RobustiQ: A Robust ANN Search Method for Billion-scale Similarity Search on GPUs

Wei Chen
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
zjuchenwei@hust.edu.cn

Jincai Chen*
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
jcchen@hust.edu.cn

Fuhao Zou*
School of Computer Science and Technology, Huazhong University of Science and Technology
Wuhan, China
fuhao_zou@hust.edu.cn

Yuan-Fang Li
Faculty of Information Technology, Monash University
Clayton, Australia
yuanfang.li@monash.edu

Ping Lu
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
luping06@hust.edu.cn

Wei Zhao
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
m201873259@hust.edu.cn

ABSTRACT

GPU-based methods represent state-of-the-art in approximate nearest neighbor (ANN) search, as they are scalable (billion-scale), accurate (high recall) as well as efficient (sub-millisecond query speed). Faiss, the representative GPU-based ANN system, achieves considerably faster query speed than the representative CPU-based systems. The query accuracy of Faiss critically depends on the number of indexing regions, which in turn is dependent on the amount of available memory. At the same time, query speed deteriorates dramatically with the increase in the number of partition regions. Thus, it can be observed that Faiss suffers from a lack of *robustness*, that the fine-grained partitioning of datasets is achieved at the expense of search speed, and vice versa. In this paper, we introduce a new GPU-based ANN search method, Robust Quantization (RobustiQ), that addresses the robustness limitations of existing GPU-based methods in a holistic way. We design a novel hierarchical indexing structure using vector and bilayer line quantization. This indexing structure, together with our indexing and encoding methods, allows RobustiQ to avoid the need for maintaining a large lookup table, hence reduces not only memory consumption but also query complexity. Our extensive evaluation on two public billion-scale benchmark datasets, SIFT1B and DEEP1B, shows that RobustiQ consistently obtains 2–3× speedup over Faiss while achieving better query accuracy for different codebook sizes. Compared to the best CPU-based ANN systems, RobustiQ achieves even more pronounced average speedups of 51.8× and 11× respectively.

*Corresponding author

CCS CONCEPTS

• Information systems → Top-k retrieval in databases; **• Computing methodologies → Visual content-based indexing and retrieval**;

KEYWORDS

GPU; billion-scale; high-dimensional; ANN search; inverted index; quantization; RobustiQ

ACM Reference Format:
Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, and Wei Zhao. 2019. RobustiQ: A Robust ANN Search Method for Billion-scale Similarity Search on GPUs. In *International Conference on Multimedia Retrieval (ICMR '19), June 10–13, 2019, Ottawa, ON, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3323873.3325018

1 INTRODUCTION

Approximate nearest neighbor (ANN) search of high-dimensional data [13] is an important task in modern computer vision and deep learning research. Quantization-based methods for ANN search have recently attracted significant attention. The current breed of high-performing billion-scale quantization-based ANN systems [1, 3, 4, 9, 15, 16] typically comprises three main components: (1) an *indexing structure* that partitions the dataset space into a large number of disjoint regions, (2) an *encoding* method that compresses dataset points into a memory-efficient representation, and (3) a search and query algorithm that, given a query point, computes the distance between points only in the regions closest to the query point.

GPUs provide massive parallelism and GPU-based methods represent state-of-the-art in quantization ANN search. It has been shown that GPU-based systems are far superior to CPU-based systems in terms of efficiency while maintaining comparable accuracy [9, 15]. To the best of our knowledge, there are two GPU-based ANN systems that are capable of handling billion-scale datasets: PQT [15] and Faiss [9]. PQT proposes a novel quantization method call line quantization (LQ) and is the first billion-scale similarity retrieval system on the GPU. Subsequently Faiss implements an indexing structure based on vector quantization (VQ) [11], and currently has the state-of-the-art performance on GPUs in both search accuracy and speed.

We observe that the VQ-based system Faiss exhibit a lack of *robustness*. There is an inherent trade-off between search accuracy and search speed. An increase in one is typically achieved at the expense of the other, and vice versa. The lack of robustness of Faiss is mainly due to the inherent conflict between its indexing structure and distance computation algorithm. To achieve high accuracy, the VQ-based indexing structure needs to store a large number of full dimensional centroids, which is memory-consuming. At the same time, Faiss' distance computation algorithm depends on a precomputed, large lookup table, whose memory usage also increases with the number of centroids. Given the limited amount of memory on GPUs, Faiss needs to reduce either the number of centroids or the size of the lookup table, resulting in reduced search accuracy. Besides, the large lookup table needs to be frequently visited during the query process, which reduces query speed.

