OTIMIZANDO TEMPO DE RESPOSTA EM BUSCAS POR SIMILARIDADE EM LARGA ESCALA

RAFAEL MARTINS DE SOUZA

OTIMIZANDO TEMPO DE RESPOSTA EM BUSCAS POR SIMILARIDADE EM LARGA ESCALA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Renato Antônio Celso Ferreira
Coorientador: George Luiz Medeiros Teodoro

Belo Horizonte
Abril de 2020

RAFAEL MARTINS DE SOUZA

OPTIMIZING RESPONSE TIME IN LARGE SCALE SIMILARITY SEARCHES

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Renato Antônio Celso Ferreira
Co-Advisor: George Luiz Medeiros Teodoro

Belo Horizonte
April 2020

Souza, Rafael Martins de.

S729o Otimizando tempo de resposta em buscas por similaridade em larga escala [manuscrito] / Rafael Martins de Souza. – 2020.
xix, 70 f. il.

Orientador:. Renato Antônio Celso Ferreira.
Coorientador: George Luiz Medeiros Teodoro
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.65-70
.
1. Computação – Teses. 2. Computação distribuída – Teses. 3. Busca por similaridade – Teses. I. Ferreira, Renato Antônio Celso. II. Teodoro, George Luiz Medeiros. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV.Título.

CDU 519.6*34(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Otimizando tempo de resposta em buscas por similaridade em larga escala

RAFAEL MARTINS DE SOUZA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. RENATO ANTÔNIO CELSO FERREIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. GEORGE LUIZ MEDEIROS TEODORO - Coorientador
Departamento de Ciência da Computação - UFMG

PROF. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação - UFMG

PROF. WILLIAM ROBSON SCHWARTZ
Departamento de Ciência da Computação - UFMG

PROF. EDUARDO ALVES DO VALLE JUNIOR
Departamento de Engenharia da Computação e Automação Industrial - UNICAMP

Belo Horizonte, 13 de Março de 2020.

Resumo

Busca por similaridade é uma operação fundamental encontrada em serviços de multimídia online. Estes serviços precisam lidar com bases de dados muito grandes, enquanto, ao mesmo tempo, eles tem que minimizar o tempo de resposta observado por seus usuários. Isto é especialmente complexo porque esses serviços lidam com taxa de consultas variáveis. Consequentemente, eles precisam se adaptar durante o tempo de execução para minimizar o tempo de resposta a medida que a taxa de consultas varia. Nesta dissertação, nós abordamos os desafios mencionados anteriormente com a paralelização, em memória distribuída, da busca por vizinhos mais próximos com quantização em produto, também conhecida como IVFADC, para máquinas híbridas com CPU e GPU. Nosso IVFADC paralelo também implementa um mecanismo de execução out-of-core, para permitir que a GPU processe bases de dados que não cabem na memória, o que é crucial para realizar buscas em base de dados muito grandes. O uso de CPU com GPU com roubo de trabalho gerou uma redução média no tempo de resposta de 1.6× quando comparado a usar a GPU somente. Inclusive, a nossa abordagem para adaptar o sistema para taxa de consultas variáveis, chamada de Dynamic Query Processing Policy (DQPP), alcançou uma redução média no tempo de resposta de 7× vs a política gulosa. Finalmente, em todas as configurações, o sistema se mostrou capaz de alcançar altas taxas de processamento de consultas e escalabilidade quase linear. Nós executamos o sistema em um ambiente com até 256 NVIDIA V100 GPUs, e uma base de dados de 256 bilhões de vetores de características SIFT.

Palavras-chave: PQANNS, Computação Distribuída, kNN, Busca por similaridade.

Abstract

Similarity search is a core operation found in several online multimedia services. These services have to handle very large databases, while, at the same time, they must minimize the query response times observed by users. This is especially complex because those services deal with fluctuating query workloads (rates). Consequently, they must adapt at run-time to minimize the response times as the load varies. In this dissertation, we address the aforementioned challenges with a distributed memory parallelization of the product quantization nearest neighbor search, also known as IVFADC, for hybrid CPU-GPU machines. Our parallel IVFADC also implements an out-of-core scheme to use the GPU for databases in which the index does not fit in its memory, which is crucial for searching in very large databases. The careful use of CPU and GPU with work-stealing led to an average reduction of the response time of 1.6× as compared to using the GPU only. Also, our approach to adapt the system to fluctuating loads, called Dynamic Query Processing Policy (DQPP), attained an average response time reduction of 7× vs. the greedy policy. Finally, in all settings, the system has been shown to attain high query processing rates and near-linear scalability. We have executed our system in an environment with up to 256 NVIDIA V100 GPUs and a database of 256 billion SIFT features vectors.

Palavras-chave: PQANNS, Distributed Computation, kNN, Similarity Search.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Similarity search is a core operation found in several online multimedia services. These services have to handle very large databases while, at the same time, they must minimize the query response times observed by users. This is especially complex because those services deal with fluctuating query workloads (rates). Consequently, they must adapt at run-time to minimize the response times as the query load varies. This is a challenging problem and we hope to address it in this dissertation. In this chapter we expand on the presented problem: reducing response time on fluctuating query workloads scenarios, list our objectives and contributions, and finish with a description of how this dissertation is structured.

1.1 Motivation

The amount of multimedia data currently available is immense and is growing at an unprecedented rate. For instance, more than three hundred million photos are uploaded on Facebook every day, more than five hundred hours of video are uploaded every minute on YouTube, and more than five thousand tweets are sent every second. With this ever-increasing amount of data available, it becomes increasingly challenging to design efficient search tools.

One challenging aspect is how to describe data such as music, video, and images automatically, since manually describing them would be unfeasible due to the huge quantity of data. This problem can be addressed by representing multimedia objects using algorithmically computed high-dimensional feature vectors or descriptors (100-1000+ dimensions), as seen in Oliva and Torralba (2001); Sivic and Zisserman (2003); Lowe (2004); Douze et al. (2009); Jégou et al. (2010); Babenko et al. (2014); Wan et al. (2014); Gong et al. (2014).

Searching in such databases is a fundamental operation for several multimedia applications, such as the ones listed in Böhm et al. (2001) and Datta et al. (2008).

In this context, a similarity search corresponds to, given a query vector, find the K closest vectors present in the database. This operation is well known as k-NN (k nearest neighbors). While a search may include other steps, the k-NN step is usually the most time consuming one. Doing an exact k-NN search is costly due to, not only the large databases currently used by web applications but also due to the high dimensionality of the vectors. An alternative to the exhaustive search would be to employ specialized data structures, such as kd-trees (Friedman et al., 1977) and ball trees (Uhlmann, 1991), to partition the search space and efficiently prune data partitions during the search. However, the pruning efficiency and, consequently, the performance of these techniques, degrades as the data dimensionality grows, due to the well-known curse of dimensionality, described in Weber et al. (1998).

The approximate nearest neighbors (ANN) search has been proposed as a solution for those applications in which the exact search is not strictly necessary, allowing for accuracy to be traded off for speed. This motivated the development of several ANN algorithms, such as the ones proposed by Indyk and Motwani (1998a); Nister and Stewenius (2006); Silpa-Anan and Hartley (2008); Muja and Lowe (2009); Jegou et al. (2011); Malkov and Yashunin (2018). The product quantization nearest neighbor search (also known as IVFADC), proposed by Jegou et al. (2011), has received special attention compared to other ANN algorithms due to its ability to minimize memory requirements while improving speed. This is attained by representing descriptors with small quantization codes and by using an inverted list to avoid exhaustive search in the quantized space.

Most of the previous work on ANN has focused on optimizing ANN algorithms for processing on a single machine. This decision does not match with the demands of modern online content-based multimedia retrieval services, which must handle increasingly large databases that would not fit in the memory of a single node. Also, while online applications are concerned with minimizing the query response times of individual queries, the aforementioned algorithms were developed to maximize throughput in a batch scenario, where several queries are bundled and processed together as a single task. The online setting presents additional challenges because these systems experience fluctuating query arrival rates, and must adapt at run-time to minimize the observed response times.

1.2 Objectives

The main objective of this work is to introduce a scalable and efficient similarity search system that works well in more realistic scenarios typically encountered by large content-based multimedia retrieval services. To this end, we adapt the IVFADC implementation from the Faiss library to work in a distributed scenario and develop query processing strategies to reduce the average response time in scenarios where the query rate fluctuates over time. We focus on the response time, rather than throughput, as is commonly done, because it is a more important metric to the end-user of such systems.

Furthermore, we also tackle the problem of how to combine the use of CPU and GPU in order to improve the average response time. This is especially important in out-of-core scenarios, where the database doesn't fit in the GPU memory since it is challenging to develop a solution that produces good response times at both low and high query rates.

1.3 Main Contributions

This dissertation makes the following major contributions:

- We implement an efficient distributed memory version of the IVFADC algorithm for hybrid CPU-GPU systems. The execution on a CPU-GPU machine can answer up to 7k queries/sec on a single GPU on a dataset with 500 million SIFT descriptors. Also, the distributed memory execution attained almost linear scalability in execution with 256 V100 NVIDIA GPUs in scenarios with in-/out-of-core in which a database with up to 256 billion SIFT descriptors was used.

- We have developed the DQPP strategy for adjusting the system at run-time under fluctuating workloads, which decides the number of queries for concurrent execution with the GPU according to the system load. DQPP improved the response time vs. the greedy approach on average by 7×.

- We implemented an out-of-core GPU execution scheme that uses work-stealing for maximizing performance on the CPU-GPU cooperation. The CPU-GPU execution with work-stealing improved the throughput and response times vs. the GPU-only execution, respectively, by up to 1.27× and 3.1×.

1.4 Dissertation Roadmap

The remainder of this dissertation is organized as follows: Chapter 2 presents a literature review on the associated background topics. Chapter 3 presents our IVFADC parallelization on distributed memory. Chapters 4 and 5 presents our response time aware cooperative in-core and out-of-core, respectively, query processing strategies. Finally, the experimental results are discussed in Chapter 6, and our conclusions in Chapter 7.

Chapter 2

Literature Review

This section describes the background to our work. As mentioned, our main goal consists of constructing a scalable searching system for multimedia data. One example of such a system is Google's "Search by Image", in which you give it an image, and it returns several images that are similar (maybe even equal) to the given one. Figure 2.1 shows Google's Image Search in action. One important aspect to notice is that the search works even when the images are rotated or scaled.

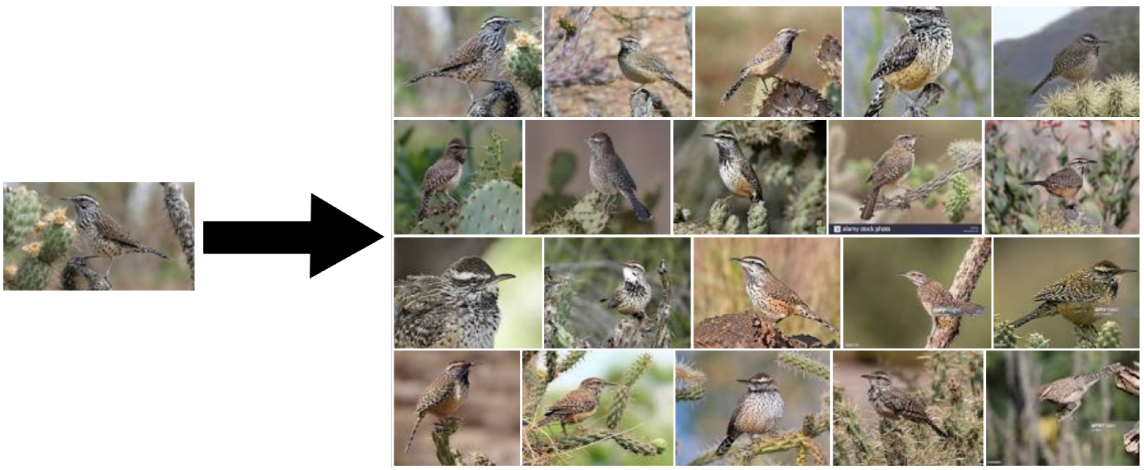

Figure 2.1. Example of an image search from Google's Image Search. We have the query image on the left, and the query results on the right.

The search process involves two phases, and several different algorithms, which we will present in this section. First, the given query (in Figure 2.1, an image) is encoded in some intermediate representation that intends to bridge the "semantic gap" between its low-level coding and its semantic concepts. For instance, when dealing with images, this intermediate representation might abstract things like rotation, scale, etc. Usually, this

intermediate representation takes the form of one or more multidimensional vectors, also known as descriptors. While similarity search can be applied to several types of media, such as songs, videos, images, etc, we will focus on images in this work. In section 2.1, we list some popular image descriptors.

In the second phase of the search, those descriptors are compared against the descriptors stored in the database, using some distance metric (e.g., Euclidean distance) to represent the similarity between them. This problem is usually generalized as the k nearest neighbors (kNN) problem: given a query vector, find the k vectors that are closest (most similar) to the query. Doing an exhaustive search is almost always too expensive, so several strategies to better explore the search space were created. Some of them are detailed in Section 2.2. When dealing with large datasets and high dimensional vectors, computing the exact kNN might be too expensive, so several approximate approaches were developed, and a few of them are introduced in Section 2.3. Among those approximate approaches, the product quantization approach has been very successful in reducing the memory requirements of large datasets. The original product quantization method and its subsequent improvements are discussed in more detail in its own section, Section 2.4, since they form the basis of this work.

Usually, the similarity search is performed in a distributed setting, since a single computing node is often not enough for the demands of modern CBMR systems that require large datasets to be processed. Section 2.6 shows some of the previous work in this area.

Last, we compare our work with related works in Section 2.7, and we summarize this chapter in Section 2.8.

2.1 Image Descriptors

An image descriptor, as the name suggests, describes the contents of the image. Usually, similar images have similar descriptors, given some similarity metric. Several image descriptors have been evaluated in the literature. They capture different properties of the image, such as color, texture, shape, and orientation; and are encoded into (usually) high-dimensional feature vectors. Some of the most popular descriptors are: SIFT (Lowe, 1999), Gist (Oliva and Torralba, 2001), Bag of Visual Words (Sivic and Zisserman, 2003), VLAD (Jégou et al., 2010), and Deep Features (Babenko et al., 2014).

2.1.1 SIFT

The scale-invariant feature transform descriptor (Lowe, 1999), also known as SIFT, is a local descriptor that describes a feature point by a "histogram" of image gradient orientation and location (Jain, 2014). It is largely invariant to changes in scale, illumination, and local affine distortions (Lowe, 1999). It is worth noticing that it is a local image descriptor, meaning that, from each image, a set of descriptors are extracted, each corresponding to a keypoint of the image. Those keypoints are automatically obtained during the SIFT feature extraction process. The extraction process works in four steps: first, by detecting local extrema in the scale-space (in order to be scale invariant), potential keypoints are selected. However, not all of them are useful. In a subsequent step, keypoints that are deemed to have too low contrast or that lie along an edge are removed. Next, each keypoint receives one or more orientation assignments, in order to make them rotation invariant. Last, the keypoint descriptor itself is computed, by measuring the local image gradients in a window around each keypoint. They are processed in a way that tries to make the resulting descriptor invariant to changes in illumination and local shape distortion.

2.1.2 Gist

As the name implies, this global descriptor tries to obtain a "gist" of the image, or, in other words, a (relatively) low dimensional scene representation acquired over a short observation time frame (Oliva and Torralba, 2001). One of the objectives of this method is to create a biologically plausible framework. First, the input image is filtered in a number of low-level visual "feature" channels. From the orientation channel, employing Gabor filters at four different angles and four spatial scales, 16 feature maps are obtained. From the color channel, twelve feature maps are obtained, and from the intensity channel, six more, giving a total of 34 feature maps. More or less channels could be used if deemed appropriate. After that, each map is divided in a 4x4 grid, and, for each cell, an average is computed. In this way, it is obtained a global 544 (34x4x4) dimensional feature descriptor. The resulting vector might have its dimensions reduced by using techniques such as PCA, ICA, etc.

