature/dimension of the query vector on that row of Z^T, essentially against the i-th feature/dimension of each dataset vector in that list. At the end, for each list L_j, there will be an array I_{L_j} of size D, where indices of the closest vectors for each feature/dimension are stored

Step 5. Search - aggregation. Once the closest vectors for each feature/dimension of the query vector are identified and stored in an array I_{L_j}, it will result in having a P number of such arrays. Therefore, in order to find the closest vector, for each I_{L_j}, first, will count the number of occurrences of each vector index in that array, and then sort them in descending order. After that, the vector with the most occurrences in I_{L_j} will be selected:

$For\ each\ of\ P\ selected\ lists$:

$OC = countOccurrences(I_{L_j})$

$ST = sortByOccurrences(OC)$

$MX_{L_j} = max(ST)$

Now, it will get raw original forms of the P selected vectors MX_{L_j} (using either of two methods described earlier), and calculate the Euclidean squared distance between each one of them and the original query vector:

$For\ each\ of\ P\ selected\ closest\ vectors$:

$V_{L_j} = fetchOriginalForm(MX_{L_j})$

$\delta_j = \|q - V_{L_j}\|^2 = \|q\|^2 + \|V_{L_j}\|^2 - 2\, q \cdot V_{L_j}$

Thus, after getting the Euclidean distance to the each selected MX_{L_j}, it finds one closest vector by selecting the one with the shortest distance $min(\delta_{j \in 1,2,\ldots,K})$.

Chapter 5

IMPLEMENTATION

In this chapter, IVFSS's overall system design of the search function will be demonstrated using examples and visuals to describe them. Moreover, it will explain the codebase that was written using C/C++ and CUDA C++, mentioning the most important functions and sections.

5.1 System Design of the Search Function

As mentioned in the previous chapter, IVFSS starts off the same way as IVF Flat; however, the main difference lies in the way the searching is performed. Thus, the step-by-step example below demonstrates specifically how the search function works:

Step 1. Assume, among the P (n_probe) number of all closest centroids/lists, this list was randomly chosen, and inside it has 100 vectors, each with 2048 features/dimensions:

	Dimension 1	Dimension 2	Dimension 3	Dimension 4	Dimension 5	Dimension ...	Dimension 2048
Vector 1	1.2423222	60.232375	4.042965	28.082372	96.023765	...	1.974210
Vector 2	2.184246	0	20.999156	31.566212	7.353532	...	5.616991
Vector 3	20.959776	4.342215	0	23.426604	41.824656	...	13.466675
Vector ...	...	...	...	...	...	...	...
Vector 100	27.724629	0	0	7.396680	9.141179	...	0

Figure 5.1: Dataset of 100 vectors, each with 2048 features/dimensions

Step 2. Before the actual search starts, this dataset gets transposed:

	Vector 1	Vector 2	Vector 3	Vector ...	Vector 100
Dimension 1	**1.2423222**	**2.184246**	**20.959776**	**...**	**27.724629**
Dimension 2	**60.232375**	**0**	**4.342215**	**...**	**0**
Dimension 3	**4.042965**	**20.999156**	**0**	**...**	**0**
Dimension 4	**28.082372**	**31.566212**	**23.426604**	**...**	**7.396680**
Dimension 5	**96.023765**	**7.353532**	**41.824656**	**...**	**9.141179**
Dimension ...	**...**	**...**	**...**	**...**	**...**
Dimension 2048	**1.974210**	**5.616991**	**13.466675**	**...**	**0**

Figure 5.2: Transposed dataset of 100 vectors, each with 2048 features/dimensions

As one can see, now rows are dimensions, and columns are vectors.

Step 3. After transposing the dataset, each row needs to be sorted in ascending order, so that the Binary Search can be performed in the future:

	Vector 1	Vector 2	Vector 3	Vector ...	Vector 100
Dimension 1	**1.2423222**	**2.184246**	**20.959776**	**...**	**27.724629**
Dimension 2	**60.232375**	**0**	**4.342215**	**...**	**0**
Dimension 3	**4.042965**	**20.999156**	**0**	**...**	**0**
Dimension 4	**28.082372**	**31.566212**	**23.426604**	**...**	**7.396680**
Dimension 5	**96.023765**	**7.353532**	**41.824656**	**...**	**9.141179**
Dimension ...	**...**	**...**	**...**	**...**	**...**
Dimension 2048	**1.974210**	**5.616991**	**13.466675**	**...**	**0**

Figure 5.3: Selecting a row from the transposed dataset

In this case, the figure demonstrates how the values of vectors at dimension 1 were chosen to get sorted.

Step 4. As every row gets sorted, vector indices change with the values, and the structure of each cell should be explained here. Each cell in the row can be seen as a tuple, containing both value and vector index, so that it is no longer a single column that decides, e.g. [value, vector index]. This structure allows for freely sorting the values, and also builds back the original raw form of the vector in the future:

	Vector 5	Vector 1	Vector 2	Vector ...	Vector 41
Dimension 1	**0.3331689**	**1.2423222**	**2.184246**	**...**	**86.316675**

Figure 5.4: Sorting selected row

As one can see, values at dimension 1 are now sorted.

Step 5. Keep sorting in this way for every row/dimension:

	Vector 5	Vector 1	Vector 2	Vector ...	Vector 41
Dimension 1	**0.3331689**	**1.2423222**	**2.184246**	**...**	**86.316675**

	Vector 2	Vector 100	Vector 1	Vector ...	Vector 18
Dimension 2	**0**	**0**	**4.042965**	**...**	**34.427765**

For all 2048 Dimensions/Features

	Vector 100	Vector 1	Vector 88	Vector ...	Vector 65
Dimension 2048	**0**	**1.974210**	**1.985148**	**...**	**91.523342**

Figure 5.5: Sorting all rows

In this case, all 2048 dimensions are sorted.