In this paper, we present RobustiQ, a novel billion-scale ANN similarity search framework. RobustiQ includes a three-level hierarchical inverted indexing structure based on Vector and Bilayer Line Quantization (VBLQ), which can be implemented on GPU efficiently and an efficient approximate distance computation method. The main contributions of our solution are threefold.

(1) We demonstrate how to increase the number of regions with memory efficiency by the novel inverted index structure. The efficient indexing contributes to high accuracy for approximate search.
(2) We propose a novel approximate distance algorithm that do not depend on a memory-consuming lookup tables like Faiss.
(3) A range of experiments shows that our system consistently and significantly outperforms state-of-the-art GPU- and CPU-based retrieval systems on both recall and efficiency on two public billion-scale benchmark datasets with single- and multi-GPU configurations.

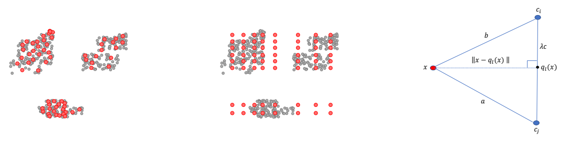
The rest of the paper is organized as follows. Section 2 introduces related works on indexing with quantization methods. Section 3 presents RobustiQ system, our approach for approximate nearest neighbor (ANN)-based similarity search method and the details of GPU implementation. Section 4 provides a series of experiments, and compares the results to the state of the art.

2 RELATED WORK

In this section, we briefly introduce some quantization methods and several retrieval systems related to our approach. For example, we assume that $\mathcal{X} = \{x_1, \ldots, x_N\} \subset \mathbb{R}^D$ is a finite set of N data points of dimension D.

2.1 Vector quantization (VQ) and Product quantization (PQ)

In vector quantization [11] (Figure 1 a), a quantizer is a function q_v that maps a D-dimensional vector x to a vector $q_v(x) \in C$, where C is a finite subset of $\mathbb{R}^D$, of k vectors. Each vector $c \in C$ is called a centroid, and C is a codebook of size k. We can use Lloyd iterations [2] to efficiently obtain a codebook C on a subset



(a) Vector Quantization. (b) Product Quantization. (c) Line Quantization.

Figure 1: Three different quantization methods. Vector and Product quantization methods are both with $k = 64$ clusters. The red dots in plot a and b denote the centroids and the grey dots denote the dataset points in both plots. Vector quantization (a) maps the dataset points to the closest centroids. Product quantization (b) performs clustering in each subspace independently (here axes). In plot (c), a 2-dimensional point x (red dot) is projected on line $l(c_i, c_j)$ with the anchor point $q_l(x)$ (black dot). The a, b, c denote the values of $\|x - c_i\|^2$,$\|x - c_j\|^2$ and $\|c_i - c_j\|^2$ respectively. We use the parameter λ to represent the value of $\|c_i - q_l(x)\|/c$. The anchor point $q_l(x)$ can be represented by c_i, c_j and λ. The distance from x to $l(c_i, c_j)$ can be calculated by a, b, c and λ.

of the dataset. According to Lloyd's first condition, to minimize quantization error a quantizer q_v should map vector x to its nearest codebook centroid. Hence, the set of points $\mathcal{X}_i = \{x \in \mathbb{R}^D \mid q_v(x) = c_i\}$ is called a cluster or a region for centroid c_i.

Product quantization [5] (Figure 1 b) is an extension of vector quantization. PQ can generate K^m centroids with m codebooks of K sub-centroids each. The benefit of PQ is that it can easily generate a much larger number of centroids than VQ with moderate values of m and K. So PQ can be used to compress datasets. Typically each sub-codebook of PQ contains 256 sub-centroids and each vector x is mapped to a concatenation of m sub-centroids $(c^1_{j_1}, \cdots, c^m_{j_m})$, for j_i is a value between 1 and 256. Hence the vector x can be encoded into an m-byte code of sub-centroid index $(j_1, \cdots, j_m)$. With the approximate representation by PQ, the Euclidean distances between the query vector and the large number of compressed vectors can be computed efficiently. According to the ADC procedure [7], the computation is performed based on lookup tables.