2.1.3 Bag of Visual Words

Introduced by the seminal work of Sivic and Zisserman (2003), it is a method inspired by text retrieval techniques. The idea is pretty simple: to represent an image as a histogram of the frequency of "visual words", in the same way that a text might be

represented as a histogram of the frequency of words that occur within itself. The first step is to create the visual vocabulary (codebook). In general, from local descriptors, they are created by clustering them in K clusters, each corresponding to a visual word. To encode the local descriptors, one simple approach is to assign each descriptor to the nearest visual word. The last step of the process is the creation of the histogram that will represent the image. It may be done either by sum-pooling or max-pooling. This approach has been improved in several other works, such as Avila et al. (2011) and Avila et al. (2013).

2.1.4 VLAD

VLAD, or vector of locally aggregated descriptors, was introduced by Jégou et al. (2010). First, a codebook of k words is learned. Then, for each visual word in the codebook, we compute the sum of the residuals of each local descriptor that is mapped to this visual word and concatenate them all to form a single descriptor. Formally, assume that the local descriptors have d dimensions and that the codebook has k visual words. The resulting descriptor, v, would have k x d dimensions. Let c_i be the i-th visual word in the codebook, and C(i) the set of local descriptors that map to the c_i visual word. The component $v_{i,j}$ of the global descriptor would be obtained as follows:

$$v_{i,j} = \sum_{x \in C(i)} x_j - c_{i,j} \tag{2.1}$$

v is then L2-normalized.

One of the great advantages of the VLAD descriptor is that it is simple, efficient, and produces good enough results. It has been used for video retrieval in Revaud et al. (2013).

2.1.5 Deep Features

Deep convolutional neural networks (LeCun et al., 1990) have been very popular for image classification tasks (Krizhevsky et al., 2012). In the work of (Babenko et al., 2014), they explored the idea of adapting deep convolutional neural networks to the task of image retrieval. The intuition behind their approach is that in the later layers of the network, their output corresponds to high-level features of the image, and, therefore, are suitable to describe the image. By extracting the results of upper layers of the deep convolutional neural network, and applying dimensionality reduction techniques such as PCA, they were able to produce (relatively) low dimensional image descriptors that achieved high accuracy in several image retrieval experiments.

2.2 Exact Similarity Search Algorithms

Given a descriptor query, the similarity search refers to finding the k nearest neighbors (k-NN) descriptors in the database. The exact brute force algorithm for this operation computes the distance from the query descriptor to each descriptor in the database and selects the k closest ones. This is, however, prohibitive for online multimedia services.

In order to reduce the cost of the exact k-NN, several data structures, such as kd-trees (Friedman et al., 1977) and ball trees (Uhlmann, 1991), were used to divide the database of descriptors into partitions according to their spatial location. This organization is intended to prune partitions during the search phase and, consequently, speed it up. However, because of the well known "curse of dimensionality" (Böhm et al., 2001; Weber et al., 1998), the sparsity of the data quickly increases as the data dimensionality grows. This reduces the pruning efficiency of those data structures and, as a consequence, their performance improvements vs. the brute force search.

2.2.1 KD-Tree

The kd-tree, as proposed by (Bentley, 1975), can be seen as a generalization of the binary search tree, for data with k keys. At each level of the tree, one of its dimensions is chosen as the discriminator, which will be the key used to decide on whether a new node will be added to the left or right subtree, in a similar manner to a binary search tree: values smaller than the discriminator of the parent node are added to the left subtree, and values higher are added to the right subtree. When searching, the node keys are compared against the corresponding discriminator for each level. A terminal node might correspond to a single element of the dataset or to a list of elements. Furthermore, the nodes themselves don't need to correspond to entries of the dataset. There are many ways to decide which key will be the discriminator of a specific node, and which value to use. Some are proposed in Bentley (1975) and Friedman et al. (1977). Figure 2.2 shows an example of a 3-dimensional kd-tree.

The work of Friedman et al. (1977) shows one way in which to use kd-trees to speed up kNN searches. At every node, first, we search in the subregion (subtree) corresponding to the query. If necessary, we search in the opposite subregion. To verify whether it is necessary, a ball is created, centered in the query, with a radius equal to the distance of the query to the kth closest element found so far, and seeing if this ball intersects the corresponding opposite subregion. If it doesn't, we know that we don't need to look into that subregion. By doing this, we can reduce the search space and improve search performance. However, it suffers in high dimensional spaces,

due to the curse of dimensionality. The work of (Beis and Lowe, 1997) extends the kd-tree based search to be more practical at higher dimensions, but it has the tradeoff of being only an approximate algorithm.

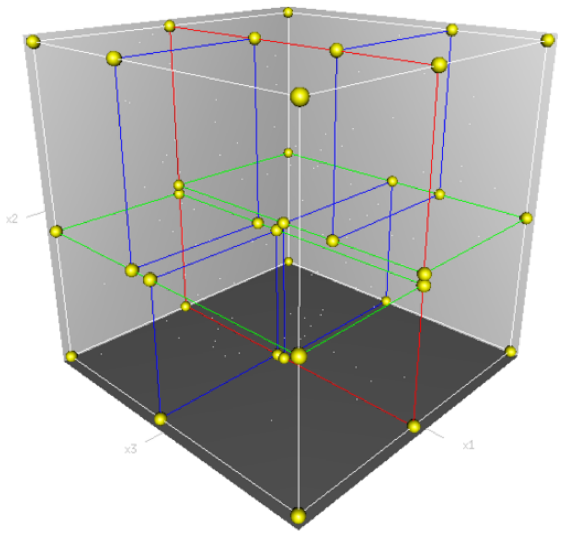

Figure 2.2. A 3-dimensional kd-tree.

2.2.2 Ball Trees

The ball tree (Uhlmann, 1991) can be seen as a generalization of the binary search tree, where every node defines a ball, of minimum radius, that contains all the elements associated with it. The radius of the ball is stored in each node. While the balls of the left and right children of a node might intersect, each point is associated with just one of them, the one closest. Again, the nodes don't necessarily correspond to entries in the dataset.

There are several ways to construct a ball tree (Omohundro, 1989). One of the simplest ones is called the k-d construction algorithm. In a top-down manner, the points are divided according to a simple less-or-greater check on the median value of the dimension that achieves the greatest spread of points. The most central point, according to the dimension chosen, will be the point that represents the node. Figure 2.3 illustrates this process.

When searching the points associated with a node, we first search the child closest to the query, and then the other one. We keep the k smallest distances found in a priority queue. To prune the search, we compute a lower bound on the distance to the

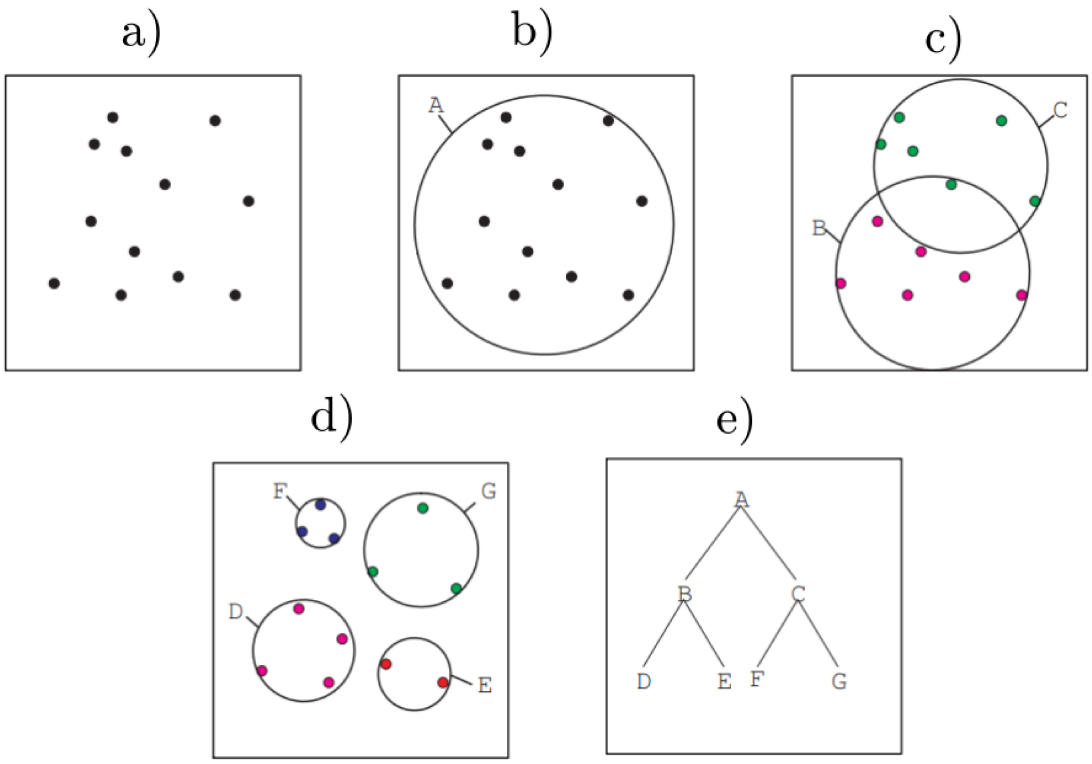

Figure 2.3. An example of the construction of a ball-tree. Steps "b" through "d" show the clustering steps. "e" shows the resulting tree. Image was taken and adapted from Liu et al. (2006)

query. This lower bound (L) can be obtained as follows:

$$L = max(L_p, d(Q, N) - R) \tag{2.2}$$

where L_p is the lower bound associated with its parent node, d is the distance function, Q is the query, N is the current node, and R is the radius of the ball associated with this node.

That bound can easily be proven. The first term of the max expression, L_p, is the bound of the parent node. Since the points associated with a node are a subset of the points associated with its parent node, it follows that L must be at least L_p. The other term of the max can be proven as follows: by the triangle inequality, $d(Q, X) + d(X, N) \geq d(Q, N)$, where X is an arbitrary point that is within the ball associated with node N. Since X is in the ball associated with N, it follows that $d(X, N) \leq R$, and, therefore $d(Q, X) + R \geq d(Q, N)$. By placing R on the other side of the inequality, we obtain that $d(Q, X) \geq d(Q, N) - R$. In other words, L must be at least $d(Q, N) - R$.

If L is bigger than the k-th smallest distance we found so far, we can prune the subtree out of the search, which saves time. However, again, this method suffers at

high dimensions, due to the curse of dimensionality.

This method has been extended in the work of Dolatshah et al. (2015) with a modified space partitioning scheme that considers the distribution of the data points in order to produce better trees.

2.3 Approximate Similarity Search Algorithms

Due to the high cost of computing the exact k-NN, the approximate nearest neighbors (ANN) search has been proposed for applications in which the exact solution is not essential. The approximation, in this case, may be bound by the number of missing correct descriptors or their distance to the query (Indyk and Motwani, 1998b). Even on the ANN case, efficiently searching large databases remains a very challenging problem.

Several ANN methods have been proposed, such as: LSH (Indyk and Motwani, 1998a), Randomized KD-Trees (Silpa-Anan and Hartley, 2008), Hierarchical K-means Tree (Nister and Stewenius, 2006), FLANN (Muja and Lowe, 2009), Hierarchical Navigable Small World Graphs (Malkov and Yashunin, 2018), and Product Quantization (Jegou et al., 2011).

2.3.1 LSH

Locality Sensitive Hashing (LSH) is a technique proposed by Indyk and Motwani (1998a), which uses locality sensitive hashing functions to index data into multiple hash tables. Descriptors in a hash's entry, or bucket, are expected to be close in the multi-dimensional space. The search visits buckets to which the query is hashed, the distances to descriptors on those buckets are computed, and the nearest neighbors from that subset will compose the query answer. The tuning of LSH parameters leads to several hash tables, which poses a scalability challenge with large databases due to high memory demands. Also, modern machines have a low random memory access performance (Teodoro et al., 2014a), which is a major pattern in LSH during hash table buckets visits.

2.3.2 Randomized KD-Trees

In Silpa-Anan and Hartley (2008), an improvement to the kd-tree based search was proposed, trading exactness for performance. The idea is to search in multiple randomized kd-trees, each constructed using a different rotation of the dataset. The trees share a priority queue containing unexplored branches, ordered by distance to the query.

In this approach, the dimensions in which the data is split at each level is decided randomly, among the D dimensions that offer the greatest variance. Muja and Lowe (2009) argues that using D = 5 performs well for a variety of datasets. The precision of the search can be controlled by choosing the number of terminal nodes that will be visited.

2.3.3 Hierarchical K-means Tree

In this approach, the data at each level is divided into K distinct regions by using k-means clustering. Then, recursively, each region is subdivided into regions, until it has less than K elements. This results in a tree that can be searched effectively. When searching, the branches whose center are closer to the query are visited, and then, once the transversal has ended, it is restarted from a new branch. In Muja and Lowe (2009), they propose the best bin approach, where unexplored branches are added to a priority queue, ordered by distance to the query. When the search restarts, the branch in the priority queue closest to the query is used as a starting point. Again, the search precision can be controlled by changing the number of terminal nodes visited.

2.3.4 FLANN

The fast library for approximate nearest neighbors (FLANN) is a popular open-source project for ANN search. It implements multiple approximate kNN algorithms and selects the most appropriate along with its parameters for a given dataset. This alleviates the hard and error-prone task of choosing from the several indexing approaches available. The user can specify, for instance, the desired precision, the importance of minimizing memory, build time, etc. It also shows that there is no one-size-fits-all solution for ANN search (Muja and Lowe, 2009). Because of its efficiency, FLANN is typically used as a baseline for performance comparisons.

2.3.5 Hierarchical Navigable Small World

The Hierarchical Navigable Small World (HSNW), introduced by Malkov and Yashunin (2018), is a graph-based approach that can be thought of as a multi-layer and multi-resolution variant of a proximity graph (Li et al., 2019). In it, points are distributed across several layers, starting from 0, which contains every point. The points are added in an arbitrary order to the structure, and, for each element, a maximum layer m is determined randomly through an exponentially decaying probability distribution. The point is added to every layer that is numbered less than or equal to m.

The insertion process can be divided in two phases: in the first, from the top layer to the $m+1$ layer, a variation of the greedy transversal is done to find the closest element to the query in that layer. Then, the search is restarted in the next layer, starting from the point found in the previous layer. In the second phase, from the layer m down to layer 0, a similar procedure is done, however instead of finding only the closest point, the ef closest are found. The point is inserted in that layer, and then, from the ef points chosen, M are chosen to be linked with the inserted element at that layer, using some heuristic (eg. the M closest ones). If at some point, a point has more than M_{max} links, some links are dropped, according to the same heuristic. The construction process continues in the next layer, starting from the ef points found in the last layer. By changing the ef parameter, the search quality of the obtained structure can be controlled.

The searching process is very similar to the insertion process of a point that was inserted with a maximum layer equal to 0. However, the closest neighbors found at layer 0 are, instead, returned as the query result.

Figure 2.4 illustrates both the searching process and the layered structure. As we go from the top to the bottom layer, the average distance between the points decreases (as the number of points increase). So, in a sense, at each layer we go through, we increase the zoom level at which we are searching.

While the HSNW produces very good performance results in practice, it consumes more memory than the traditional product quantization methods, and also is harder to do in a distributed setting.

2.4 Product Quantization for Approximate Nearest Neighbor Search

In this section we will describe the product quantization based approach for approximate k nearest neighbor search, as introduced by Jegou et al. (2011). Since we use product quantization directly in our work, we will describe it in greater detail.