Step 6. As mentioned in the last chapter, IVFSS's search function was implemented using MASS CUDA. Therefore, in IVFSS, each dimension/row gets represented by a Place, and each

Agent, which will be dispatched to its own Place, will carry a value of the query vector specifically for that dimension. Note that the number of Agents is equal to the number of Places in this case:

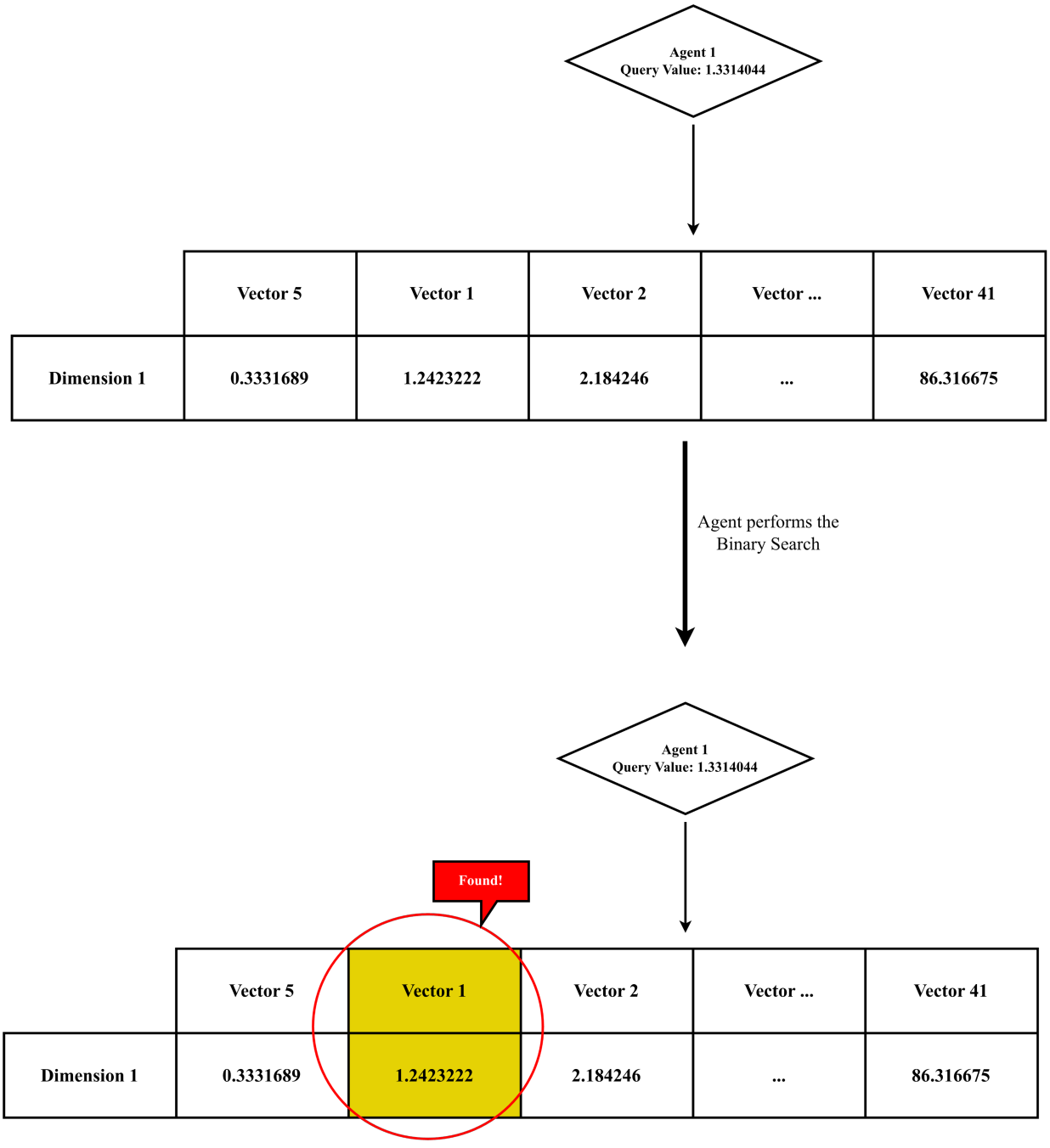

Figure 5.6: Agent getting dispatched with query value to the place and performing the Binary Search

Once Agent gets dispatched to the Place, it starts performing the Binary Search on the data available at that Place. Here, it is found that ‘Vector 1‘ has the closest value to the query value that ‘Agent 1‘ is carrying. This step is done for all other 2047 dimensions/places, using 2047 more agents.

Step 7. The end result of the previous step is an array of 2048 vector indices that were selected as the closest to the query vector’s value at that respective dimension:

Closest Vector ID is retrieved for each dimension/feature

Dimension 1	**Vector 1**
Dimension 2	**Vector 54**
Dimension 4	**Vector 22**
Dimension ...	**Vector ...**
Dimension 2048	**Vector 45**

Figure 5.7: Retrieval of the closest vector index from each dimension

Step 8. Now, this array needs to be used to count the number of occurrences of each vector:

Aggregate Vector IDs by Number of Occurrences

Vector 2	**26**
Vector 1	**15**
Vector 22	**8**
Vector ...	**Vector ...**
Vector 32	**0**

Figure 5.8: Aggregation by the vector index

In the figure above, 'Vector 2' has occurred 26 times, making it the most probable nearest neighbor in this list for the given query vector. However, there are $P - 1$ more of such vectors from $P - 1$ other lists, and in order to find the closest one, the Euclidean square distance is calculated between the query vector and each one of them.