$$\|y - x\|^2 \approx \|y - q_p(x)\|^2 = \sum_{i=1}^{m} \|y^i - c^i_{j_i}\|^2 \tag{1}$$

where y^i is the ith subvector of a query y. The Euclidean distances between the query sub-vector y^i and each sub-centroids $c^i_{j_i}$ can be precomputed and stored in lookup tables that reduce the complexity of distance computation from $O(D)$ to $O(m)$. Due to the high compression quality and efficient distance computation approach, PQ is considered the top choice for compact representation of large-scale datasets[3, 4, 6, 10, 12].

2.2 Line quantization (LQ)

Line quantization (LQ) [15] quantizes a data point to a closest project point $q_l(x)$ on a line. We also call the project point as an *anchor point*. LQ owns a codebook C of K centroids like VQ. As shown in Figure 1 (c), with any two different centroids $c_i, c_j \in C$, a line is formed and denoted by $l(c_i, c_j)$, and the anchor point is denoted by a_{ij}. A line quantizer q_l quantizes a point x to the closest anchor point on a line as follows:

$$q_l(x) = \arg\min_{a_{ij}} d(x, a_{ij}), \tag{2}$$

where $d(x, a_{ij})$ is the Euclidean distance from x to anchor point a_{ij} on the line $l(c_i, c_j)$, and the set $\mathcal{X}_{i,j} = \{x \in \mathbb{R}^D | q_l(x) = a_{ij}\}$ is called a cluster or a region for line $l(c_i, c_j)$. The squared distance $d(x, a_{ij})^2$ can be calculated as follows:

$$\begin{aligned} d(x, a_{ij})^2 = (1-\lambda)\|x - c_i\|^2 + (\lambda^2 - \lambda)\|c_j - c_i\|^2 \\ + \lambda\|x - c_j\|^2 \end{aligned} \tag{3}$$

Because the values of $\|x - c_j\|^2$, $\|x - c_i\|^2$, $\|c_j - c_i\|^2$ can be precomputed between x and all centroids, Equation 3 can be calculated efficiently. The anchor point a_{ij} of x is computed in the following way:

$$a_{ij} = (1-\lambda) \cdot c_i + \lambda \cdot c_j, \tag{4}$$

where λ is a scalar parameter that can be computed as follows:

$$\lambda = 0.5 \cdot \frac{(\|x - c_i\|^2 + \|c_j - c_i\|^2 - \|x - c_j\|^2)}{\|c_j - c_i\|^2}. \tag{5}$$

When x is quantized to a region of $l(c_i, c_j)$, then the displacement of x from a_{ij} can be computed as follows:

$$r_{q_l}(x) = x - a_{ij}. \tag{6}$$

For a codebook with K centroids, LQ can form $K \cdot (K-1)$ lines, or $K \cdot (K-1)$ regions. Thus, the benefit of an LQ-based indexing structure is that it can produce many more regions than that of VQ-based regions. However it is considerably more complicated to find the nearest line for a point when K is large. So we use LQ as an indexing approach with a codebook of a few lines.

Faiss [9] is a very efficient GPU-based retrieval approach, by realizing the idea of IVFADC [7] on GPUs. Faiss uses the inverted index based on VQ [14] for non-exhaustive search and compresses the dataset by PQ. The inverted index of IVFADC owns a vector quantizer q with a codebook of K centroids. Thus there are K regions for the data space. Each point $x \in \mathcal{X}$ is quantized to a region corresponding to a centroid by a VQ quantizer q_v. The displacement of each point from the centroid of a region it belongs to is defined as

$$r_q(x) = x - q(x), \tag{7}$$

where the displacement $r_q(x)$ is encoded by a PQ quantizer q_p with m codebooks shared by all regions. For each region, an inverted list of data points is maintained, along with PQ-encoded displacements.