2.4.1 Vector Quantization

Quantization is the process in which a continuous (or almost continuous) set of values are mapped to a discrete set. Quantization is used in nearest neighbor search with the intent of reducing the cardinality of the search space, decreasing both time and space costs of the search. The tradeoff is that the search accuracy suffers. Formally, a vector

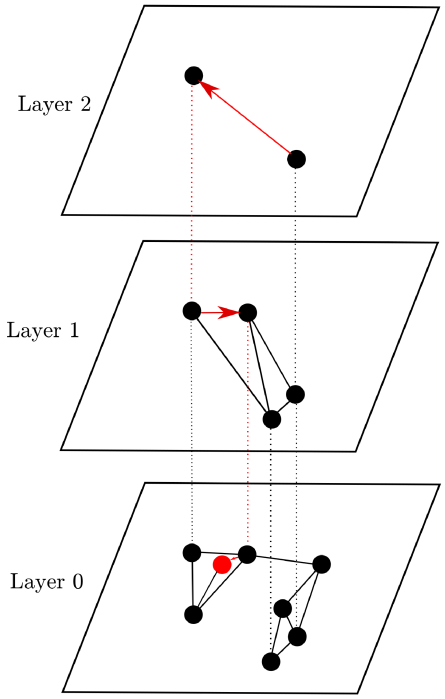

Figure 2.4. This image illustrates both the layered structure and the searching process. The red dot corresponds to the query. Each red arrow correspond to a step of the greedy transversal. Starting from the top layer, we try to go to nodes closer to the query.

quantizer (q) is a function mapping a d-dimensional vector to a finite set of vectors. This set of vectors (C) is called the codebook, and each individual vector is a centroid (c_i).

Associated with each c_i, there is a set V_i, usually referred to as a Voronoi cell, which corresponds to the set of all vectors that map to the same c_i centroid. Formally:

$$V_i = \{x \in R^d : q(x) = c_i\} \tag{2.3}$$

Figure 2.5 shows the respective Voronoi cells obtained from a certain arbitrary set of centroids. Vectors within the same cell are reconstructed as the same centroid.

For a quantizer to be optimal, with respect to the mean squared error between the vector x and its quantization, q(x), it has to obey the following two properties (known as Lloyd optimality conditions):

- a vector must be quantized to its nearest centroid

$$q(x) = c_i : \forall c_j \in C, d(x, c_i) \leq d(x, c_j) \tag{2.4}$$

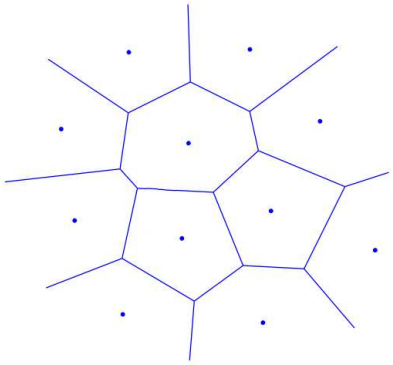

Figure 2.5. This image illustrates the Voronoi cells associated with each centroid, represented as points in the image. Vectors in the same cell are reconstructed as the same centroid. The image was adapted from Jégou et al. (2011).

- the centroid of the Voronoi cell corresponds to the expected value of the vectors within the same Voronoi cell:

$$c_i = \int_{V_i} p(x)dx \tag{2.5}$$

There are many ways to compute the centroids. One that works well is the Lloyd quantizer, which corresponds to the kmeans clustering algorithm. It produces only a local optimum in terms of quantization error, but works well enough in practice.

2.4.2 Product Quantization

One limitation of the naive quantization approach is that it requires a large codebook (Jegou et al., 2011) to achieve reasonable quantization errors. For instance, if we quantized 128 dimensional SIFT vectors, with each dimension corresponding to a 32-bit float, into 64-bit codes (ie. a reduction of 64 times in the space requirements), we would require 2^{64} centroids, which is not viable.

To address this, Jégou et al. (2011) proposed the product quantization approach for nearest neighbor search. The idea is that, instead of quantizing the entire vector at once, each part of the vector is quantized independently by a subquantizer, and the resulting centroid is the concatenation of the centroids from each subquantizer. More formally, the d-dimensional input vector x is split into m subdimensions. Associated with each subdimension is a subquantizer (q_i).

Let u_i be the projection function such that:

$$u_i(x) = x_{(i-1)\frac{d}{m}+1}, ..., x_{i\frac{d}{m}} \tag{2.6}$$

which, in essence, picks the ith slice of $\frac{d}{m}$ components of the vector x.

Then, the product quantizer q_p, can be defined as follows:

$$q_p(x) = q_1(u_1(x)), ..., q_m(u_m(x)) \tag{2.7}$$

The resulting codebook will correspond to the cartesian product of the individual codebooks:

$$C = C_1 \times ... \times C_m \tag{2.8}$$

where C_i is the codebook associated with the subquantizer q_i.

Assuming that each subquantizer has the same number of reproduction values associated with it (k^*), the total number of centroids would be:

$$k = (k^*)^m \tag{2.9}$$

When m = 1, we obtain the naive vector quantization, as described previously.

Instead of storing the codebook explicitly, the codebooks associated with each subquantizer are stored, reducing space requirements. Vectors in the dataset are stored as a tuple of the indexes of the respective centroids (one for each subquantizer).

2.4.3 Distance Computation

There are two basic methods to compute the distance between a query vector x and the database vector y, of which we have access only to its quantized version $q_p(y)$:

- **Symmetric distance computation (SDC):** in this case, the query vector is also quantized, and the distance is approximated by:

$$d(x, y) \approx d(q_p(x), q_p(y)) = \sqrt{\sum_j d(q_j(u_j(x)), q_j(u_j(y)))} \tag{2.10}$$

  where $d(q_j(u_j(x)), q_j(u_j(y)))$ is read from a lookup table, precomputed in an offline phase.

- **Asymmetric distance computation (ADC):** in this case, the query vector is not quantized, and the distance is approximated by:

$$d(x, y) \approx d(x, q_p(y)) = \sqrt{\sum_j d(u_j(x), q_j(u_j(y)))} \tag{2.11}$$

where $d(u_j(x), q_j(u_j(y))$ is read from a lookup table, precomputed for each query vector x in a preliminar (but online) step.

Figure 2.6 illustrates both scenarios.

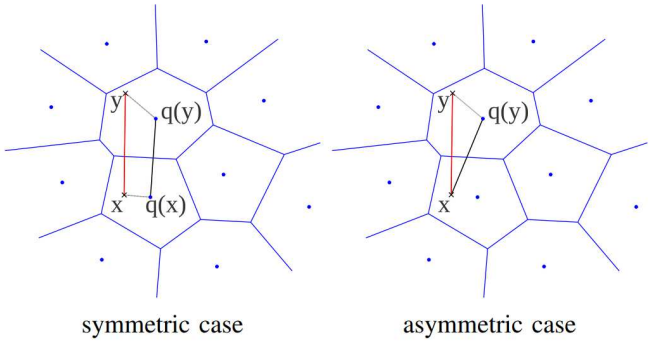

Figure 2.6. This image contrasts the symmetric and asymmetric distance computations. The image was adapted from Jégou et al. (2011).

Both methods have the same execution complexity, but ADC produces less distance distortion. The advantage of the SDC approach is that it uses less memory to store the queries, however, this is irrelevant in most cases, and the ADC approach should be preferred.

2.4.4 IVFADC

While the product quantization approach is very effective in reducing the memory usage, it still is an exhaustive approach, which doesn't scale well. One improvement proposed by Jégou et al. (2011) is to combine the ADC approach with an inverted file, resulting in the IVFADC approach. In this way, only a fraction of the database needs to be visited, improving performance drastically.

Each index in the inverted file corresponds to a centroid obtained from a coarse quantizer, which is independent from the product quantizer. And, instead of encoding the raw database vectors, the residual vectors (with respect to the coarse centroid) are encoded. In this way there is less distance distortion and the performance increases, at the cost of a few bytes per descriptor.

Let q_c be the coarse quantizer. The database vector y is approximated by:

$$y \approx q_c(y) + q_p(y - q_c(y)) \tag{2.12}$$

and, the distance between query vector x and database vector y is approximated by:

$$d(x, y) \approx d(x - q_c(y), q_p(y - q_c(y))) \tag{2.13}$$

which can be expanded to:

$$d(x, y) \approx \sum_j d(u_j(x - q_c(y)), q_j(u_j(y - q_c(y)))) \tag{2.14}$$

Again, the distances to the centroids are precomputed for each query vector x.

While it would be more precise to have a different product quantizer for each Voronoi cell, it would be much more expensive to compute, and it would need much more memory to store the codebook. Therefore, they chose instead to train a unique product quantizer for all voronoi cells.

When searching, instead of visiting all elements of the database, only the entries that are associated with centroids close to the query vector are visited. For each query, w indexes on the inverted file are visited, the w*th* closest to the query.

Figure 2.7 illustrates the database indexing and query processing phases of the IVFADC algorithm.

The indexing and query processing steps of the IVFADC are presented formally in Algorithm 1.

Algorithm 1: Indexing and Searching with IVFADC

```
1:  function Indexing(y)
2:      k' ← q_c(y)
3:      r_y ← y - C_c[k']
4:      code ← q_p(r_y)
5:      index[k'].append(y.id, code)
6:  end
7:  function Searching(x, w, k)
8:      nnc ← kNN(C_c, x, w)
9:      for i ∈ 1 ... w do
10:         k' ← q_c(x)
11:         r_x ← x - C_c[ncc[i]]
12:         distList ← dist(index[ncc[i]], r_x))
13:         ann ← kNN(ann, distList)
14:     end
15:     return ann
16: end
```

During the indexing, each feature vector y is quantized using the coarse centroids to find the list (k') in the inverted file in which it should be inserted (line 2). Furthermore, in lines 3 and 4, the residual value (r_y) of y to its coarse quantizer centroid is

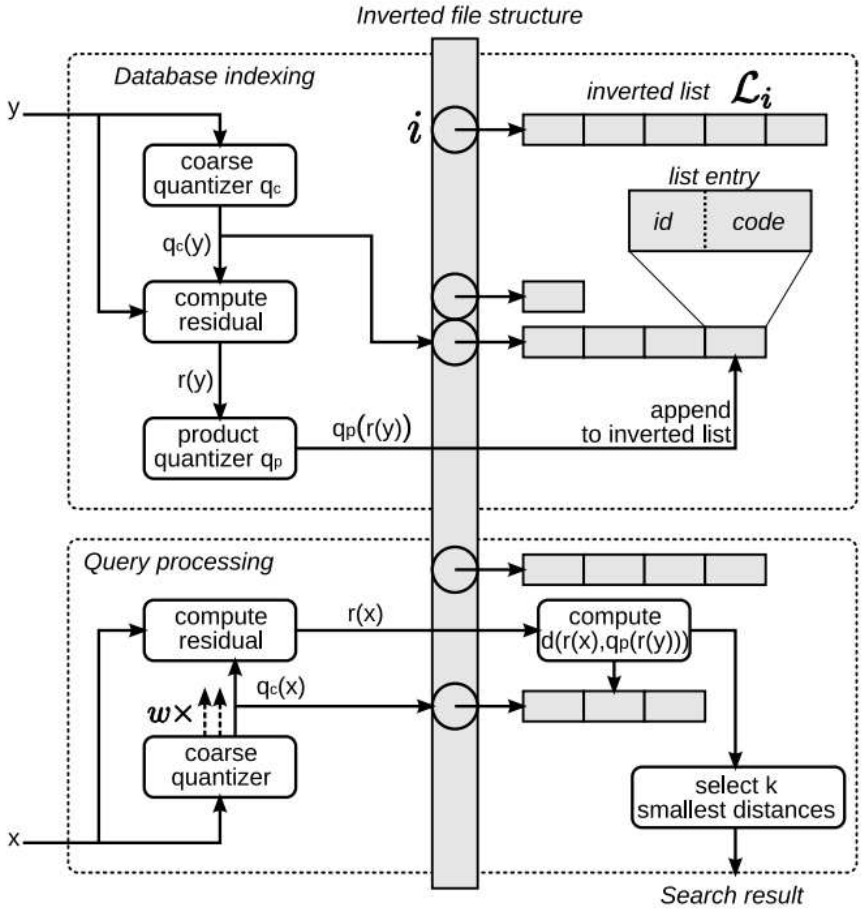

Figure 2.7. Overview of the database indexing and query processing phases of the IVFADC. The image was taken from Jégou et al. (2011).

computed, and r_y is quantized using the product quantizer q_p with the m subvectors and corresponding centroid sets. Finally, the identifier of the multimedia object described along with the product quantizer index (*code*) are inserted in the $L_{k'}$ inverted list in line 5.

The query processing procedure is presented in lines 7-16. It receives as input the query descriptor x, the number w of lists in the inverted file that should be searched, and the number of neighbors to return (k). The searched lists are those associated with the w nearest neighbors of x in the coarse codebook (C_c) as computed in line 8. Furthermore, for each of the w lists, the residual (r_x) of x to the coarse centroid associated to that list ($C_c[nnc[i]]$) is computed (line 11). The respective list in the inverted file is then accessed to compute the distance from r_x to the residual value of elements it stores (line 12). The elements with the k smallest distance values are selected and merged with nearest neighbors (ann) already found (line 13). After visiting

w lists, ann contains the final results and is returned.

2.5 Product Quantization Improvements

In this section we explore several subsequent improvements to the initial product quantization approach: inverted multi-index (Babenko and Lempitsky, 2015), optimized product quantization (Ge et al., 2013), locally optimized product quantization (Kalantidis and Avrithis, 2014), polysemous codes (Douze et al., 2016), and GPU IVFADC (Johnson et al., 2019).

2.5.1 Inverted Multi-Index

In Babenko and Lempitsky (2015), a new indexing structure was proposed: the inverted multi-index. The idea is to apply product quantization to the inverted indices of the traditional IVFADC algorithm. So, if using a k-order inverted multi-index, the input query would be divided in k equal pieces, and each one would be quantized independently to obtain the corresponding index in the inverted file. The index would be, then, a tuple with k elements. The classical inverted index can be thought of as a first order inverted multi-index. In this way, they achieve a denser subdivision of the search space, while still being very memory efficient. They claim they obtained shorter candidate lists at the same recall level.

When doing product quantization, usually more than one entry of the inverted file is visited, ordered by the distance of the respective coarse centroids to the query. However, in the multi-index case, since the number of possible indices (tuples) is much larger, computing the distance of the query to all tuples is unfeasible in practice. To deal with this, they introduced the multi-sequence algorithm: let this be a n-order multi-index. Then, the centroids associated with each of the n subdimensions are ordered with respect to the distance to the query. Let $c_{i,j}$ represent the i-th centroid closest to the query on the j subdimension. A priority queue of index tuples is created, ordered by the distance to the query, and initialized with the tuple $(c_{1,1}, ..., c_{1,n})$. At each step of the algorithm, the top tuple is popped (ie. the closest to the query) and more tuples are added to the priority queue. All of the "successors" of the popped tuple are considered for inclusion, but only those which all "predecessors" were already included are included. A tuple $(c_{a_1,1}, ..., c_{a_n,n})$ is a predecessor of the tuple $(c_{b_1,1}, ..., c_{b_n,n})$ if, and only if, for some i, $a_i + 1 = b_i$, and, for all $v \neq i$, $a_v = b_v$. If a tuple A is a predecessor of tuple B, then tuple B is a successor of tuple A. In this way, the tuples are visited

according to their distance to the query, while avoiding the need of precomputing and sorting the distance of all tuples to the query.

They tested the second and fourth-order inverted multi-index, and found out that the second-order would be a better choice in most situations.

2.5.2 Optimized Product Quantization

In the original Product Quantization approach, contiguous dimensions were quantized together. However, this is not the optimal approach. In Ge et al. (2013), they explored how to reduce quantization error by doing a better space decomposition scheme. They model the problem as an optimization problem over quantization error with the following free variables: a rotation matrix R, that represents the space decomposition, and the sub-codebooks for each subspace. They propose two approximate solutions: a non-parametric one, where they split the problem into optimizing for sub-codebooks, and optimizing for the R matrix, and optimize for each subproblem alternatively, and a parametric solution, which assumes that the data follows a parametric Gaussian distribution. They called their solution Optimized Product Quantization (OPQ). When encoding the database descriptors, we only need to multiply the descriptors by the R matrix, and then proceed with the product quantization as usual.