5.2 Code Logic

Now, coming to the programming part, the program itself starts with $main()$, and sets up variables to build the IVF index. Then, $main()$ launches $ivf\_search()$, which in turn starts selecting clusters on the GPU using $selectClusters()$. Once the closest clusters are selected, $ivf\_search()$ calls the lambda function $create\_agents()$, which will spawn MASS CUDA agents. Then, it calls them using $call\_all\_agents()$, which invokes $callMethod()$ of each agent, and thus runs the Binary Search on each place. Once finished, it runs $cudaSynchronize()$ to synchronize all threads, and calls $aggregation()$ to aggregate the results. The result of the $aggregation()$ is the actual result of the program. Below is Fig. 5.9, which is the program running sequence of IVFSS implementation

using MASS CUDA:

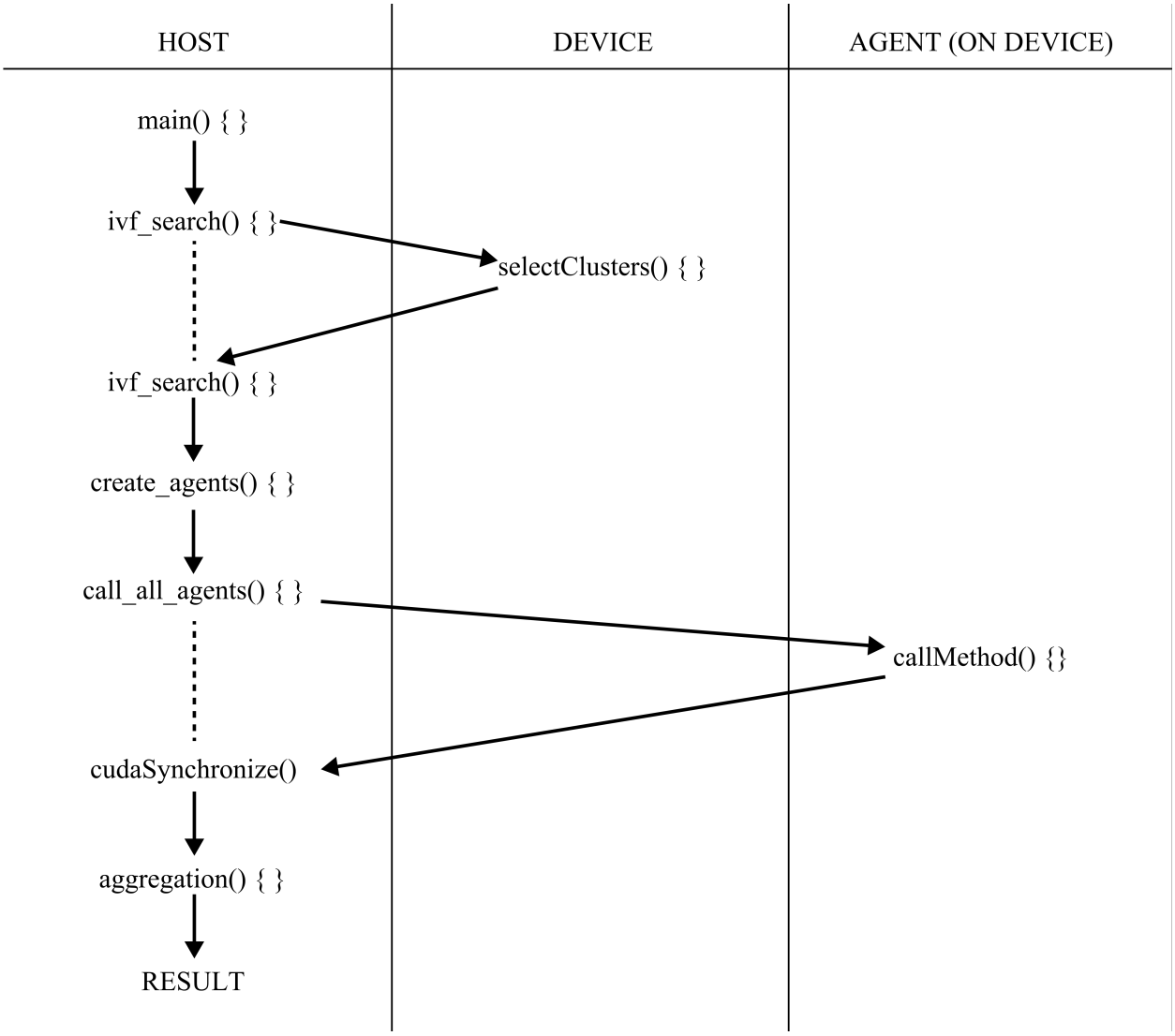

Figure 5.9: IVFSS Program Running Sequence

During the implementation of the IVFSS code, the CUDA Vector Search (cuVS) [17] library developed by NVIDIA Corporation was employed to construct the IVF index and to support several fundamental operations. The library additionally provided the reference implementations for both IVF Flat and IVFPQ.

Chapter 6

RESULTS

In this chapter, the results of the IVFSS implementation and execution will be provided for various vector datasets, as well as comparing them with the results of IVF Flat and IVFPQ.

6.1 Setup

First, we go through the initial setup of the system and specifications of the evaluation metrics.

6.1.1 Hardware Specifications

For the purposes of benchmarking, one machine with the following specifications was used:

Item	Specification
Architecture	x86_64
Byte Order	Little Endian
CPU(s)	16
Thread(s) per Core	2
Vendor ID	AuthenticAMD
Model Name	AMD EPYC 7232P 8-Core Processor

Table 6.1: Machine Specifications Table

On board, this machine had one GPU with the following specifications:

Item	Specification
Name	NVIDIA RTX A5000
CUDA Version	12.8
Available Memory	24564 MegaBytes
NVIDIA-SMI:	570.86.10
Driver Version	570.86.10

Table 6.2: GPU Specifications Table

6.1.2 Evaluation Metrics

The table below outlines the metrics used to assess algorithms' performance:

Metric	Definition
Speed	The total time it took for an algorithm to execute from start to end in milliseconds (ms)
Maximum Memory	The maximum amount of memory, in megabytes (MB), that was utilized while algorithm was running
Hit@k	Measures whether at least one relevant item appears within the top-k retrieved results for a query. It is binary for each query: $Hit@k = \begin{cases} 1, & \text{if a correct item is in top-k} \\ 0, & \text{otherwise} \end{cases}$
First-Hit Rank (FHR)	The first index at which the retrieval list contains a correct result. Defined as: $k^* = \min\{ k \mid \text{Hit@k} = 1 \}$

Table 6.3: Evaluation Metrics Table

6.2 Performance Analysis

6.2.1 Dataset 1: Imagenetood 32K Dataset

The first dataset is *imagenetood* image dataset, that was converted to the vector dataset, containing 31802 vectors, each having 2048 dimensions, using ResNet-50 [18], which is a convolutional neural network (CNN) that excels at image classification.