When the candidate list $\mathcal{L}_c$ for query is collected, the squared distance between y and data point $x \in \mathcal{L}_c$ can be approximated by following equation:

$$\|y - x\|^2 \approx \|y - q(x)\|^2 = \|y - q_v(x) - q_p(x - q_v(x))\|^2. \tag{8}$$

Moreover, the $\|y - q(x)\|^2$ in Equation 8 can be further decomposed to following expression:

$$\underbrace{\|q_p(\cdots)\|^2 + 2\langle q_v(x), q_p(\cdots)\rangle}_{\text{term1}} + \underbrace{\|y - q_v(\cdots)\|^2}_{\text{term2}} - \underbrace{2\langle y, q_p(\cdots)\rangle}_{\text{term3}}, \tag{9}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product between two points. The term1 in Expression 8 is independent to of the query. It can be precomputed from the quantizers, and stored in a lookup table $\mathcal{T}$ of size $K \times m \times 256$. However the storage of Table $\mathcal{T}$ is memory consuming, when K and m is large. For example, when $K = 2^{18}$ and $m = 16$, it requires more than 2GB memories for the lookup table. The large codebook and lookup table will slow down query speed by a large margine. We will discuss that in Sec.4.1.

Ivf-hnsw [4] proposed on a two-level inverted index structure, which can partition the data space better than that of Faiss with a subtle memory overhead. Ivf-hnsw can splits the data space into $K \cdot n$ regions with a vector quantizer q with a codebook of K centroids and an n-nearest neighbor (n-NN) graph. The n-NN graph is a directed graph in which nodes are first-level centroids and edges connect a centroid to its n nearest neighbors. Then the second-level centroids are built on the edges of the n-NN graph, and the second-level regions are splited by the sub-censtroids. Ivf-hnsw has a better query accuracy than Faiss, however, its query speed is much slower than that of Faiss, because it is only implemented on CPU.

Inspired by the above two systems, We propose an efficient indexing structure that is improved based on that of Ivf-hnsw and a better GPU implementation than that of Faiss. The two factors make our system has a better query accuracy and speed than Faiss and Ivf-hnsw.

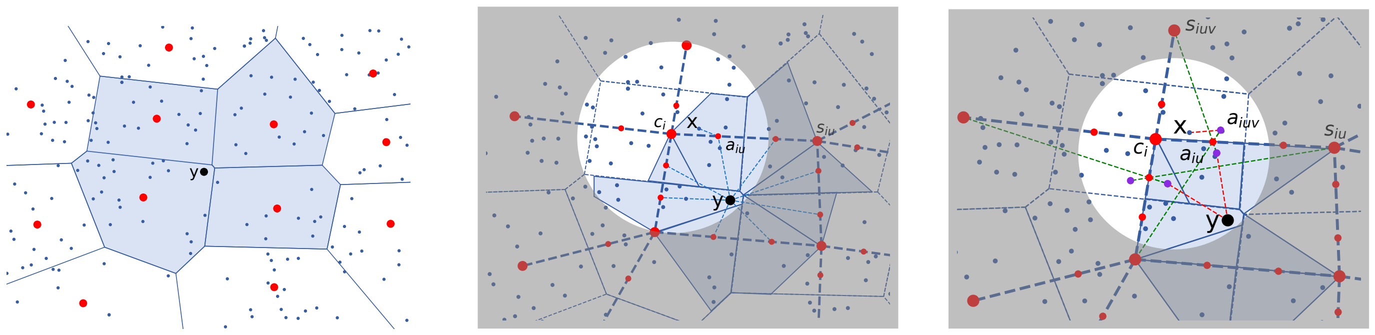
3 THE ROBUSTIQ SYSTEM

In this section we introduce our GPU-based ANN retrieval system, RobustiQ, that contains a three-layer hierarchical indexing structure based on vector and bilayer line quantization (VBLQ) and an asymmetric distance computation method (Sec. 3.1). The indexing and encoding process will be presented in Sec. 3.2, and the querying process is discussed in Sec. 3.3.

Comparing with the state-of-the-art GPU-based system Faiss, RobustiQ has two main advantages. (1) Our VBLQ indexing structure can generate shorter and more accurate candidate lists for the query point, which will considerably accelerate query speed. (2) RobustiQ incorporates a novel encoding method that removes the need for maintaining a large lookup table that is required by Faiss. This reduction significantly reduces the complexity in distance computation and hence contributes to the improvement in query speed. In the remainder of this section we use Figure 2 to illustrate our framework.