2.5.3 Locally Optimized Product Quantization

One of the limitations of the product quantization approaches described previously is that a lot of centroids might end up not having data associated with them in some multimodal distributions. Figure 2.8 illustrates this.

As can be seen, various centroids in the PQ and OPQ approaches ended up without support data. To address this, Kalantidis and Avrithis (2014) proposed the Locally Optimized Product Quantization (LOPQ) approach. In it, an individual product quantizer is optimized per coarse quantizer. Within a single coarse quantizer, the data distribution is largely unimodal, and, therefore, more suitable to be used with the OPQ approach (or alternatives). This results in less quantization distortion, but it has a large overhead both at offline (training) and online (query processing) phases. However, while large, it is constant in regards to the data size, which, when considered against large databases, is not significant. This approach can be combined with both the traditional inverted index and the multi-index.

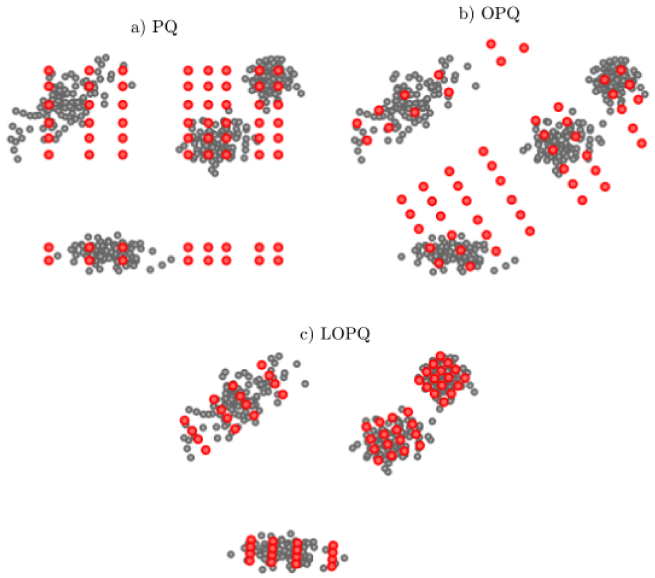

Figure 2.8. In this image, centroids (64) are shown in red, and were obtained by training with a random set of 2D points by three methods: a) product quantization (PQ), b) optimized product quantization (OPQ), c) locally optimized product quantization (LOPQ). Image was adapted from Kalantidis and Avrithis (2014)

2.5.4 Polysemous codes

In Douze et al. (2016), the idea of a polysemous code was introduced. The idea of multiple meanings comes from the dual nature of the codes obtained. They can be seen as either a binary code, which can be efficiently compared by using the Hamming distance, or as the typical product quantization based codes. After obtaining the centroids, in the usual product quantization fashion, their binary representation is changed, so that centroids that are close, have binary representations that are also close in the Hamming space. Figure 2.9 illustrates the dual nature of the obtained code.

When searching, first a pre-filtering based on Hamming distance is done, which is very fast, and, after that, the usual, and more expensive, asymmetric distance calculation is done. While training takes longer, the performance and accuracy improvements on the online phase is very significant, and, at the time, it could compete with state-of-the-art GPU aproaches (Wieschollek et al., 2016).

2.5.5 GPU-based IVFADC

The first GPU implementation of IVFADC able to handle a database with billion-scale descriptors was introduced in the work of Wieschollek et al. (2016). It used multi-level

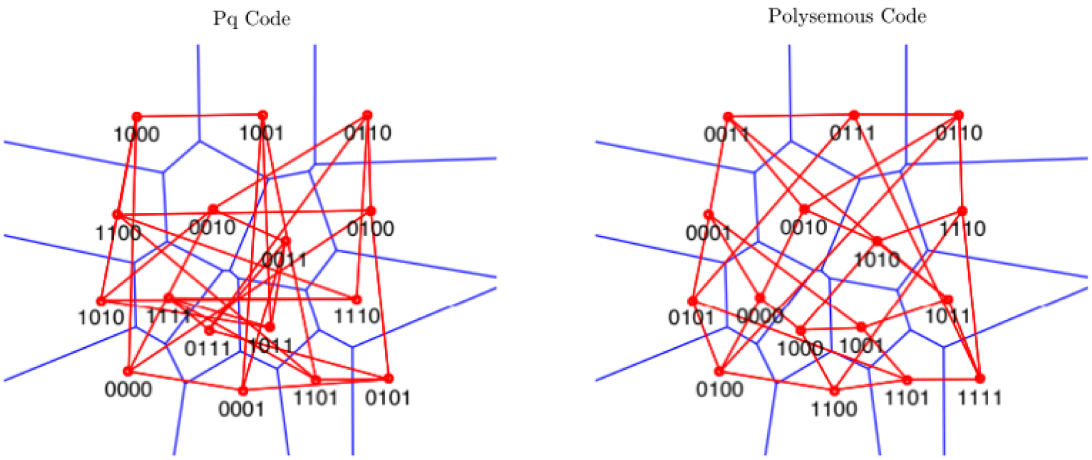

Figure 2.9. In this image, codes that have a Hamming distance of 1 are connected. Note how in the polysemous code, codes that are close in the euclidian space are also close in the Hamming space. Image was adapted from Douze et al. (2016)

quantization with a tree scheme to minimize the memory demands.

One of the main challenges with the efficient implementation of IVFADC to GPUs refers to the k-selection phase, which offers low parallelism opportunities in update operations and presents high warp divergence. These challenging aspects were addressed in Johnson et al. (2019) with a new efficient k-selection algorithm for GPUs and an optimized overall IVFADC implementation. With a variant of the Batcher's bitonic sorting network Batcher (1968), clever use of the GPU register file and warp shuffle instruction, this implementation of IVFADC attained significant performance improvements when compared to Wieschollek et al. (2016). As far as we know, the GPU-based IVFADC implemented in Johnson et al. (2019) is the state-of-the-art ANN GPU implementation. Also, it is available, open-source, in the Faiss library, developed and maintained by the Facebook AI Research group.

In this project, we used Faiss to handle the processing of queries in each node of the distributed system. We also used Faiss to train the database (ie. to obtain the centroids and quantize the database). In particular, we used its CPU and GPU IVFADC algorithm implementation. We choose Faiss because, among other reasons, it is well optimized, has good support from its developers, is in active development, and is used by many developers.

2.6 Distributed Similarity Search Algorithms

Although most of the ANN research has focused on optimizing the speed and quality of sequential algorithms, this is changing because of current demands for indexing very large databases. Multiple works have evaluated multi-core CPUs, GPUs, and distributed memory machines to perform large-scale multimedia similarity search: Bahmani et al. (2012); Stupar et al. (2010); Moise et al. (2013); Johnson et al. (2019); Cayton (2012); Kruliš et al. (2012); Wieschollek et al. (2016); Muja and Lowe (2014); Pan and Manocha (2011); Teodoro et al. (2014b); Andrade et al. (2019).

2.6.1 Distributed LSH

Distributed memory implementations of LSH were developed on top of MapReduce (Bahmani et al., 2012; Stupar et al., 2010). The approach of Stupar et al. (2010) stores the data into a distributed file system. Because LSH may require several hash tables, the data is replicated in each of the tables. This increases the query latency and reduces efficiency, making this solution impractical for large databases (Stupar et al., 2010). Bahmani et al. (2012) implemented a variant of the LSH using an Active Distributed Hash Table (DHT) to store the database in memory, and it assumes that a single LSH hash table is instantiated. In this case, multiple queries to the LSH table are required to attain the desired search quality as performed by multi-probe LSH (Lv et al., 2007), but it is not evaluated. In another relevant work, Moise et al. (2013) discussed the deployment of the Index Tree on top of MapReduce. This work has implemented a full search engine in a distributed memory system and discussed the technical challenges in this process. Similarly to the previous works, this solution is optimized for batch processing only.

2.6.2 Distributed FLANN

Muja and Lowe (2014) proposed a distributed memory parallelization of FLANN, using MPI (Message Passing Interface). They use a similar approach to MapReduce. Each node contains an equal fraction of the database. Once a query arrives at the master node, it is broadcasted to every other node. Each node process the query against its partition of the database, and then sends its results to the master node, which will aggregate all the results into a final result, doing essentially a reduce operation. Their approach, as discussed in Muja and Lowe (2014), suffers from high memory demands due to the algorithms implemented, limiting the scalability for large datasets.

2.6.3 Distributed IVFADC

In a previous work by Andrade et al. (2019), a distributed memory IVFADC was presented that executes on CPU-only machines. It works in a similar way to distributed FLANN, however, instead of having just one master node with the dual role of broadcasting queries and aggregating local results, it separates the two roles, and allows for more than one node to deal with each role. You could have, for instance, 8 nodes aggregating final results.

In that work, they adapted the system dynamically to the fluctuating query rate. In particular, they changed dynamically three parameters: inner parallelism, outer parallelism, and task granularity. Inner parallelism refers to how many computing cores are used to process each query individually. Outer parallelism refers to how many queries are computed concurrently. The product of the inner and outer parallelism is the number of available computing cores or threads. Task granularity refers to the number of queries that are sent to be computed at once, as a package. By changing those three parameters dynamically, they substantially reduced the average response time, when compared to the best static configuration.

2.7 Our work in context

The work in this dissertation is most related to the work of Andrade et al. (2019). In particular, our work extends that work in several new directions:

- we explore the use of GPUs, which have much higher performance on IVFADC compared to CPUs. GPUs have very different characteristics from CPUs, in particular, in regard to how to minimize the response time.
- we propose an approach to use, cooperatively, the GPU and CPU in this task.
- we tackle the problem of minimizing the response time in an environment where the query arrival rate fluctuates over time.
- we study how to handle databases that don't fit in the GPU memory and propose an optimized out-of-core execution scheme that takes into account response time when the query arrival rate fluctuates over time.
- we propose the use of work-stealing, to minimize the load imbalance between CPU and GPU in the out-of-core scenario.

- we use a much better implementation of the PQANNS algorithm, from Faiss, which brings new challenges. For instance, since our processing time is much smaller, we have much less headroom to make elaborate heuristics to reduce response time.

2.8 Summary

In this chapter, we introduced the concept of "similarity search", looked at one real-world example from "Google's Image Search", and gave a high-level view of how it works. Then, we looked at several popular image descriptors: SIFT, Gist, Bag of Visual Words, VLAD and Deep Features. We also explored several exact (Kd-Tree and Ball Tree based) and approximate (LSH, Randomized KD-Trees, Hierarquical K-means Tree, FLANN, Hierarchical Navigable Small World, Product Quantization) k-NN approaches. Since Product Quantization is the focus of our work, we looked at it in-depth, including some further advances in the area: inverted multi-index, optimized product quantization, locally optimized product quantization, polysemous codes and GPU based IVFADC. Furthermore, we discussed several distributed versions of approximate kNN algorithms, in particular: distributed LSH, distributed FLANN and distributed IVFADC. Last, we showed how our work relates and improves upon the work of Andrade et al. (2019).

In the next chapter, we will introduce and discuss our distributed architecture.

Chapter 3

Distributed PQANNS

In this chapter we explain the motivation and challenges behind creating a distributed version of the IVFADC, we introduce our overall distributed architecture, and then we explore each of its components individually: the offline phase, the query loader, the local index, and the global aggregator.

3.1 Overview

As mentioned earlier, the growth on the demand for multimedia search has been increasing dramatically over the past few years. As a consequence, a single machine usually does not have enough memory to handle such demands, even with the current strategy of using GPUs as processing accelerators, if we consider the limited memory sizes on typical GPUs. For instance, an NVIDIA P100 GPU has only 16GB of memory. Even if a single machine had enough memory, the response time would suffer, since, as the dataset size grows, the processing time also grows.

One simple solution to deal with this problem is to store the data in more than one computing node. In this way, we can either reduce the average response time or increase the amount of data that we are able to handle. Of course, this also brings new challenges and questions. First, we need to decide how to divide the data among the computing nodes. The simple solution is to divide the database into equal pieces and let each node have one piece. However, there is also the approach of replicating the database across all nodes, if the objective is to simply reduce the average response time. Another aspect that needs consideration is how the queries are going to be sent to the computing nodes, and how the local results produced by those computing nodes would be merged into a final result. Another fundamental aspect is deciding which underlying approximate k nearest neighbors algorithm to utilize. Some can be

more easily distributed than others. For instance, it is not trivial to create an efficient distributed version of the HSNW algorithm. Last, if each computing node is composed of more than one computing device, for instance, has more than one GPU, it might be necessary to create an intra-node parallelization strategy that will have to be well integrated with the overall distributed strategy.

Given the previous considerations, we will introduce our distributed architecture in the next section.

3.2 Distributed Architecture

We chose the IVFADC algorithm as the underlying approximate k nearest neighbors algorithm because it is one of the most efficient approaches memory-wise, which is necessary to deal with very large datasets, and also because it has a great GPU implementation, which helps in reducing the average response time. Another advantage of the IVFADC is that creating a distributed version is very straightforward, and suffers little overhead.

Regarding how to split the dataset, we chose to divide it into equal parts, which are scattered among the distributed nodes. This decision is motivated, again, by the desire to allow the processing of larger datasets. Every node will contain every entry of the inverted file, however, the elements within each entry will be distributed among the computing nodes. This is done to reduce potential load imbalances among the different nodes since every node will have to do roughly the same number of computations for every query.

With respect to the intra-node parallelization, we implemented both approaches: replicating the data and splitting the data. They will be explored in Chapters 4 and 5, respectively.

Our approach is very simple: the queries arriving at the system are forwarded to all distributed nodes storing partitions of the database, which are responsible for computing their local k-NN using the regular IVFADC algorithm and forwarding them to an aggregator node, which will aggregate the local k-NN results into a global k-NN result.

The architecture of the distributed IVFADC is presented in Figure 3.1. As shown, there are three stages or processes involved in the system: Query Loader (QL), Local Index (LI), and Global Aggregator (GA). Our design is scalable and allows the use of multiple instances of all stages.

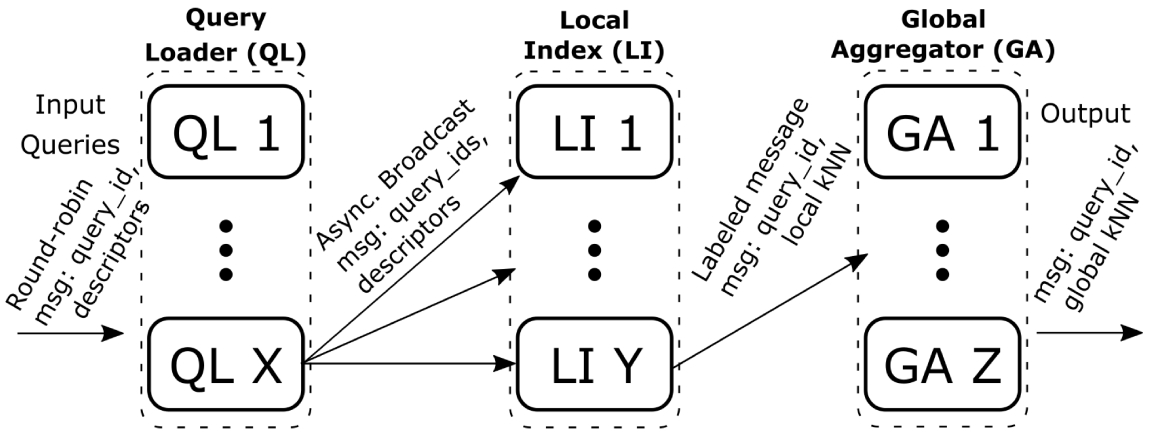

Figure 3.1. Distributed IVFADC indexing architecture.