For the benchmarking stage of this dataset, the parameter n_probe was set to 50, while the top-k retrieval depth was fixed at 10, and remained constant across all datasets and all runs (except for IVFSS, as it returns only one result).

For IVFPQ-specific configurations, the parameters pq_bits and pq_dim were fixed at 8 and 32, respectively, throughout all benchmarks. The parameter pq_bits specifies the number of bits used

to encode each sub-vector centroid index, while pq_dim denotes the dimensionality of the quantized vector after decomposing the original high-dimensional vector during product quantization.

Metric 1. Speed:

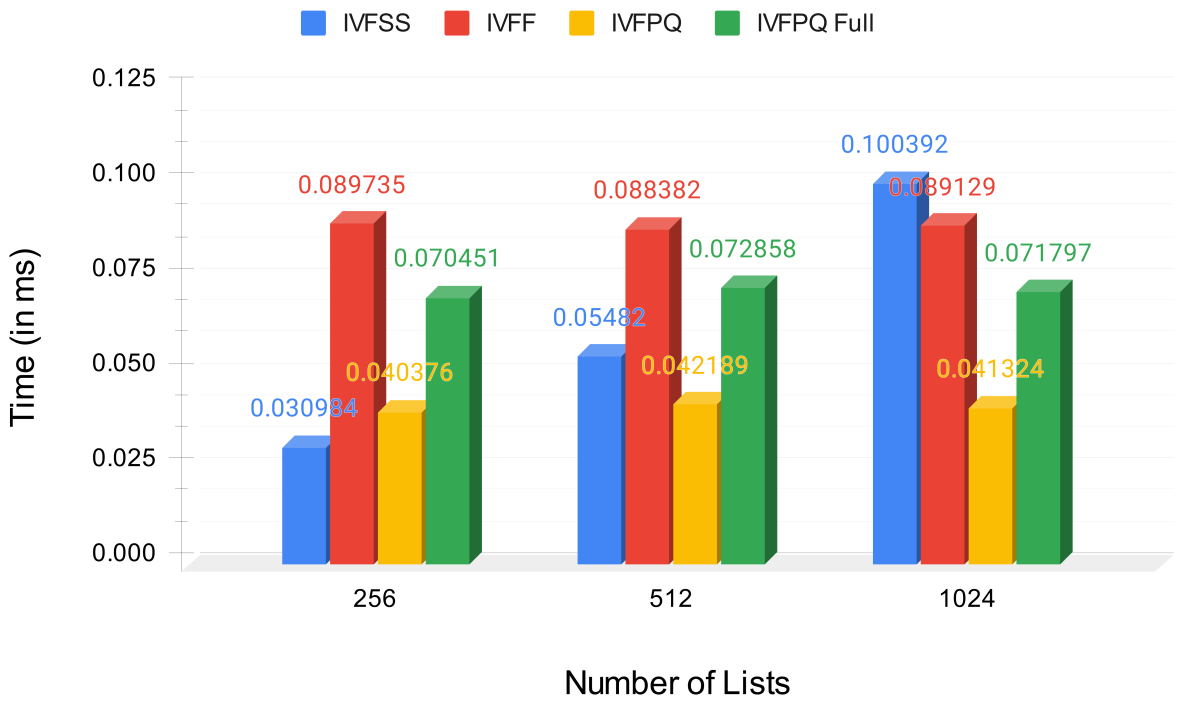

Figure 6.1: Time it takes for each algorithm to perform on Dataset 1

The speed of IVFSS linearly increases with the number of lists, as the centroids that were computed need to be copied from the GPU to the host, and it takes most of the execution time. Nevertheless, for 256 lists, it shows the best performance compared to all other algorithms. Note that IVFPQ Full simply means the time taken by IVFPQ, added to the time it takes for the IVFPQ's re-ranking step to produce more accurate results. When the number of lists reaches 1024, IVFSS actually performs the worst, where copying cluster centroids takes up 90% of all time. Due to its structure, IVFPQ keeps almost a constant execution speed, regardless of the number of lists defined.

Metric 2. Memory:

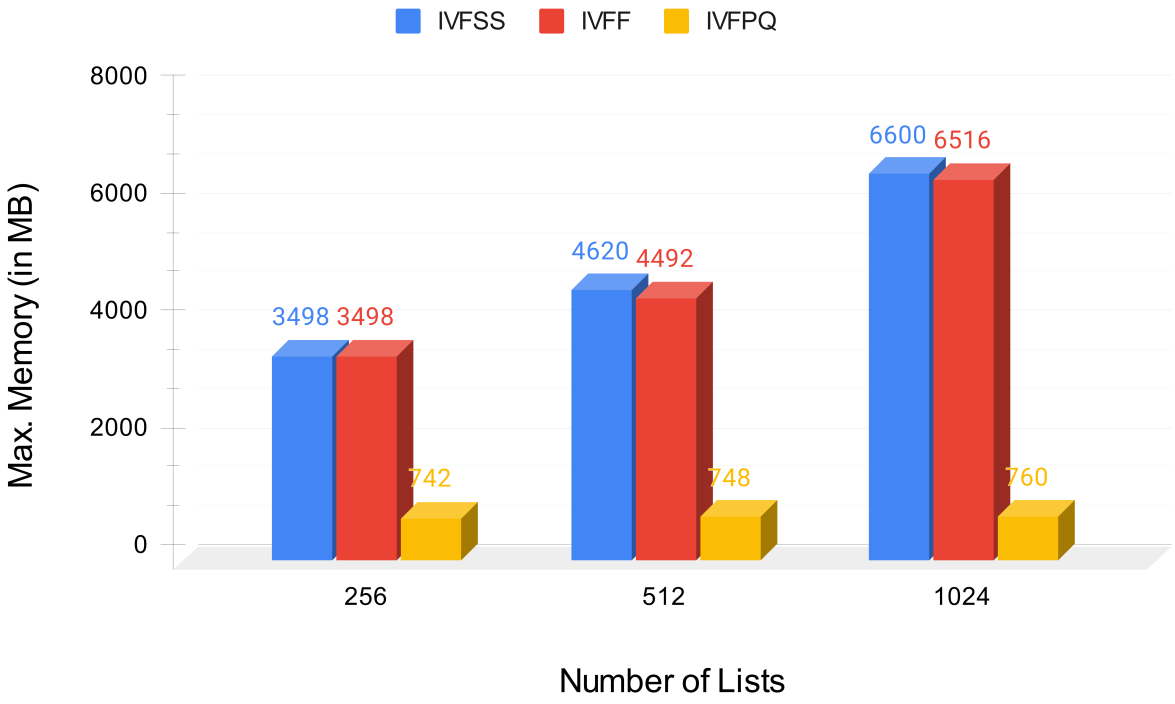

Figure 6.2: Maximum Memory it takes for each algorithm to perform on Dataset 1

The maximum memory utilized by the IVFSS is roughly the same as for IVF Flat, and considerably more compared to the IVFPQ. This makes sense, as IVFPQ applies the product quantization, which allows it to store compact versions of vectors and save up more space.

Metric 3. First-Hit Rank:

Algorithms \ Number of Lists	256	512	1024
IVFSS	1	1	1
IVFF	1	1	1
IVFPQ	1	1	2
IVFPQ Full	1	1	1

Figure 6.3: First-Hit Rank of each algorithm on Dataset 1

The First-Hit Rank represents the position within the returned top-k list at which the correct vector index appears (top-k is always 1 for IVFSS, due to its structure). As shown in the results, nearly all algorithms performed well in this metric, with the exception of IVFPQ at 1,024 lists, where the correct index appeared at the second position. These observations indicate that with 256 lists, IVFSS achieved the best overall performance in both speed and accuracy, though it was slightly outperformed by IVFPQ when the number of lists increased to 512.

6.2.2 *Dataset 2: Tiny-Imagenet-200 10K Dataset*

The second dataset is the $tiny - imagenet - 200$ image dataset, which was converted to the vector dataset, containing 10000 vectors, each having 2048 dimensions, using ResNet-50. Again, n_probe was equal to 50 for this dataset's benchmarking.

Metric 1. Speed:

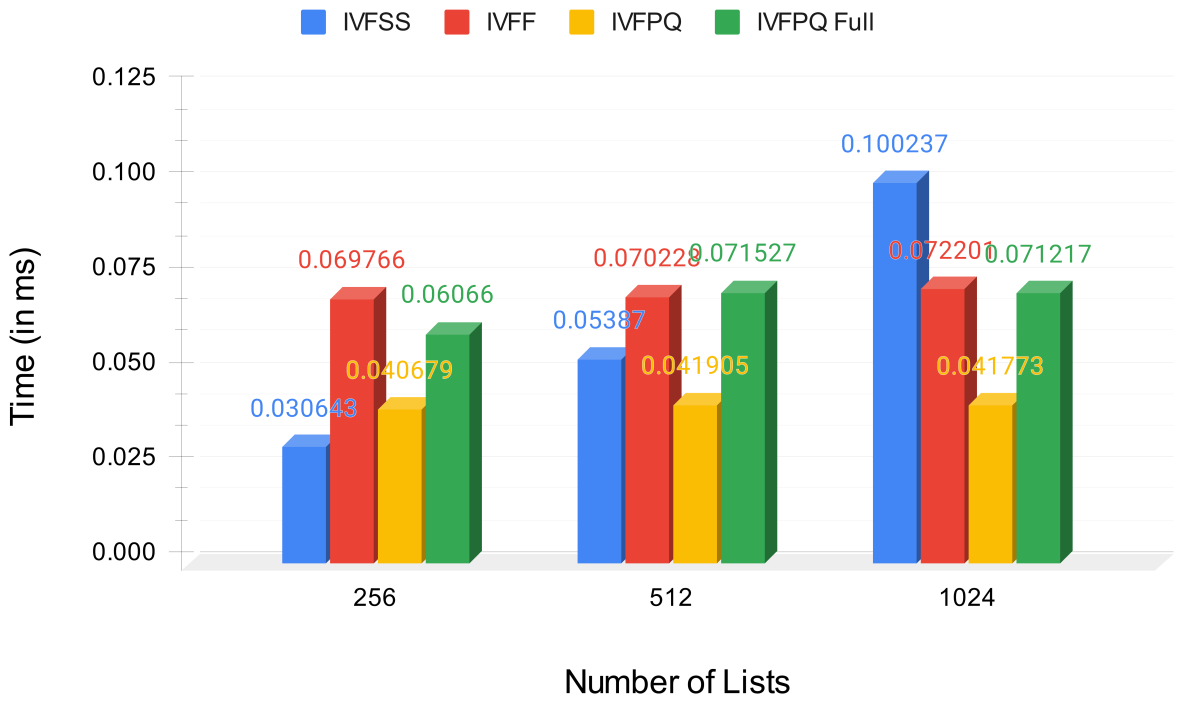

Figure 6.4: Time it takes for each algorithm to perform on Dataset 2

Here, the pattern for IVFSS is similar to what it had with the previous dataset. However, since IVF Flat seems to be tuned better, it performs nearly at the same speed as IVFPQ Full. Similarly to the previous dataset, IVFPQ performs at a constant time, regardless of the number of lists.