3.1 The VBLQ-based indexing structure

For billion-scale datasets with a moderate number of regions (e.g. 2^{17}) produced by vector quantization (VQ), the number of data points in most regions is too large, which negatively affects search accuracy (high quantization error) and query speed (a large fraction to traverse). To alleviate this problem, we propose a hierarchical indexing structure, the vector and bilayer line quantization (VBLQ) method. In VBLQ, VQ is used as a global coarse quantizer to partition the dataset space into several global regions, such as the four shaded regions in Figure 2 (left). Each region is then locally quantized into several sub-regions by a two-layer line quantizer. Therefore, the data points in each region can be further distributed into the sub-regions to avoid overcrowding in regions and also



First-level indexing structure (VQ) **Second-level indexing structure (LQ)** **Third-level indexing structure (LQ)**

Figure 2: An illustration of our VBLQ-based indexing structure on data points (small blue dots) of dimension 2 ($D = 2$). Each level of our indexing structure is described in a separate plot. The large red dots denote the first-level centroids, the small red dots denote the second-level anchor points and the purple dots denote the third-level anchor points. The data point x (small blue dots) is quantized to the closest anchor point in the middle and right plots. The middle and right plots are zoomed in and masked to highlight the part relevant to the discussion. Left: The 4 shaded areas represent the first-level regions, which are produced by VQ, one for each first-level centroid. They represent the fraction of the dataset that needs to be traversed for the query point y by a VQ-based system such as Faiss. Middle: For the top-left centroid in the middle plot, $n_1 = 4$ closest top-level centroids are found. In other words the n_1-NN graph consists of all the centroids and the first-level edges (thick dashed lines) connecting each centroid to the n_1 closest centroids. Each first-level region in the left figure is split into 4 second-level regions, each of which encloses the data points closest to the second-layer anchor point a_{iu} (small red dots) on the corresponding edge. During query, only 1/2 of the closest second-level regions are retained for y. Right: We connect each second-layer anchor point to another $n_2 = 2$ nodes s_{iuv} which connect the same centroid c_i and regard the lines between the anchor points and nodes as the second-level edges (green dashed lines). Each second-level region is further split into 2 third-level regions, each of which represents the data points closest to the third-layer anchor point a_{iuv} (small purple dots) on the corresponding second-level edge. Again only 1/2 of the closest third-level subregions (shaded in blue) need to be traversed for a query. As can be seen, VBLQ allows search to process a substantially smaller fraction (1/4) of the dataset than a VQ-based approach such as Faiss.

reduce quantization error. Thus the proposed indexing structure can substantially increase the search accuracy (lower quantization error) and query speed (smaller fraction to traverse).

Our indexing structure is a three-layer hierarchical structure which consists of two kinds of quantizers. The first level contains a vector quantizer q_v with a codebook of K centroids. The vector quantizer q_v partitions the data point space $\mathcal{X}$ into K regions. Each first-level region is further split into n_1 second-level regions by a first-layer line quantizer q_l^1 with an n_1-nearest neighbor (n_1-NN) graph. Finally, each second-level region contains n_2 third-level regions by a second-layer line quantizer q_l^2 with an n_2-edge graph.

Illustrated in Figure 2 (middle), the line quantizer q_l^1 is responsible for quantizing the data points in each first-level region to the closest anchor points (small red dots) of first-layer edges (thick blue dashed lines) in the region. In Figure 2 (right), within each second-level region, we connect each anchor point (e.g. a_{iu}) on a first-layer edge to another n_2 ($\leq n_1 - 1$) nodes (e.g. s_{iuv}) which connect to the same centroid (e.g. c_i) to form the n_2 second-layer edges (green dashed lines). Then the data points in each second-level region is further quantized to n_2 anchor points (small purple dots) on the corresponding second-layer edges by q_l^2. Thus each second-level region is divided into n_2 third-level regions. Overall there are $K \cdot n_1 \cdot n_2$ third-level regions in total.