Our implementation has been created using the Message Passing Interface (MPI) (Forum, 1994) for inter-process communication.

In the following subsections, each stage will be explained in detail.

3.2.1 Offline Phase

The first step is to train the product quantization centroids and the coarse centroids. We train it using the Faiss library, which allows the training to occur both in the CPU and in the GPU. After that, the dataset is divided among the computing nodes, each quantizes its dataset partition using the global product quantization centroids, and builds its own inverted file structure using the global coarse centroids. The local structure is called an index. This step occurs in an offline phase.

3.2.2 Query Loader

After the index is created, the actual search in the distributed system can start. The QL works as a front-end to the search engine and is responsible for receiving the input queries (e.g., from a web server) and to forward them to all LI processes. This is carried out using an asynchronous multicast to minimize synchronization overheads.

In order to reduce network overhead, queries are bundled together before being sent. For our tests, we found out that five was a good default since the network used in our experiments was very fast. However, this number can be easily changed in our system to better suit the network characteristics. This "bundling" propagates over the rest of the system, as queries are concurrently processed and bundled for communication from the LI stage to the GA stage.

The QL stage is implemented in a way to allow for more than one instance to be run at the same time, in order to increase the scalability of the system. Queries are sent to each QL instance in a round-robin fashion. The ith query will be sent to the $i \% X$ QL instance, where X is the number of QL instances. Each QL instance works independently of the other instances.

3.2.3 Local Index

Once an LI instance receives a query, it visits its local index, retrieves the local k-NN feature vectors, and sends its results to the GA process responsible for aggregating the results of that specific query. If the LI instance has more than one computing device (more than one GPU for instance), it might be necessary to divide the arriving queries among the different devices and to combine the results of each device into a final local result.

The queries are processed by calling the IVFADC implementation available on the Faiss library.

In order to allow for multiple GA instances, all LIs must direct messages related to a given query to the same GA instance. This routing is performed using a labeled communication channel between LI and GA (Teodoro et al., 2008). It employs a hash function to map a label ($query\_id$ in our case) to the GA instance that should receive the message. This function must return a value from $1 \ldots Z$, where Z is the number of GA instances, to be used in the message routing. Because each LI can take this decision independently, this is performed without any synchronization.

3.2.4 Global Aggregator

The GA receives the local results from the LI instances and merges them into a global k-NN output. To merge the local results, a priority queue is used, ordered by the distance to the query. There is one priority queue for every query. Again, multiple GA instances might be used to improve the scalability of the system. The results about each specific query are sent to a specific GA instance, as described in the previous subsection.

3.3 Summary

When dealing with huge datasets and high demand, a single machine is often not enough. Therefore, it is necessary to develop a distributed strategy of computation.

We chose to use the IVFADC algorithm as the basis of our system, since previous work has shown that it scales very well. The approach that we took to distribute the IVFADC algorithm was to divide the database into partitions of equal size, and let each local index node handle a partition. When processing a query, the query loader broadcasts the query to all local index nodes, which will compute a local result. Then, they send their local results to a global aggregator, which will aggregate the local results into a global one, and send it to the user. Each of the previous steps might be handled by one or more nodes.

One thing that was not touched in this chapter was how each local index will process the arriving queries. This is an important topic because a bad processing strategy might result in increased response times, specially when handling heterogeneous systems with GPUs and CPU. This topic will be addressed in detail in the next two chapters.

Chapter 4

In-Core Response Time Aware Distributed PQANNS

In this chapter we study how the block size (number of queries processed concurrently in the GPU) relates with the response time and query arrival rate, we introduce our Dynamic Query Processing Policy (DQPP), which tries to minimize the observed response time, and we explore how to use the CPU together with the GPU in this context.

4.1 Response Time

While throughput is important for online CBMR services that answer a large number of queries, their users are mainly concerned with the response time they experience with each submitted query. This represents a major challenge with these services, as improving throughput and reducing response times may be conflicting goals. Also, these systems deal with query arrival rates or workloads that vary during execution and, as such, must adapt to those changes at run-time.

The query response time observed by the user can be expressed by the following equation: $query\ response\ time = queue\ waiting\ time + processing\ time$. The queue waiting time refers to the time between the query being sent by the user and its processing beginning. While its processing doesn't start, it waits on a first-in-first-out queue. The processing time is, as the name suggests, the time it took for the query to be processed, once its turn comes.

Ideally, we want to minimize both. Minimizing the processing time is simple, just process one query at a time. Minimizing the queue waiting time, however, is not so simple. Simply processing every queue as soon as it arrives, while ideal, might not be possible if the throughput of the system is not enough to handle the rate at which

queries arrive. The problem is that the throughput of the system increases as the number of queries processed concurrently increases, up to a certain point. This is due to the fact that the GPU might not be fully utilizing all of its SMs, and, therefore, not achieving its maximum throughput. This is illustrated in Figure 4.1, which shows how the time per query decreases as the number of queries processed concurrently increases.

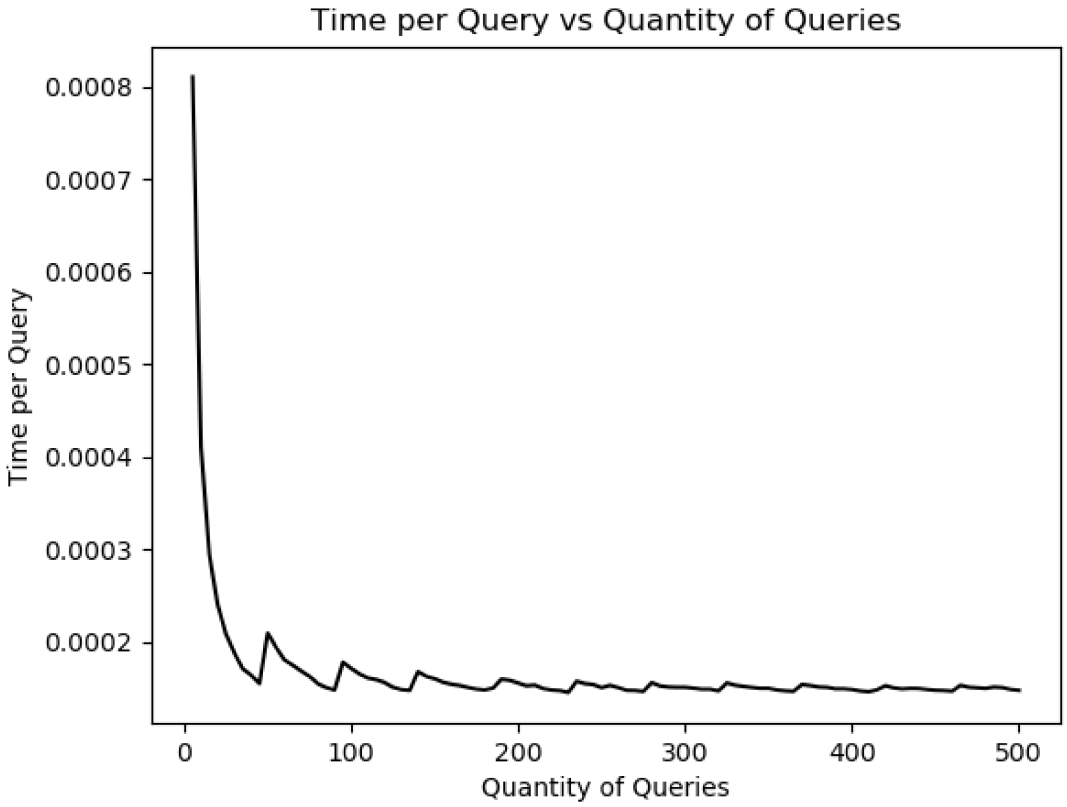

Figure 4.1. Time per query as the number of concurrent queries processed increases. This was measured using our distributed system, in the same conditions that it would encounter in normal usage.

So, we have a conflict: on one hand, we want to process smaller blocks of queries at once, and, on the other hand, we want to process bigger blocks of queries at once. The right answer depends on the query arrival rate. If the query rate is low, the throughput necessary to keep up with the arriving queries is also small, therefore, the ideal block size should be small. On the other hand, if the query rate is high, we need to increase the throughput of the system to keep up with the arriving queries (Menasce and Almeida, 2001), and, therefore, we should use bigger blocks.

4.2 Dynamic Query Processing Policy

In order to adapt the GPU processing block size at runtime, we propose a strategy called Dynamic Query Processing Policy (DQPP). It decides which block size to use based on the size Q of the input query queue (IQQ), the observed query arrival rate (QAR), and the expected GPU processing time (TPT) for a given block size. The QAR is computed dividing the number of queries arriving at the system by the time interval between subsequent calls to the GPU. The GPU processing times for different block sizes are obtained in an off-line benchmark and are stored in the TPT table.

Algorithm 2: Dynamic Query Processing Policy (DQPP)

```
1:  while True do
2:      IQQ.waitUntilNotEmpty()
3:      Q ← IQQ.length
4:      if Q ≥ B_g then
5:          process(IQQ, B_g)
6:      else
7:          TTB ← (B_g - Q) / QAR
8:          TFL ← TPT[Q] - TTB
9:          if TTB × Q > (B_g - Q) × TFL then
10:             process(IQQ, Q)
11:         else
12:             IQQ.wait(TTB, B_g)
13:             process(IQQ, min(IQQ.length, B_g))
14:         end
15:     end
16: end
```

Algorithm 2 presents the DQPP strategy. The main loop of the algorithm is executed by the GPU manager thread while there is work to do. It will block if IQQ is empty, as shown in line 2. If there are queries to be processed, it may process a block of queries already received or wait for more queries to arrive. If the number of queries ready is sufficient to execute with the block size configuration that leads to the best throughput (B_g), the algorithm dispatches B_g queries for the GPU execution. Please note that making a call with a larger number of queries is suboptimal as it would increase the query processing time of the queries in that block without improving the throughput.

When the number of queries Q in IQQ is smaller than B_g, it decides whether waiting for more queries to arrive to execute a larger number of queries in a GPU call is more efficient than dispatching the currently available queries for execution now. To

decide whether it is worthwhile waiting, DQPP estimates the weighted query waiting time increase in each case (wait or execute now), and selects the smallest one. Line 7 of Algorithm 2 computes the time estimate to have B_g queries ready for execution that is the configuration with the best efficiency. This is done by dividing the number of queries that should be received in this case (B_g-Q) by the query arrival rate (QAR). This is the additional waiting time that the Q queries ready to execute would pay to wait.

On the other hand, if the GPU executes now with the Q queries available, the ones arriving in the interval between the GPU call and its return will sit waiting in the queue. Then DQPP has to decide which waiting time weighted by the number of queries in each case is smaller. It then computes TFL that is the time interval between the B_g-Q queries received (TTB) and the time the GPU would finish if the Q queries ready to execute were processed (TPT[Q]) as shown in line 8 of Algorithm 2. Please notice that if TFL is smaller than zero, it means that the execution of the Q queries will end before B_g-Q are received. Thus, it is obvious that the Q queries should be dispatched for execution now. Further, having computed the waiting time of the Q queries (TTB) and the one of the B_g-Q queries (TFL), they are multiplied by the number of queries to select the configuration smallest weighted waiting time (line 9). If processing the currently queued queries is the best option line 10 is executed. Otherwise, the system will wait for TBB units of time or until IQQ.length (Q) $\geq B_g$, whatever occurs first, and process the queries queued, even if it is smaller than B_g. Setting this timer (TTB) avoids the algorithm from letting the GPU sit idle more than expected, which could be significant when the system is transitioning from high to low load scenarios.

4.3 Using the CPU together with the GPU

While the performance of the IVFADC on the CPU is relatively small compared to the GPU, it is not negligible. By using the CPU together with the GPU, we can increase the system throughput, and decrease the average response time, especially when the query rate is low. There are two basic approaches:

- **Mirror the data:** in this approach, the CPU holds a copy of the data in the GPU, and every query must be processed either on the CPU or on the GPU.
- **Shard the data:** in this approach, the local index is divided between CPU and GPU, and every query must be processed in both.

In the "Mirror the data" approach, we have to decide for every query whether to execute it on the GPU or on the CPU. We choose to divide the queries among CPU and GPU according to their respective throughput, measured in an offline phase. In this way, we expect that the time spent computing queries in the CPU and in the GPU to be roughly equal, and, therefore, the throughput maximized. While we are not explicitly minimizing the response time, our experiments showed that this approach produces a significant reduction in the average response time. How the CPU and GPU process their individual queries can be decided individually, without communication between them. For instance, in the GPU we simply use the DQPP method, while in the CPU we chose to use a greedy approach: whenever the CPU is idle, process all the queries waiting in the query queue at once.

In the "Shard the data" approach, every query will be processed in both CPU and GPU. Since the CPU is much slower than the GPU for product quantization, this means that the throughput and response time will suffer considerably. Therefore, the "Mirror the data" approach is usually preferable. However, by using the "Shard the data" approach, it is possible to handle datasets that would not fit in the total available GPU memory. Since CPU memory is much cheaper and more easily available than GPU memory, this is a scenario worth of consideration. In the next chapter, we will explore this scenario further.

4.4 Summary

How we process the arriving queries has a huge impact on the observed response time. For instance, if the query rate is high, processing one query at a time probably would result in a congestion in the query queue due to low throughput, which would affect negatively the observed response times. On the other hand, if the query rate is sufficiently low, processing one query at a time would achieve the best response times. Furthermore, since the query rate might vary over time, we can't simply use a static solution. In order to deal with those challenges, we introduced our Dynamic Query Processing Policy (DQPP) strategy. By looking at the near past to predict how many queries will arrive in the system, and by benchmarking how much time it takes to compute a block of queries of varying sizes, we estimate whether it is better to execute the queries that we currently have or to wait a little more. Furthermore, we explored how to use the CPU together with the GPU, in the case where the dataset fits in the available GPU memory.

In the next chapter, we will explore the case in which the dataset doesn't fit in

the available GPU memory, which brings its own set of challenges and optimization opportunities.

Chapter 5

Out-of-Core Response Time Aware Distributed PQANNS

In this chapter, we consider the scenario where the dataset does not fit in the available GPU memory. We compare several approaches to handle this, discuss their shortcomings, and propose a new solution that addresses those shortcomings.

5.1 Motivation

Given the large amount of data used by our target application, restricting the use of the GPU only to cases in which the device's memory is sufficient to store the whole index may be inefficient.

The simple solution would be, since the dataset does not fit in the GPU memory, to not use it, and use only the CPU, which is much cheaper memory-wise. However, the CPU has a much higher processing time than the GPU in this task, which leads to unacceptable response times.

The next logical improvement would be to put as much as possible of the dataset in the GPU, and put the rest in the CPU. While this drastically increases the performance of the system, it still is bottlenecked by its CPU component. In fact, at a 50% - 50% division of the dataset, the processing time would be reduced by, at most, half, which, while a huge improvement, is not enough to produce acceptable response times.

Another approach would be to ignore the CPU altogether and use only the GPU. In this approach, since the dataset doesn't fit entirely in the GPU memory, the partition on the GPU would need to be switched with one from the CPU from time to time. While this approach reduces the response time substantially at high loads when

compared to the CPU-GPU fixed division approach, it suffers at low query rates, since every query will have to pay the fixed cost of the (partial) dataset transfer to the GPU.

In order to address the problem with the GPU only approach, and create a solution that works well at both low query rates and high query rates, we propose in the next section an out-of-core approach that uses the CPU and GPU and allows the GPU to steal some partitions from the CPU when deemed profitable.