Metric 2. Memory:

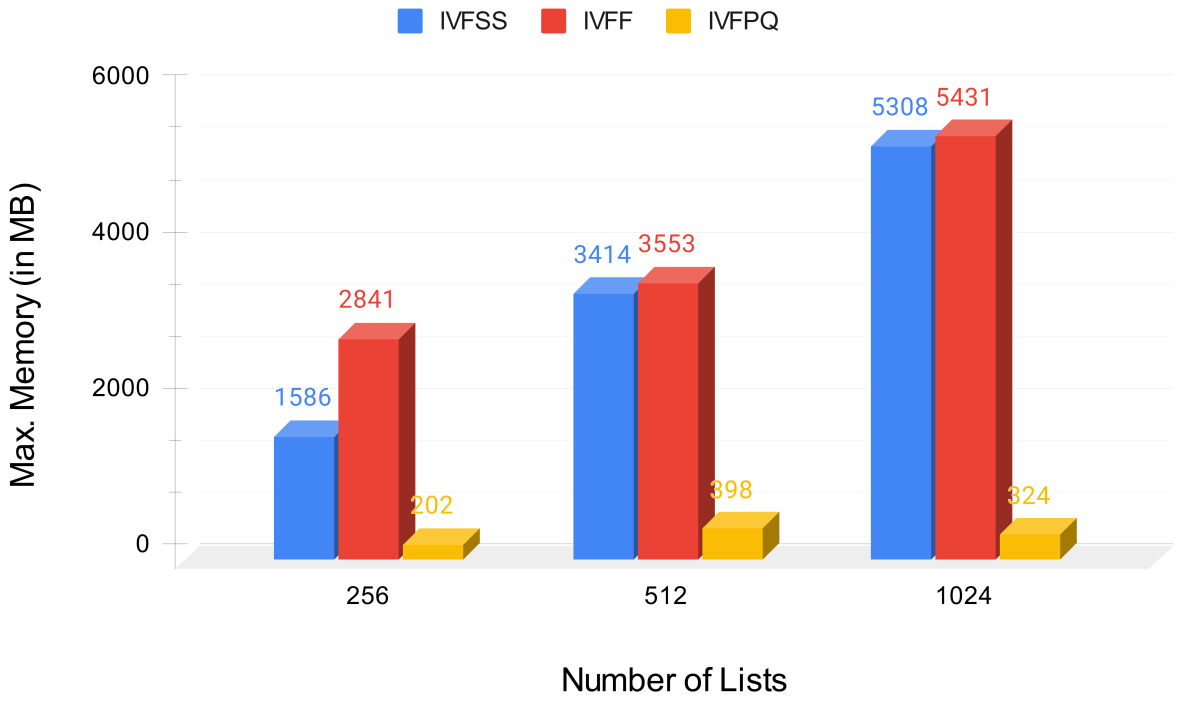

Figure 6.5: Maximum Memory it takes for each algorithm to perform on Dataset 2

Again, IVFSS utilizes almost the same amount of memory as IVF Flat throughout the benchmarking cycle. However, unlike in the previous dataset, for 256 lists, IVFSS utilizes almost half as much memory as IVF Flat. IVFPQ maintains leadership by showing a consistently high utilization rate.

Metric 3. First-Hit Rank:

Algorithms \ Number of Lists	256	512	1024
IVFSS	1	1	1
IVFF	1	1	1
IVFPQ	1	>10	>10
IVFPQ Full	1	>10	>10

Figure 6.6: First-Hit Rank of each algorithm on Dataset 2

IVFSS and IVF Flat achieved perfect performance on this dataset across all evaluated list sizes, consistent with their behavior on the other datasets. In contrast, IVFPQ and its refined variant, IVFPQ Full, produced satisfactory results at 256 lists but failed entirely at 512 and 1,024 lists, where the correct vector index did not appear within the top-k; therefore, their FHRs are shown as > 10. These findings indicate that IVFPQ is highly sensitive to dataset characteristics and parameter settings, requiring careful tuning to achieve reliable performance.

6.2.3 Dataset 3: GTZAN/Blues 100 Dataset

The third dataset is the $GTZAN$ audio dataset, specifically the $blues$ genre, that was converted to the vector dataset, containing 100 vectors, each having 512 dimensions, using OpenL3 [19][20].

For the sake of testing, this dataset was intentionally chosen to represent different kinds of information and be smaller compared to the previous two datasets. Therefore, the figures with results for this dataset have their x-axis/row defined as "Number of Lists / n_probe", allowing the adjustment of the n_probe parameter along with "Number of Lists".