Memory overhead of indexing structure. One advantage of our indexing structure is its ability to produce substantially more subregions with little additional memory consumption. Same as VQ, our first layer codebook needs $K{\cdot}D{\cdot}sizeof(\texttt{float})$ bytes. In addition, for second-level indexing, for each of the K first-layer centroids, the n_1-NN graph only needs to store (1) the indices of its n_1 nearest neighbors, (2) the distances to its n_1 nearest neighbors, and (3) the distances from the first-layer anchor points to its n_2 nodes, which amounts to $K \cdot n_1 \cdot (sizeof(\texttt{int}) + (n_2 + 1) \cdot sizeof(\texttt{float}))$ bytes. Because We choose the n_2 nodes in an fixed order, we don't need to store the indices of second-layer nodes. Moreover, we need two sets of scalar parameters $\{\lambda_{11}, \cdots, \lambda_{1K}\}$, $\{\lambda_{21}, \cdots, \lambda_{2K}\}$ to compute the two-layer anchor points . That will comsume another $K \cdot n_1 \cdot 2 \cdot sizeof(\texttt{float})$ bytes. Note we do not need to store the full-dimensional points. For a typical values of $K = 2^{17}$ centroids, $n_1 = 32$ and $n_2 = 4$, the additional memory overhead for storing the graph is $2^{17} \cdot 32 \cdot 8 \cdot 4$ bytes (134 MB), which is acceptable for billion-scale datasets.

Instead of utilizing line quantization, an alternative way to produce the subregions is by utilizing vector quantization (VQ) hierarchically in each region. However, that would require storing full-dimensional subcentroids and thus consume too much additional memory. For the same configuration as above ($K = 2^{17}$ centroids and $n = 32 \cdot 4$ third-level subcentroids) and a dimension of $D = 128$, the additional memory overhead for a VQ-based three-layer hierarchical indexing structure would be $2^{17} \cdot 32 \cdot 4 \cdot 128 \cdot sizeof(\texttt{float})$

additional bytes (4,096MB). As can be seen, our VBLQ-based hierarchical indexing structure is substantially more compact, consuming only 3.3% of the memory required by a VQ-based hierarchical approach.

We note that the PQ-based indexing structure requires $O(K \cdot (D + K))$ memory to maintain the indexing structure, which is memory inefficient as it is quadratic in K. This is a limitation of PQ-based indexing structure. In contrast, the space complexity of our hierarchical indexing structure is $O(K \cdot (D + n_1 \cdot n_2))$, where typically $n_1 \cdot n_2 \ll K$ ($n_1 \cdot n_2$ is much smaller than k), hence making our index much more memory efficient.

3.2 Indexing and encoding

In this subsection we describe the indexing and encoding processes. For our three-level VBLQ indexing structure, the indexing process comprises two different quantization procedures, VQ for the first layer and LQ for the second and third layers.

3.2.1 First-level indexing. Similar to the IVFADC scheme [7], each dataset point is quantized by the vector quantizer q_v to the first-level regions surrounded by the solid lines in Figure 2 (left).

3.2.2 Second-level indexing. Let $\mathcal{X}^i$ be a region of $\{x_1, \ldots, x_{N_i}\}$ that corresponds to a centroid c_i, for $1 \le i \le K$. In constructing the n_1-NN graph, let $S_i = \{s_{i1}, \ldots, s_{in_1}\}$ denote the set of the n_1 centroids closest to c_i and $l(c_i, s_{iu})$ denote the edge between c_i and s_{iu}, for $1 \le u \le n_1$. Using the scalar parameter λ_{1i}, The n_1 anchor points (small red dots in Figure 2) can be computed for the n_1 edges using Equation 4.

$$a_{iu} = (1 - \lambda_{1i}) \cdot c_i + \lambda_{1i} \cdot s_{iu}. \tag{10}$$

The points in $\mathcal{X}^i$ are then quantized to the n_1 second-level regions by a line quantizer q_l^1 with the codebook $\mathcal{A}_i$ of n_1 anchor points $\{a_{i1}, \ldots, a_{in_1}\}$. Thus the region $\mathcal{X}^i$ is split into n_1 subregions $\{\mathcal{X}^i_1, \ldots, \mathcal{X}^i_{n_1}\}$ and each point $x \in \mathcal{X}^i$ is quantized to a second-level subregion $\mathcal{X}^i_u$, which corresponds to the closest *anchor point* a_{iu} on the edge $l(c_i, s_{iu})$.