5.2 Out-of-Core CPU-GPU with Work Stealing

In our proposed approach, the index partition assigned to an LI stage instance is subdivided into smaller, non-overlapping subpartitions. Queries are processed against each subpartition, and the k-NN results from each subpartition are merged. This strategy is similar to that used in the distributed-memory parallelization, where each node holds a partition of the database, and the kNN results of each node are merged to form the final result for each query. In other words, we perform a hierarchical parallelization with multi-level partitioning, which adapts to the computing system at each level: distributed memory and local node.

In our approach, there is a queue of input queries to be processed and a list holding k-NN results already computed for each subpartition. Once a subpartition is assigned to the CPU or GPU, it then calls the IVFADC implementation (from Faiss) to process a block of queries from a subpartition. The CPU-GPU cooperative execution is interesting in this context because of the relative performance of CPU vs. GPU (or speedup attained by the GPU) may vary according to the system load (i.e. queries available for processing) and the number of subpartitions the GPU can hold simultaneously. For instance, the larger the number of queries to be evaluated on a subpartition, the higher tends to be the GPU efficiency of that task. Consequently, performing a static division of the subpartitions to be processed by each device would be suboptimal in this case.

We address this problem with an initial work partitioning between CPU and GPU in which the GPU is assigned the maximum number of subpartitions it can store, because, as observed experimentally, the GPU is much more efficient than the CPU, and fully utilizing it results in the best performance. The subpartitions assigned to the devices are stored either in the $workTasks.GPU$ or $workTasks.CPU$ lists and the GPU will preferably process queries related to a partition it owns to avoid unnecessary data transfers.

To reduce imbalances during execution that may result from this initial work

partitioning, the GPU sometimes steals subpartitions assigned to the CPU. Since the cost of stealing a subpartition is high, due to the fact that we need to transfer the subpartition from the CPU memory to the GPU memory, it should only be done when there are enough queries waiting to be processed to make it worthwhile.

Algorithm 3: GPU Manager Thread Control Flow

```
1:  function GPU_Thread(workTasks)
2:      while True do
3:          for task ∈ workTasks.gpu do
4:              processGpu(task)
5:          end
6:          if workTasks.gpu.largest() != 0 then
7:              continue
8:          end
9:          if workTasks.cpu.largest() ≥ threshold then
10:             workTasks.swapTasks(CPU2GPU)
11:         end
12:     end
13: end
```

The overall execution scheme is shown in Algorithm 3. First, from lines 3 to 5, the GPU manager thread will iterate over all subpartitions assigned to the GPU and process them. Once it has finished, it checks whether new queries have arrived for one of the subpartitions it owns. If no work is available for the GPU, it checks if it is worthwhile stealing a subpartition currently attributed to the CPU.

The stealing condition is shown in line 9, and, when it is true, we swap one of the GPU subpartitions with one of the subpartitions owned by the CPU (line 10). We choose the one that has the largest amount of queries waiting to be processed.

The CPU manager control flow is not presented, but it would consist of processing subpartitions assigned to the CPU only. It is also important to highlight that we assume that the entire index (subpartitions) is stored in the CPU memory for simplicity and efficiency. In this case, releasing space in the GPU for a new subpartition is efficient, as it would not require copying data back to the CPU memory, which is expensive.

Determining the right value for the *threshold* value is not trivial. In fact, a bad choice on the threshold value might lead to a decrease in performance, since it might lead to excessive data transfers from the CPU memory to the GPU memory. The optimal *threshold* value clearly depends on the CPU throughput and on the time it takes to transfer a partition of the dataset from the CPU memory to the GPU memory.

In the next section, we will explore, from a mathematical perspective, what would be the ideal *threshold* value.

5.3 Work Stealing Threshold

Stealing a subpartition from the CPU has a high overhead since it is necessary to transfer that subpartition to the GPU. Therefore, it is profitable only when we have a large number of queries ready to be executed on that subpartition. The variable *threshold* represents the minimum number of queries required for the stealing to be worthwhile.

Stealing will occur when the GPU is idle, and the number of queries to be executed in the CPU is sufficiently high.

It would be worthwhile to steal that subpartition if it resulted in a reduction in the queries' average response time. Thus, we need to derive the response times with and without the GPU stealing a partition to understand when it would be beneficial. If all queries (Q) available were processed by the CPU using a block size with maximum throughput (B_c), which takes time T_c to be processed in the CPU, the average response time in the CPU (R_c) would be:

$$R_c = \frac{1}{Q} \sum_{i=1}^{\frac{Q}{B_c}} B_c * T_c * i \tag{5.1}$$

Basically, the ith block of B_c queries will finish in time $i * T_c$, since it will have to wait $(i-1)$th blocks to finish (in $(i-1) * T_c$ time), and itself will take T_c time to be processed. Equation (5.1) can be simplified to:

$$R_c = \frac{1}{2}\left(\frac{Q}{C} + T_c\right) \tag{5.2}$$

where $C = B_c/T_c$ is the CPU maximum throughput. If, instead, the GPU steals a subpartition from the CPU, the response time (R_g) would be:

$$R_g = L + \frac{1}{2}\left(\frac{Q}{G} + T_g\right) \tag{5.3}$$

where L is the time it takes to transfer the subpartition to the GPU, B_g the block size such that the GPU throughput is maximum, T_g the time it takes to execute B_g queries in the GPU, and $G = B_g/T_g$ the maximum GPU throughput. Thus, it would be worth performing the stealing when:

$$R_g < R_c \tag{5.4}$$

By substituting (5.2) and (5.3) into (5.4), and manipulating the resulting expression, we obtain:

$$Q > \frac{2 * C * G}{G - C} * \left[L - \frac{1}{2} * (T_c - T_g) \right] \tag{5.5}$$

In other words:

$$threshold = \frac{2 * C * G}{G - C} * \left[L - \frac{1}{2} * (T_c - T_g) \right] \tag{5.6}$$

Note that we assume that $G - C > 0$, i.e. the GPU throughput is higher than the CPU throughput.

While we use that exact *threshold* expression in our algorithm, it can be simplified to lead to a better understanding of its meaning. In practice, T_c and T_g are very small when compared to L (the data transfer time). By making the following approximation:

$$L - \frac{1}{2} * (T_c - T_g) \approx L \tag{5.7}$$

we can simplify the threshold expression to:

$$threshold \approx \frac{2 * C * G * L}{G - C} \tag{5.8}$$

The resulting expression coincides with our intuition about the conditions necessary for the stealing to be profitable: the higher the CPU throughput, the more queries waiting for processing in a subpartition are necessary. Also, the more time it takes to transfer the subpartition from the CPU to the GPU (L), the higher is the number of queries that need to be ready to be processed. Furthermore, it is interesting to note that if the GPU throughput is much higher than the CPU throughput ($G >> C \rightarrow \frac{G}{G-C} \approx 1$), we could simplify the threshold expression to:

$$threshold \approx 2 * C * L \tag{5.9}$$

We have compared (5.6) to (5.9) in practice, and have noticed that the difference in the obtained thresholds were smaller than 5%. Further, this difference led to an insignificant impact on the performance of the system.

5.4 Asynchronous CPU/GPU data transfers

Finally, we want to mention that we have evaluated the use asynchronous data transfers between CPU and GPU in our application. The execution of asynchronous CPU/GPU data transfers, however, requires the availability of a free or reserved space in the GPU memory for it to take place while computation is carried out in another data chunk already in the device memory. In our application domain, the asynchronous transfers makes sense only in the out-of-core case, but in this scenario, the extra space that would be necessary for the asynchronous transfer occupies an area that would otherwise be used by the application. Thus, we implemented a double-buffering scheme to evaluate whether asynchronous data transfers would be worthwhile in our case, and we have noticed that it does not improve the performance of our application. The limitation here is that, for instance, when 50% of the data fits in the GPU memory, we would need to have two partitions of the GPU memory to store 25% of the data that is being processed while the other is used for data transfer. However, computing multiple smaller partitions with 25% of the index data is more expensive than processing a single larger partition with 50% of the data with a single GPU call. This computational overhead offsets the gains with the data transfer optimization, making asynchronous data transfers not worthwhile in our case.

5.5 Summary

In this chapter we motivated the need of an out-of-core approach, and explored some simple approaches: CPU Only, CPU-GPU fixed division and GPU Only. We argued that the CPU Only approach produces unacceptable response times, and that the CPU-GPU with fixed division is still bottlenecked by the low throughput of the CPU on high query rates. We also argued that the GPU Only approach would produce bad response times at low query rates, since every query would have to a pay the (large) fixed cost of the (partial) dataset transfer to the GPU. In order to address the limitations of the GPU Only approach, we introduce our CPU-GPU fixed division with work stealing approach. The main idea is that, when the query queue of the CPU gets too big, the GPU steals a data partition from the CPU in order to increase the throughput of the system. This is done only when we can show that doing so would improve the average response time. In this way, our system behaves like the GPU-Only when the query rate is high, and like the CPU-GPU with fixed division when the query rate is low, achieving the best of both worlds.

In the next chapter, we will put our theories into test, evaluating both our in-core

and out-of-core approaches under several scenarios, with respect to both throughput and response time.

Chapter 6

Experimental results

In this chapter, we will describe the machine and cluster configurations used in our experimental evaluation in Section 6.1, we will compare the performance of the single-node IVFADC of Faiss against FLANN and exhaustive search in Section 6.2, introduce the dataset and index configuration used in Section 6.3, and show the performance, in terms of throughput and response time, of our In-Core and Out-of-Core approaches in Sections 6.4 and 6.5, respectively. Last, we will analyze the scalability of our approach, running it in a cluster with up to 256 GPUs and about 30TB of data in Section 6.6.

6.1 Machine and Cluster configuration

Our single node experimental evaluation was carried out on a machine running Linux, equipped with a NVIDIA P100 GPU, the Intel Broadwell E5-2683 v4 CPU, and 128GB of RAM. This machine was used in the experiments performed from Section 6.2 up to Section 6.5. We used OpenMPI version 2.1.2.

Our scalability tests, presented in Section 6.6, were done on 64 computing nodes interconnected using EDR InfinitBand, each of them equipped with 4 NVIDIA V100 GPUs, 2 IBM Power 9 (AC922) CPUs, and 320 GB of RAM. We used Spectrum MPI. We tested the scalability of our system with a database with up to 256 billion SIFT descriptors, but other descriptors, such as VLAD (Jégou et al., 2010), would also benefit from our optimizations.

6.2 IVFADC vs. FLANN and Exhaustive Exact k-NN

The FLANN Muja and Lowe (2009); Muja and Lowe (2014) implements several ANN algorithms, as described before, and is able to choose the most efficient for a given database. As such, it is a good reference for baseline performance comparisons. Thus, here we compare IVFADC to FLANN to demonstrate IVFADC good quality vs. speed trade-off. The evaluation uses a SIFT database with 1 million descriptors, 10K queries, and measures the 1-recall@1, or the average proportion of the NNs ranked first in the returned results, also referred to as precision Muja and Lowe (2009). The parameters of FLANN were selected automatically, while we varied w and the number of coarse centroids (shown as w/# of centroids) as shown in the graph. Further, the same experiment was executed on the exact exhaustive k-NN using the Yael library, which employs BLAS/LAPACK underneath to attain high performance. The exact k-NN took 1048.4 seconds or about 0.1 seconds per query.

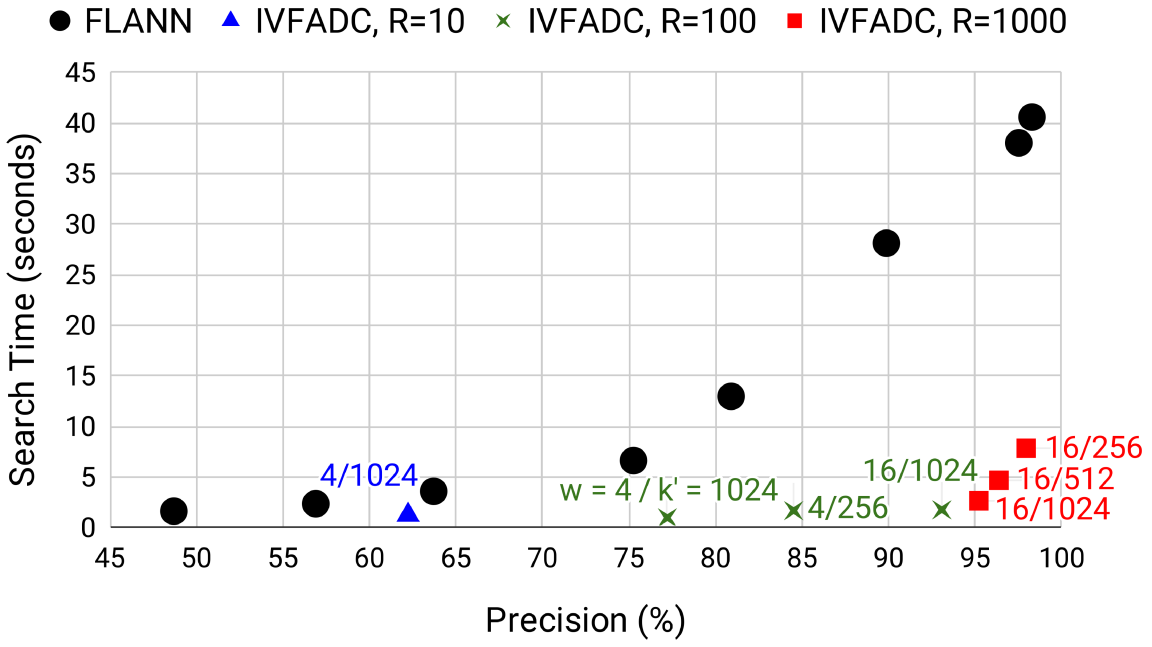

Figure 6.1. IVFADC vs. FLANN: trade-offs between search quality (precision) and search time using CPU.

The results comparing IVFADC and FLANN as precision varies are shown in Figure 6.1. As may be observed, for the same search quality, IVFADC can execute significantly faster than FLANN, and their performance gap increases with higher

precision. It is also impressive that FLANN uses 600 MB of memory, whereas IVFADC requires only 26 MB. For 98% precision, the amortized search time per query with IVFADC is less that 0,00078 sec or 127× faster than the exact k-NN.

6.3 Datasets and Index Configuration Used in the In-Core and Out-of-Core tests

For the single node In-Core and Out-of-Core experiments, we used the SIFT500M dataset, introduced in Jégou et al. (2011), which consists of 500 million 128 dimensional SIFT vectors. We also tested on the Tiny Images dataset, introduced in the work of Torralba et al. (2008), which contains roughly 80 million 384 dimensional GIST descriptors.

For the scalability tests, we replicated the SIFT1B dataset (similar to SIFT500M, but with 1 billion vectors) across up to 256 GPUs and 64 CPUs, reaching a total of up to 30TB of data and 256 billion SIFT vectors.

The IVFADC has been configured to use a coarse codebook with 4096 coarse centroids, 8 subquantizers, and 256 centroids per sub-dimension. This configuration attained a precision of 76% on the SIFT500M dataset, and 67% on the Tiny Images dataset. This configuration was used because it achieves a good compromise between favoring the GPU and favoring the CPU while maintaining a reasonable precision.

We argue that our system would work well with other datasets or IVFADC configurations, but we have not tested them in this work.

6.4 The Performance of the In-Core Response Time Aware Distributed PQANNS

In this section, we will show the baseline single node throughput of our implementation on multi-core CPU, GPU, and cooperative CPU-GPU execution on Subsection 6.4.1, and we will compare our DQPP policy against static block sizes and greedy policies, with regards to response time, on Subsection 6.4.2.

6.4.1 Throughput

This section presents the baseline single node throughput of our implementation on multi-core CPU, GPU, and cooperative CPU-GPU executions using our single-node machine.

100k queries were sent for execution at the beginning of the experiments, in groups of 5 queries, one group at a time.

Table 6.1. Throughput (queries/s) with different configs.