Metric 1. Speed:

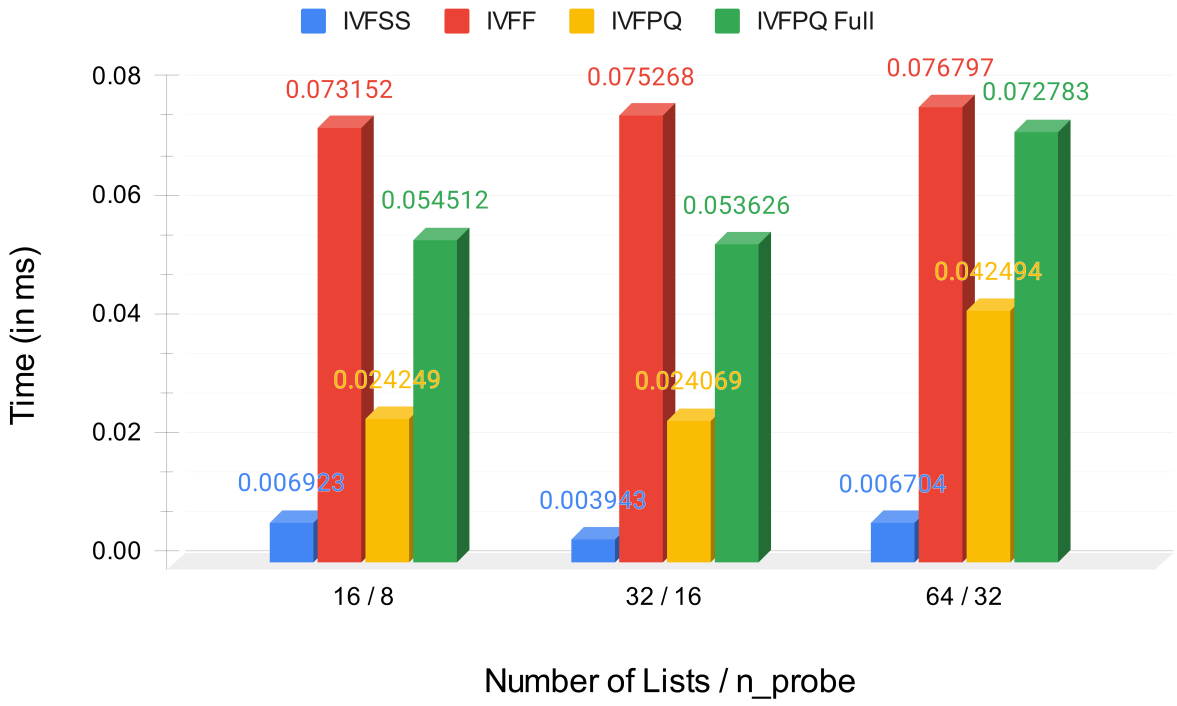

Figure 6.7: Time it takes for each algorithm to perform on Dataset 2

Surprisingly, when given the small dataset, IVFSS performs considerably better compared to the two other algorithms. Regardless of how the number of lists and n_probe change, it is tens of times faster than IVF Flat, IVFPQ, and IVFPQ Full. Owing to its structural properties, IVF Flat exhibits nearly constant query-time behavior across datasets, as observed consistently here and in the two preceding datasets, despite differences in absolute execution times.

Metric 2. Memory:

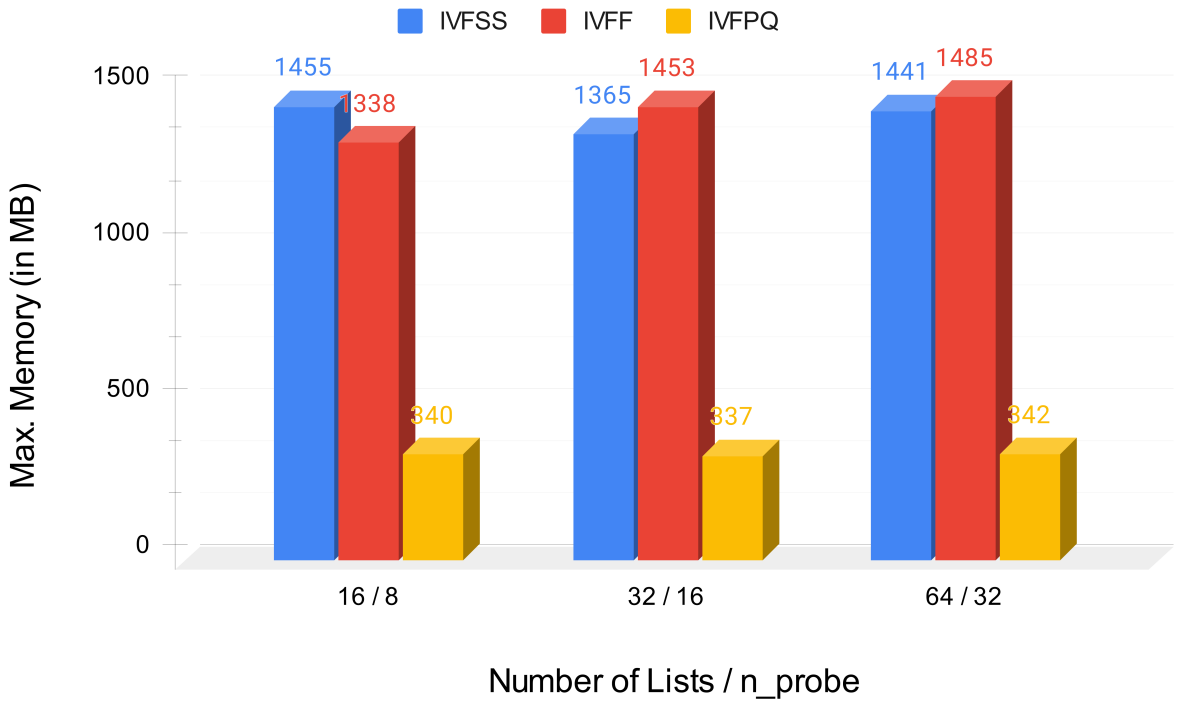

Figure 6.8: Time it takes for each algorithm to perform on Dataset 2

The maximum memory consumption pattern did not change significantly for the IVFSS, as it is still roughly the same as IVF Flat, and IVFPQ is still at a constant memory utilization rate.

Metric 3. First-Hit Rank:

Number of Lists / n_probe

Algorithms	16 / 8	32 / 16	64 / 32
IVFSS	1	1	1
IVFF	1	1	1
IVFPQ	1	1	1
IVFPQ Full	1	1	1

Figure 6.9: First-Hit Rank of each algorithm on Dataset 3

All algorithms performed well on this small audio dataset, consistently returning the

correct index of the nearest vector. Notably, IVFSS achieved this performance while also demonstrating substantially faster retrieval times compared to the other algorithms.