3.2.3 Third-level indexing. Let $S_{iu} = \{s_{iu1}, \ldots, s_{iun_2}\}$ denote the set of the n_2 nodes sampled from S_i, where $s_{iuv}, s_{iu} \in S_i$ and $s_{iuv} \neq s_{iu}$. We connect the anchor point a_{iu} to all the nodes in the S_{iu} to form a n_2-edge graph, and the $l(a_{iu}, s_{iuv})$ denotes a second-layer edge between a_{iu} and s_{iuv}, for $1 \le v \le n_2$. Using the scalar parameter λ_{2i}, The n_2 anchor points (small purple dots in Figure 2) can be computed for the n_2 edges using Equation 4.

$$a_{iuv} = (1 - \lambda_{2i}) \cdot a_{iu} + \lambda_{2i} \cdot s_{iuv}. \tag{11}$$

Thus, each second-level region $\mathcal{X}^i_u$ is further split into n_2 third-level regions $\{\mathcal{X}^i_{u1}, \ldots, \mathcal{X}^i_{un_2}\}$. The points in $\mathcal{X}^i_u$ are quantized to the n_2 third-level regions by a line quantizer q_l^2 with a with a codebook $\mathcal{A}_{iu}$ of n_2 anchor points $\{a_{iu1}, \ldots, a_{iun_2}\}$. Each point $x \in \mathcal{X}^i_u$ is assigned to a third-level subregion $\mathcal{X}^i_{uv}$, which corresponds to the closest *anchor point* a_{iuv} (small purple dots in Figure 2 (right)) on the edge $l(a_{iu}, s_{iuv})$). So the entire space $\mathcal{X}$ are divided into $K \times n_1 \times n_2$ third-level regions. As can be seen in the figure, comparing to first-level centroids (large red dots), the second-level anchor points are closer to the data points and query point, resulting in smaller quantization errors in VBLQ than in VQ. Therefore, RobustiQ is able to improve search accuracy while reducing search space.

Sampling the second-level nodes. We use a uniform-interval sampling method to choose the second-level nodes. We assume that the $n_1 = 32$ and $n_2 = 4$, so the interval equals to $32/4 = 8$. So we choose s_{i1}, s_{i9}, s_{i17} and s_{i25} from the first-level nodes set S_i to form a subset S_iu for the anchor point a_{iu}. If u is in the numbers of $\{1, 9, 17, 25\}$, i.e. s_{iu} is sampled to the subset S_iu, we use $s_{i(u+1)}$ to replace the node s_{iu}. Although the uniform-interval sampling method is not the optimal way to choose the second-level nodes, it is the most straightforward way to construct an n_2-edge graph for the second-level anchor points and proved to be efficient and effective.

Learning λ_{1i} and λ_{2i}.According to Line quantization, each data point x has its own anchor point when quantized to a edge, and each anchor point corresponds to a scalar parameter λ, however that will increase the complexity of the distance computation for our system. In order to simplify the computation we unified all the anchor points on the same edge to a single anchor point by a uniformed parameter λ. So we use two parameters λ_{1i} and λ_{2i} for the line quantizers q_l^1 and q_l^2 respectively within each first-level region $\mathcal{X}^i$. The value of λ_{1i} and λ_{2i} in Equation 10 and 11 are learnt on the training set. Given a first-level region $\mathcal{X}^i$ corresponding to first-level centroid c_i, let $\mathcal{X}^i$ contain N_i data points. Each data point $x_j \in \mathcal{X}^i$ is quantized to a nearest first-level edge $l(c_i, s_{iu})$ by line quantizer q_l^1.

The scalar parameter λ_{1i} is used to compute the anchor point a_{iu}, and its value is computed by averaging the projections of all data points in the second-level subregions, by computing λ_{1j}.

$$\lambda_{1i} = \frac{1}{N_i} \sum_{j=1}^{N_i} \lambda_{1j}. \tag{12}$$

According to Equation 5, the scalar parameters λ_{1j} can be computed as follows:

$$\lambda_{1j} = 0.5 \cdot \frac{(\|x_j - c_i\|^2 + \|s_{iu} - c_i\|^2 - \|x_j - s_{iu}\|^2)}{\|s_{iu} - c_i\|^2} \tag{13}$$

When λ_{1i} has been learnt, the anchor point a_{iu} is determined for the edge $l(c_i, s_{iu})$. x_j is then quantized to a nearest second-level edge $l(a_{iu}, s_{iuv})$ by line quantizer q_l^2. Similarly, the scalar parameter λ_{2j} is computed by the following equation:

$$\lambda_{2j} = 0.5 \cdot \frac{(\|x_j - a_{iu}\|^2 + \|s_{iuv} - a_{iu}\|^2 - \|x_j - s_{iuv}\|^2)}{\|s_{iuv} - a_{iu}\|^2}. \tag{14}$$