Dataset	CPU only	GPU only	CPU-GPU
SIFT 500M	801	7417	8069
Tiny Images	4119	36451	39558

The performance is presented in Table 6.1 according to the system configuration used in the LI phase of the parallel algorithm: CPU only, GPU only, or CPU-GPU. The CPU only execution employs the 16 CPU cores available, and the CPU-GPU maintains a copy of the index in each device and divides queries between CPU and GPU proportionally to their relative performance, as detailed in Chapter 4. The query rate attained is very high in all cases with the GPU being about 9× faster than the CPU. The CPU-GPU execution, in turn, attains a speedup of 1.1× on top of the GPU on the SIFT500M and Tiny Images dataset. The total throughput is slightly smaller than the sum of the processors' throughput because (i) a CPU core is reserved to manage the GPU; and, (ii) there are inevitable overheads, including load imbalance, query partitioning, etc.

6.4.2 Response Time

We evaluate GPU and CPU-GPU executions with multiple query processing strategies in on-line scenarios with fluctuating loads.

Table 6.2. Average query response time (secs.) with varying query rates using a Poisson distribution (with average Q/s queries per second) on the SIFT500M dataset with static block sizes using GPU only.

Q/s	025	050	075	100	125	150	175	200
600	0.026	0.046	0.067	0.088	0.108	0.130	0.150	0.172
2200	0.078	0.059	0.050	0.059	0.061	0.069	0.074	0.086
3800	0.365	0.204	0.131	0.139	0.123	0.131	0.128	0.140
5400	1.499	0.822	0.492	0.496	0.362	0.395	0.336	0.361
7000	3.027	1.979	1.216	1.217	0.915	0.983	0.844	0.899

In Table 6.2 we show how the average query response time depends on both the query rate and block size. As can be seen, the best block size varies as the query rate

varies. In particular, at 600, 2200, 3800, 5400 and 7000 queries per second, the best block sizes were, respectively, 25, 75, 125, 175 and 175.

In Table 6.3, our DQPP policy (Section 4.2) is compared to the Best Static (BS) and Dynamic Greedy Dispatch (DGD) policies.

The BS policy uses a static block size that tries to achieve a good response time at different query rates. To obtain this block size, we followed a simple algorithm: given a table like Table 6.2, we attribute to each cell, a number from 1 to n, where n is the number of block sizes. This value represents how low the response time of that block size is when compared to the other block sizes at that query rate. For instance, the cell with query rate 2200 and block size 50, would be filled with the number 2, since only one other block size (75) produces a smaller response time at that query rate. In case of a draw, average the involved values. Once this is done, we take the average of the values in each column, and it will be the score associated with the corresponding block size. For Table 6.2, the best block size would be 125, with a score of 3.2. While it is the best only at 3800 queries per second, it produces reasonable results at all query rates, when compared to other static block sizes.

DGD is a dynamic approach that calls the GPU execution for all queries available in the query queue whenever the GPU becomes idle. Although DGD would be able to adapt to fluctuating workloads, it does not include the smart mechanisms of DQPP that also looks into the near past to decide whether to call or not the GPU for a given queue size.

The fluctuating query workload is implemented using a Poisson distribution with an expected average rate Q/s.

Table 6.3. Average query response time (secs.) with varying query rates using a Poisson distribution (with average Q/s queries per second) for GPU only and CPU-GPU execution on the SIFT500M dataset.

Q/s	GPU			CPU-GPU		
	BS	**DGD**	**DQPP**	**BS**	**DGD**	**DQPP**
600	0.108	0.011	0.011	0.109	0.013	0.013
2200	0.061	0.034	0.028	0.057	0.027	0.025
3800	0.123	0.151	0.097	0.105	0.114	0.082
5400	0.362	0.876	0.281	0.235	0.392	0.200
7000	0.915	2.856	0.772	0.652	1.502	0.569

The average query response times of the strategies are presented in Tables 6.3 and 6.4.

Table 6.4. Similar to Table 6.3, but using the Tiny Images dataset. The BS block size is 125 in this case, with a score of 3.2. Coincidentally, it is the same result obtained for the SIFT500M dataset.

Q/s	GPU			CPU-GPU		
	BS	**DGD**	**DQPP**	**BS**	**DGD**	**DQPP**
6800	0.0111	0.0021	0.0020	0.0120	0.0030	0.0029
13600	0.0107	0.0103	0.0064	0.0115	0.0083	0.0069
20400	0.0177	0.4830	0.0155	0.0182	0.0375	0.0157
27200	0.0330	0.9637	0.0341	0.0327	0.6698	0.0328
34000	0.0813	1.8714	0.0899	0.0652	0.9280	0.0712

The DGD policy had good results at low query rates, but was unable to deal with higher query rates. We believe this happens due to it getting stuck in large block sizes when operating at high query rates, due to the fact that, after some block size, the time per query stops decreasing, as can be seen in Figure 4.1. This makes the DGD policy vulnerable to peaks in the query rate.

The BS policy, which in our scenario corresponded to a static block size of 125, produced good results overall, but couldn't match the DGD policy at low query rates. This is due to the fact that, at such low query rates, the DGD policy can simply process all queries that arrive as soon as they arrive, therefore minimizing both the queue waiting time and the query processing time. With a block size of 125, the BS approach has to wait more time before being able to process a block of queries, and, also, the processing time itself is higher. Therefore, the average response time suffers. Note that this is only true because we are assuming that the query queue will not be congested due to the low query rate.

The DQPP approach performed as well as the DGD approach at low query rates, and as well as the BS approach at high query rates, achieving our goal of working well at different query rates. Notably, it was as good as the BS and DGD approaches at all tested query rates in the SIFT500M experiments. We believe it performed better in the SIFT500M dataset than in the Tiny Images dataset because it is a bigger dataset, and, therefore, the overhead of the DQPP approach is smaller relatively.

Also, the use of the CPU together with the GPU was able to reduce the average response time in up to 29% in the DQPP approach, which is higher than the corresponding increase in the throughput of 9%. As expected, the overhead of using the CPU at low query rates outmatched its benefits. At low query rates, the queue waiting time is almost 0, and, therefore, the query processing time is more relevant. Since the processing time of the CPU is higher, it follows that the response time would increase.

However, while it increased, the increase was so small that we argue that it is irrelevant in practice.

6.5 The Performance of the Out-of-Core Response Time Aware Distributed PQANNS

This section evaluates our out-of-core execution scheme with GPU only and CPU-GPU. It employs the same datasets and IVFADC configuration used in the previous section. However, for the sake of analyzing the out-of-core effects to the performance, the amount of data that may be kept in the GPU memory at any time during the execution is limited. Furthermore, this amount is varied with the purpose of analyzing multiple GPU memory capabilities and evaluating the performance in different scenarios. The choice of using the same datasets and IVFADC configuration as employed in the previous section here is intended to compare the out-of-core configurations directly to the in-core and, consequently, analyze the penalty in performance with this approach. However, in Section 6.6, we an execute experiment in which a dataset with 1 billion SIFT vectors is used per GPU, while only half of it fits in the device memory.

6.5.1 Throughput

This section presents the baseline single node throughput of our out-of-core implementation, under several GPU memory capacities using our single-node machine. 100k queries were sent for execution at the beginning of the experiments, in groups of 5 queries, one group at a time.

Table 6.5. Throughput (queries/s) in an out-of-core execution using the SIFT500M dataset.

% of data in GPU	GPU only	CPU-GPU Fixed Division	CPU-GPU Work Stealing
12.5%	4739	914	5353
25%	6627	1066	6367
50%	6987	1586	7679

The throughput of the system is presented in Tables 6.5 and 6.6 as the percentage of the index that would fit in the GPU memory is varied from 12.5% up to 50%. As shown, the GPU-only configuration attained significant performance even in cases in

Table 6.6. Same as Table 6.5, but using the Tiny Images dataset

% of data in GPU	GPU only	CPU-GPU Fixed Division	CPU-GPU Work Stealing
12.5%	8462	4633	10765
25%	14476	5380	17315
50%	21493	7767	23861

which the amount of the index it can hold up is small (12.5%) and, as expected, its performance improves as more data fits in memory. The GPU only out-of-core execution reached a throughput of up to 94% compared to the GPU only in-core throughput. Also, it is up to 8.7× faster than using only the CPU, which would be the reference for processing this dataset without the out-of-core strategy.

Further, it is presented the performance of the CPU-GPU execution with a fixed data division and using work-stealing. In the fixed division case, because the GPU can only process the index subpartition assigned to it, the CPU becomes the bottleneck and the cooperative execution is ineffective. However, the CPU-GPU with work stealing improved the GPU-only in all cases and attained a speedup of up to 1.27× compared to the GPU-Only approach.

6.5.2 Response Time

In this section, we evaluate our out-of-core approach with respect to the obtained average response time. Again, we compare it with the CPU+GPU Fixed Division strategy, the GPU only strategy, and our CPU+GPU Work Stealing strategy.

This section evaluates our out-of-core approach by varying the amount of data (% of the index) that is stored in the GPU memory, while the query rate is varied in a similar manner to the previous section. The results are presented in Tables 6.7 and 6.8, in which the CPU only execution is also included for reference.

As presented, the average response time across the board are higher than in the case with the in-core data. In general, our approach with work-stealing was the best performer, followed by the GPU only approach, the CPU-GPU fixed division and the CPU only.

The CPU-GPU fixed approach generated much better results than the CPU only approach, however, its overall performance was lackluster. This is due to the low throughput of the CPU part, which bottlenecked the system, since every query must be processed in both CPU and GPU.

Table 6.7. Average query response time (secs.) with varying query rates using a Poisson distribution (with average Q/s queries per second) for GPU only and CPU-GPU execution on the SIFT500M dataset.

% of data in GPU	Q/s	CPU only	GPU only	CPU-GPU Fixed	CPU-GPU Work Stealing
12.5%	1400	29.35	8.36	29.11	5.48
	2800	45.86	10.85	45.63	7.97
	4200	50.95	13.19	50.86	10.25
	5600	57.70	14.79	53.18	11.84
	7000	59.36	15.91	54.89	13.24
25%	1400	29.35	5.56	21.65	3.22
	2800	45.86	6.86	39.37	4.68
	4200	50.95	8.92	43.19	6.66
	5600	57.70	10.20	45.58	8.14
	7000	59.36	10.91	47.05	8.75
50%	1400	29.35	4.14	6.50	1.57
	2800	45.86	5.67	21.45	3.32
	4200	50.95	6.50	26.67	4.45
	5600	57.70	8.30	29.32	6.07
	7000	59.36	8.18	31.013	7.52

Table 6.8. Same as Table 6.7, but using the Tiny Images dataset

% of data in GPU	Q/s	CPU only	GPU only	CPU-GPU Fixed	CPU-GPU Work Stealing
12.5%	6800	6.59	6.68	5.03	3.21
	13600	9.88	8.40	8.40	5.17
	20400	11.04	9.05	9.60	5.95
	27200	11.65	9.46	10.15	6.39
	34000	12.00	9.75	10.51	6.54
25%	6800	6.59	2.83	3.41	1.68
	13600	9.88	3.94	6.79	2.49
	20400	11.04	4.54	7.94	3.13
	27200	11.65	4.89	8.55	3.68
	34000	12.00	5.12	8.81	3.81
50%	6800	6.59	1.40	0.89	0.45
	13600	9.88	1.82	3.67	1.02
	20400	11.04	2.29	4.75	1.63
	27200	11.65	2.50	5.35	1.66
	34000	12.00	2.66	5.69	1.89

While the GPU only approach was able to produce good average response times at high query rates, it did not perform as well at lower query rates. This is due to the fact that, for every query, its response time has to include, at least, the time to transfer the parts of the dataset that are not in the GPU memory from the CPU memory. In our tests, this meant transferring from 50% of the dataset up to 87.5% of it. Since this transfer time is rather large, it follows that, at low query rates, the increase on the response time would be very significant.

The CPU-GPU with work-stealing produced the lowest average response times. It performed much better than the GPU Only at low query rates, achieving a reduction of the average response time of up to 3.1×. It performed only slightly better at higher query rates, due to the fact that, at high query rates, the best choice is to increase the throughput, which means processing as much as possible on the GPU. Still, since we are able to use the CPU while the GPU is busy with the data transfer, our CPU-GPU with work-stealing approach was able to slightly decrease the average response time on those cases.

In Tables 6.9 and 6.10 we can see how our smart work stealing policy was able to reduce the amount of data transfers between CPU and GPU, which was fundamental in reducing the average response time observed in our approach, when compared to the GPU Only approach. This is specially true when the query rate is small, as in those cases, doing excessive transfers would have a much higher impact on the average response time.

6.6 Distributed Memory Scalability: In-core and Out-of-Core

In this section, we evaluate in-core and out-of-core executions of our systems in a distributed memory, multi-GPU configuration. Our experiments focus on a weak scaling scenario in which the dataset and the available computing resources are increased in the same proportion. We used this experiment design because in this application domain, the system is expected to deal with very large and ever-increasing datasets that would not fit into the memory of a single GPU or computing node. The experiments were performed by varying the number of computing nodes from 1 up to 64 (each compute node is equipped with 4 GPUs and 44 CPU cores). The dataset used contains 500 million and 1 billion SIFT descriptors per GPU, respectively, for the in-core and out-of-core tests. As a consequence, we have indexed and searched up to 256 billion SIFT descriptors in the case of the out-of-core with 64 nodes/256 GPUs, which cor-

Table 6.9. Average number of dataset partition transfers between CPU and GPU on the SIFT500M dataset. Each test was executed ten times. The standard deviation is listed between parentheses.

% of data in GPU	Q/s	GPU only	CPU-GPU Work Stealing
12.5%	1400	61.8 (±2.4)	42.6 (±1.9)
	2800	29.2 (±0.4)	25.3 (±0.5)
	4200	21.3 (±0.5)	17.2 (±0.4)
	5600	17.9 (±0.3)	14.0 (±0.0)
	7000	16.0 (±0.0)	12.1 (±0.3)
25%	1400	43.1 (±0.6)	23.9 (±0.7)
	2800	20.0 (±0.0)	17.5 (±0.5)
	4200	13.5 (±0.5)	11.0 (±0.0)
	5600	11.0 (±0.0)	9.0 (±0.0)
	7000	9.7 (±0.5)	7.0 (±0.0)
50%	1400	25.1 (±0.7)	7.0 (±0.0)
	2800	11.4 (±0.5)	8.7 (±0.5)
	4200	7.0 (±0.0)	5.8 (±0.4)
	5600	5.9 (±0.3)	4.0 (±0.0)
	7000	5.0 (±0.0)	3.0 (±0.0)

Table 6.10. Same as Table 6.9, but using the Tiny Images dataset

% of data in GPU	Q/s	GPU only	CPU-GPU Work Stealing
12.5%	6800	25.0 (±0.0)	17.3 (±0.5)
	13600	16.0 (±0.0)	11.0 (±0.0)
	20400	13.0 (±0.0)	9.0 (±0.0)
	27200	12.0 (±0.0)	9.0 (±0.0)
	34000	11.0 (±0.0)	8.0 (±0.0)
25%	6800	20.5 (±0.5)	14.0 (±0.0)
	13600	11.0 (±0.0)	7.9 (±0.3)
	20400	8.0 (±0.0)	6.0 (±0.0)
	27200	7.0 (±0.0)	5.0 (±0.0)
	34000	7.0 (±0.0)	5.0 (±0.0)
50%	6800	16.1 (±0.3)	2.0 (±0.0)
	13600	8.0 (±0.0)	5.3 (±0.5)
	20400	5.0 (±0.0)	4.0 (±0.0)
	27200	4.0 (±0.0)	3.0 (±0.0)
	34000	4.0 (±0.0)	2.0 (±0.0)

responds to an uncompressed database of about 30 TB. All cases executed 100,000 queries.