Chapter 7

CONCLUSION

7.1 Contributions

This thesis showed that IVFSS can be a good competitor for both IVF Flat and IVFPQ. It drastically outperforms IVF Flat in most of the benchmarks, and slightly loses to IVFPQ in some of them. The results also demonstrate that the MASS library can function not only as the core foundation upon which an application is typically built, but also as an independently integrated third-party component. Throughout this work, the thesis and the MASS CUDA library informed and reinforced one another: MASS CUDA streamlined the implementation process, while the IVFSS exposed several limitations and areas for improvement within MASS CUDA.

7.2 Limitations

One of the biggest limitations was the time constraint given to design and implement this algorithm. As it turns out, it is not that easy to come up with a new algorithmic solution. Another limitation would be MASS CUDA's inability to scale up to more than one GPU device at a time.

7.3 Future Work

First of all, IVFSS needs to be implemented without MASS CUDA, by handling all the low-level work by itself. This will improve the speed and significantly reduce the current memory consumption, which mainly occurs due to the memory consumed by the MASS places. It will also enable IVFSS to work on both GPUs connected through NVLink. Then, the index storage structure can be modified to optimize and remove unnecessary components. Lastly, more benchmarks will be needed, potentially using several clusters/servers, simulating hundreds of queries simultaneously.

BIBLIOGRAPHY

[1] Sivic and Zisserman. “Video Google: a text retrieval approach to object matching in videos”. In: *Proceedings Ninth IEEE International Conference on Computer Vision*. 2003, 1470–1477 vol.2. DOI: 10.1109/ICCV.2003.1238663.

[2] Herve Jégou, Matthijs Douze, and Cordelia Schmid. “Product Quantization for Nearest Neighbor Search”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57.

[3] Jeff Johnson, Matthijs Douze, and Hervé Jégou. *Billion-scale similarity search with GPUs*. 2017. arXiv: 1702.08734 [cs.CV]. URL: https://arxiv.org/abs/1702.08734.

[4] Lisa Kosiachenko, Nathaniel Hart, and Munehiro 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*. Lecture notes in computer science. Cham: Springer International Publishing, 2019, pp. 139–152.

[5] John Emau, Timothy Chuang, and Munehiro 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*. 2011, pp. 369–375. DOI: 10.1109/PACRIM.2011.6032921.

[6] *MASS*. URL: https://depts.washington.edu/dslab/MASS/.

[7] Peter N. Yianilos. “Data structures and algorithms for nearest neighbor search in general metric spaces”. In: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’93. Austin, Texas, USA: Society for Industrial and Applied Mathematics, 1993, pp. 311–321. ISBN: 0898713137.

[8] Alexander Ponomarenko, Nikita Avrelin, Bilegsaikhan Naidan, and Leonid Boytsov. “Comparative Analysis of Data Structures for Approximate Nearest Neighbor Search”. In: Jan. 2014.

[9] M H Hadid, Qasim Mohammed Hussein, Z T Al-Qaysi, M A Ahmed, and Mahmood M Salih. “An overview of content-Based Image Retrieval methods and techniques”. en. In: *ijcsm* (July 2023), pp. 66–78.

[10] Erik Bernhardsson. *ANNOY*. URL: https://github.com/spotify/annoy.

[11] Piotr Indyk and Rajeev Motwani. “Approximate nearest neighbors: towards removing the curse of dimensionality”. In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. STOC ’98. Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 604–613. ISBN: 0897919629. DOI: 10.1145/276698.276876. URL: https://doi.org/10.1145/276698.276876.

[12] Yu A Malkov and D A Yashunin. “Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs”. In: (2016). eprint: 1603.09320 (cs.DS).

[13] Xin Jin and Jiawei Han. “K-Means Clustering”. In: *Encyclopedia of Machine Learning*. Boston, MA: Springer US, 2011, pp. 563–564.

[14] T. Kitagawa and Y. Kiyoki. “A mathematical model of meaning and its application to multidatabase systems”. In: *Proceedings RIDE-IMS ‘93: Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*. 1993, pp. 130–135. DOI: 10.1109/RIDE.1993.281933.

[15] Yasushi Kiyoki, Takashi Kitagawa, and Takanari Hayama. “A metadatabase system for semantic image search by a mathematical model of meaning”. In: 23.4 (1994). ISSN: 0163-5808. DOI: 10.1145/190627.190639. URL: https://doi.org/10.1145/190627.190639.

[16] Alex Li and Munehiro Fukuda. “Agent-Based Parallelization of a Multi-Dimensional Semantic Database Model”. In: *2023 IEEE 24th International Conference on Information Reuse and Integration for Data Science (IRI)*. 2023, pp. 64–69. DOI: 10.1109/IRI58017.2023.00019.

[17] *cuVS*. URL: https://github.com/rapidsai/cuvs.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[19] Aurora Linh Cramer, Ho-Hsiang Wu, Justin Salamon, and Juan Pablo Bello. “Look, Listen, and Learn More: Design Choices for Deep Audio Embeddings”. In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 3852–3856. DOI: 10.1109/ICASSP.2019.8682475.

[20] Relja Arandjelović and Andrew Zisserman. *Look, Listen and Learn*. 2017. arXiv: 1705.08168 [cs.CV]. URL: https://arxiv.org/abs/1705.08168.

Appendix A

CODE AND DATASETS

The datasets that were used in this thesis can be found through these link:

Dataset 1. imageneetood: *Link*

Dataset 2. tiny-imagenet-200: *Link*

Dataset 3. GTZAN: *Link*

The code of the IVFSS implementation for the GPU using MASS CUDA can be found through this link: *Link*

The modified code of MASS CUDA, adjusted for IVFSS, can be found through this link: *Link*