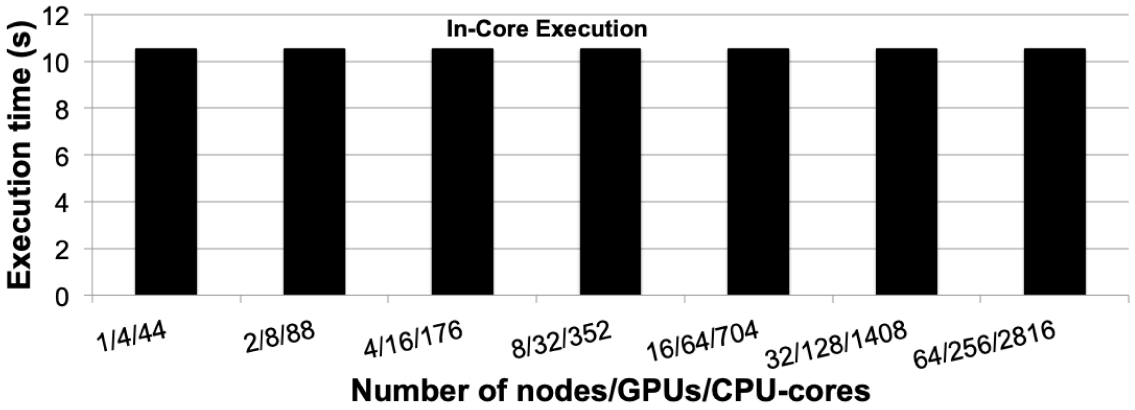

Figure 6.2. In-core weak-scaling execution: a dataset of 500 million SIFT vectors × the number of GPUs is used in each experiment.

Figures 6.2 and 6.3 show execution times of the system as the number of nodes/GPUs/CPU cores are increased with a proportional growth of the dataset used. It is noticeable that the system attained very good scalability in both cases, reaching a parallel efficiency of over 0.99. This is a result of our effective parallelization that is asynchronous and bottleneck free. Further, we have also measured the network utilization to understand the scalability barriers in that respect. In our solution, the network utilization grows linearly with the number of nodes used, and it was only about 7 MB/s and 2 MB/s, respectively, for the int-core and out-of-core executions with 64 nodes using 256 GPUs and 2816 CPU cores cooperatively. This a small demand for execution at this scale.

6.7 Summary

In this chapter, we compared the IVFADC algorithm with FLANN, showing that it had good efficiency while consuming little memory, which is important when dealing with huge datasets. We evaluated our in-core and out-of-core approaches in two datasets: SIFT 500M and Tiny Images, which contains, respectively, 500 million SIFT descriptors and 78 million GIST descriptors. We showed that our DQPP strategy and our CPU-GPU with work-stealing approach produced good results across a variety of query rates for both datasets. The queries arrival was modeled by using a Poisson distribution. Furthermore, we showed that our CPU-GPU with work-stealing approach works well even if we vary the GPU memory capacity. We also showed that our work

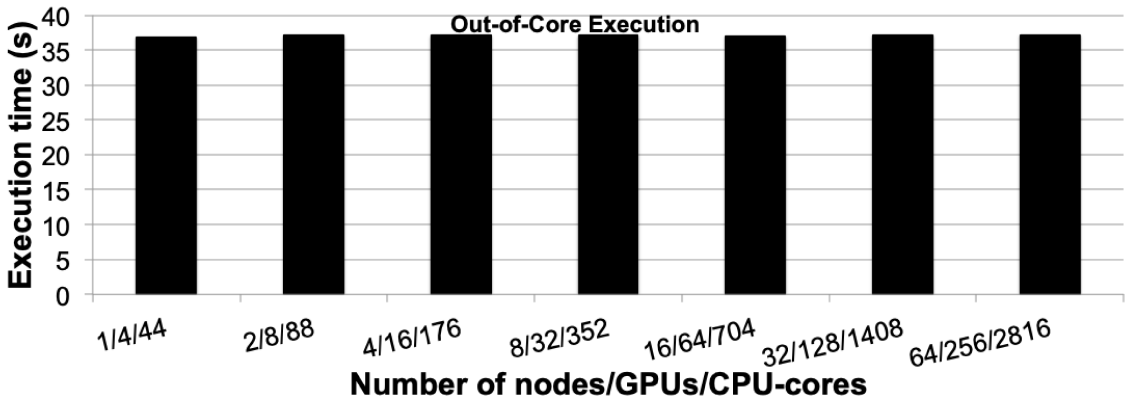

Figure 6.3. Out-of-core weak-scaling execution: a dataset of 1 billion SIFT vectors × the number of GPUs is used in each experiment. Only half of the index can be stored in the GPU memory.

stealing approach was effective in reducing the number of data transfers from CPU to GPU, which played a significant role in reducing the average response times. Last, we evaluated the scalability of our system with up to 256 V100 GPUs and a dataset of up to 30TB of data. The results were promising, showing a very strong parallel efficiency of over 0.99.

In the next chapter, we state our conclusions and consider future directions for this project.

Chapter 7

Conclusions and future directions

This dissertation addresses the problem of indexing very large multimedia datasets with an efficient distributed-memory implementation of the IVFADC for hybrid distributed memory machines equipped with CPU and GPU. Because these datasets are continually growing at an unprecedented rate, we have also developed mechanisms to support out-of-core GPU executions to enable the use of GPUs in the execution when the index can not fit in its memory. Along with the IVFADC ability of describing the data descriptors using small quantization codes, this parallel system can handle very large databases while, at the same time, delivering high throughput and achieving good parallel efficiency.

We have also developed strategies to adapt the system during the execution in order to minimize response times under fluctuation workloads, as observed in online multimedia services. For instance, our DQPP approach that adapts the system at run-time has been able to reduce the average response time in 7× compared to a trivial greedy policy. The improvement in the out-of-core execution with our optimizations was able to reduce the response time in about 1.6× on average with work-stealing. The distributed memory execution in a machine with 256 GPUs and 2816 CPU cores attained a parallel efficiency of about 0.99. At the same time, this distributed memory execution using all 256 GPUs consumed a small network bandwidth (≈7 MB/s), which indicates our system should be able to scale to even larger machines.

In future works, we intend to evaluate our techniques using other ANN algorithms. We believe that our proposed strategies are robust enough to handle most algorithms without needing a lot of changes, but this needs to be evaluated in an experimental setting. Also, it might be useful to study the possibility of using different ANN algorithms in the CPU and GPU. For instance, we might use the HSNW algorithm in the CPU. This might help to reduce the gap between the performance of the

GPU and CPU. However, one aspect that must be considered in this context is that the accuracy of the query results might change depending on which device it was executed, due to the approximate nature of the algorithms. Therefore, it will be necessary to add a theoretical layer to the system to bound the difference between algorithms and devices to an acceptable level. Otherwise, the user may experience query results that may significantly diverge depending on how the system executes them.

While in this work we explored the use of CPUs and GPUs, these are not the only devices available. Recently, Zhang et al. (2018) proposed an FPGA (Field Programmable Gate Array) based implementation of the product quantization approximated nearest neighbor search. According to them, it significantly outperforms state-of-the-art methods on CPU and GPU. Studying how our approach would map to FPGAs might be a worthwhile endeavor, since their innate characteristics are very different from CPUs and GPUs.

We might even take this further, and create a general framework that can automatically decide what would be the best configuration and algorithms, given the devices in the system, the desired accuracy etc.

While the GPU implementation of the IVFADC algorithm in Faiss is very well optimized, it focuses on achieving maximum performance in batch scenarios, with a large number of queries to be processed at once. According to Johnson et al. (2019), they use one warp of the GPU per query. Clearly, when the number of queries is small, this will not use the full processing power of the GPU. So, one possible future work would be to adapt Faiss's algorithm to better use the available GPU resources when the number of queries is small.

One of the most expensive steps in the out-of-core execution is the constant transfers of the database to the GPU. Studying how to reduce this cost, maybe with the use of new data structures, might increase significantly the performance of the out-of-core algorithm.

Our experiments ran in a very fast network. What if the network was slower? What if the current traffic in the network was a significant factor? Exploring this scenario might produce worthwhile results.

Last, one interesting project would be to, when the workload is low, increase the accuracy of the searches, instead of simply becoming idle. Conversely, if the workload becomes too high, maybe decreasing the accuracy of the searches might be a viable alternative.

Bibliography

Andrade, G., Fernandes, A., Gomes, J. M., Ferreira, R., and Teodoro, G. (2019). Large-scale parallel similarity search with product quantization for online multimedia services. *Journal of Parallel and Distributed Computing*, 125:81 – 92. ISSN 0743-7315.

Avila, S., Thome, N., Cord, M., Valle, E., and Araújo, A. d. A. (2011). Bossa: Extended bow formalism for image classification. In *2011 18th IEEE International Conference on Image Processing*, pages 2909--2912. IEEE.

Avila, S., Thome, N., Cord, M., Valle, E., and AraúJo, A. D. A. (2013). Pooling in image representation: The visual codeword point of view. *Computer Vision and Image Understanding*, 117(5):453--465.

Babenko, A. and Lempitsky, V. (2015). The Inverted Multi-Index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(6):1247–1260. ISSN 0162-8828.

Babenko, A., Slesarev, A., Chigorin, A., and Lempitsky, V. (2014). Neural codes for image retrieval. In *European conference on computer vision*, pages 584--599. Springer.

Bahmani, B., Goel, A., and Shinde, R. (2012). Efficient Distributed Locality Sensitive Hashing. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 2174--2178, New York, NY, USA. ACM.

Batcher, K. E. (1968). Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307--314.

Beis, J. S. and Lowe, D. G. (1997). Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *cvpr*, volume 97, page 1000. Citeseer.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509--517.

Böhm, C., Berchtold, S., and Keim, D. (2001). Searching in high-dimensional spaces - index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373.

Cayton, L. (2012). Accelerating nearest neighbor search on manycore systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 402–413. ISSN 1530-2075.

Datta, R., Joshi, D., Li, J., and Wang, J. Z. (2008). Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys (Csur)*, 40(2):1--60.

Dolatshah, M., Hadian, A., and Minaei-Bidgoli, B. (2015). Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. *arXiv preprint arXiv:1511.00628*.

Douze, M., Jégou, H., and Perronnin, F. (2016). Polysemous codes. In Leibe, B., Matas, J., Sebe, N., and Welling, M., editors, *Computer Vision – ECCV 2016*, pages 785--801, Cham. Springer International Publishing.

Douze, M., Jégou, H., Sandhawalia, H., Amsaleg, L., and Schmid, C. (2009). Evaluation of GIST Descriptors for Web-scale Image Search. In *Proceedings of the ACM International Conference on Image and Video Retrieval*, CIVR '09, pages 19:1--19:8, New York, NY, USA. ACM.

Forum, M. P. (1994). Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA.

Friedman, J., Bentley, J., and Finkel, R. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM TOMS*, 3:209–226.

Ge, T., He, K., Ke, Q., and Sun, J. (2013). Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946--2953.

Gong, Y., Wang, L., Guo, R., and Lazebnik, S. (2014). Multi-scale orderless pooling of deep convolutional activation features. In Fleet, D., Pajdla, T., Schiele, B., and Tuytelaars, T., editors, *Computer Vision – ECCV 2014*, pages 392--407, Cham. Springer International Publishing.

Indyk, P. and Motwani, R. (1998a). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604--613. ACM.

Indyk, P. and Motwani, R. (1998b). Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604--613, New York, NY, USA. ACM.

Jain, M. (2014). *Enhanced image and video representation for visual recognition*. PhD thesis.

Jegou, H., Douze, M., and Schmid, C. (2011). Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128. ISSN 0162-8828.

Jégou, H., Douze, M., Schmid, C., and Pérez, P. (2010). Aggregating local descriptors into a compact image representation. In *CVPR 2010-23rd IEEE Conference on Computer Vision & Pattern Recognition*, pages 3304--3311. IEEE Computer Society.

Jégou, H., Tavenard, R., Douze, M., and Amsaleg, L. (2011). Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861--864. IEEE.

Johnson, J., Douze, M., and Jégou, H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, pages 1–1. ISSN 2332-7790.

Kalantidis, Y. and Avrithis, Y. (2014). Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2329–2336. ISSN 1063-6919.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097--1105.

Kruliš, M., Skopal, T., Lokoč, J., and Beecks, C. (2012). Combining CPU and GPU architectures for fast similarity search. *Distributed and Parallel Databases*, 30(3):179--207. ISSN 1573-7578.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396--404.

Li, W., Zhang, Y., Sun, Y., Wang, W., Li, M., Zhang, W., and Lin, X. (2019). Approximate nearest neighbor search on high dimensional data-experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*.

Liu, T., Moore, A. W., and Gray, A. (2006). New algorithms for efficient high-dimensional nonparametric classification. *Journal of Machine Learning Research*, 7(Jun):1135--1158.

Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150--1157. Ieee.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91--110.

Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K. (2007). Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 950--961. VLDB Endowment.

Malkov, Y. A. and Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*.

Menasce, D. A. and Almeida, V. (2001). *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition. ISBN 0130659037.

Moise, D., Shestakov, D., Gudmundsson, G., and Amsaleg, L. (2013). Indexing and Searching 100M Images with Map-reduce. In *Proceedings of the 3rd ACM Conference on International Conference on Multimedia Retrieval*, ICMR '13, pages 17--24, New York, NY, USA. ACM.

Muja, M. and Lowe, D. (2009). Fast approximate nearest neighbors with automatic algorithm configuration. volume 1, pages 331–340.

Muja, M. and Lowe, D. G. (2014). Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240. ISSN 0162-8828.

Nister, D. and Stewenius, H. (2006). Scalable recognition with a vocabulary tree. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2161--2168. Ieee.

Oliva, A. and Torralba, A. (2001). Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145--175.

Omohundro, S. M. (1989). *Five balltree construction algorithms*. International Computer Science Institute Berkeley.

Pan, J. and Manocha, D. (2011). Fast GPU-based Locality Sensitive Hashing for K-nearest Neighbor Computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 211--220, New York, NY, USA. ACM.

Revaud, J., Douze, M., Schmid, C., and Jégou, H. (2013). Event retrieval in large video collections with circulant temporal encoding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2459--2466.

Silpa-Anan, C. and Hartley, R. (2008). Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1--8. IEEE.

Sivic, J. and Zisserman, A. (2003). Video google: A text retrieval approach to object matching in videos. In *null*, page 1470. IEEE.

Stupar, A., Michel, S., and Schenkel, R. (2010). RankReduce - processing K-Nearest Neighbor queries on top of MapReduce. In *In LSDS-IR*.

Teodoro, G., Fireman, D., Guedes, D., Jr., W. M., and Ferreira, R. (2008). Achieving Multi-Level Parallelism in the Filter-Labeled Stream Programming Model. In *2008 37th International Conference on Parallel Processing*, pages 287–294. ISSN 0190-3918.

Teodoro, G., Kurç, T. M., Kong, J., Cooper, L. A. D., and Saltz, J. H. (2014a). Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1063--1072.

Teodoro, G., Valle, E., Mariano, N., Torres, R., Meira, W., and Saltz, J. H. (2014b). Approximate similarity search for online multimedia services on distributed cpu–gpu platforms. *The VLDB Journal*, 23(3):427--448. ISSN 0949-877X.

Torralba, A., Fergus, R., and Freeman, W. T. (2008). 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 30(11):1958--1970.

Uhlmann, J. K. (1991). Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4):175 – 179. ISSN 0020-0190.

Wan, J., Wang, D., Hoi, S. C. H., Wu, P., Zhu, J., Zhang, Y., and Li, J. (2014). Deep Learning for Content-Based Image Retrieval: A Comprehensive Study. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 157--166, New York, NY, USA. ACM.

Weber, R., Schek, H.-J., and Blott, S. (1998). A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194--205, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Wieschollek, P., Wang, O., Sorkine-Hornung, A., and Lensch, H. (2016). Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2027--2035.

Zhang, J., Khoram, S., and Li, J. (2018). Efficient large-scale approximate nearest neighbor search on opencl fpga. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4924--4932.