And the value of scalar parameters λ_{2i} is computed as follows:

$$\lambda_{2i} = \frac{1}{N_i} \sum_{j=1}^{N_i} \lambda_{2j}. \tag{15}$$

3.2.4 Encoding. To encode data point x by PQ, the displacement between x and the corresponding anchor point is required first. Suppose data point x is quantized to the third-level region of anchor point a_{iuv} on second-level edge $l(a_{iu}, s_{iuv})$, the displacement can be computed as follows:

$$r_{q_l}(x) = x - a_{iuv}, \tag{16}$$

where a_{iuv} can be computed according to Equation 11.

The value of $r_{q_l}(x)$ is first computed by Equation 16 and encoded into m bytes using PQ [7]. The PQ codebooks are denoted by $C^1, \ldots, C^m$, each containing 256 sub-centroids. The vector $r_{q_l}(x)$ is mapped to a concatenation of m sub-centroids $(c^1_{j_1}, \cdots, c^m_{j_m})$, for j_i is a value between 1 and 256. Hence the vector $r_{q_l}(x)$ is encoded into an m-byte code of sub-centroid index $(j_1, \cdots, j_m)$. In Figure 1(c), assuming c_i is the closest centroid to x, we can observe that the anchor point of point x is closer to it than c_i. So the dataset points can be encoded more accurately with the same code length. This will improve the recall rate of search, as can be seen in our evaluation in Section 4.

3.3 Query

One important advantage of our VBLQ indexing structure is that at query time, a specific query point only needs to traverse a small number of regions whose edges are closest to the query point, as shown in Figure 2 (right). There are three steps for query processing: (1) region traversal, (2) distance computation and (3) re-ranking.

3.3.1 *Region traversal.*

The region traversal process consists of two steps: first-level regions traversal and second-level regions traversal. During first-level regions traversal, a query point y is quantized to its w_1 nearest first-level regions produced by quantizer q_v, and the w_1 nearest first-level regions correspond to $w_1 \cdot n_1$ second-level regions. The second-level regions traversal is then performed within only the $w_1 \cdot n_1$ second-level regions. Moreover, y is quantized again to $w_2 = \alpha_1 \cdot w_1 \cdot n_1$ nearest third-level regions by quantizer q^1_l.

Next, the third-level regions traversal is performed again within only the $w_2 \cdot n_2$ third-level regions. And y is further quantized again to $w_3 = \alpha_2 \cdot w_2 \cdot n_2$ nearest third-level regions by quantizer q^2_l. Then the candidate list of y is formed by the data points only within the w_3 nearest third-level regions.

Because the w_3 third-level regions is obviously smaller than the w_1 first-level regions, the candidate list produced by our VBLQ-based indexing structure is shorter than that produced by the VQ-based indexing structure, such as the one used by Faiss. This will result in a faster query speed of RobustiQ.

We use parameter α_1 to determine the portion of $w_1 \cdot n_1$ second-level regions and parameter α_2 to determine the portion of $w_2 \cdot n_2$ third-level regions to be traversed give a query, such that $w_3 = \alpha_1 \cdot \alpha_2 \cdot w_1 \cdot n_1 \cdot n_2$. In practice, we observed that $\alpha_1 = 0.25$ and $\alpha_2 = 0.5$ provide significant search acceleration with negligible (less than 1%) accuracy loss.

3.3.2 *Distance computation.*

Distance computation is a prerequisite condition for candidate re-ranking. Given a query point y and a candidate point x, the approximate distance between y and x can be evaluated by asymmetric distance computation (ADC) as follows [7]:

$$\|y - q_1(x) - q_2(x - q_1(x))\|^2 \tag{17}$$

where $q_1(x)$ is the anchor point on the closest line and $q_2(\cdots)$ is the PQ approximation of the x_i displacement.

Expression 17 can be further decomposed as follows [3]:

$$\|y - q_1(x)\|^2 + \|q_2(\cdots)\|^2 + 2\langle q_1(x), q_2(\cdots)\rangle - 2\langle y, q_2(\cdots)\rangle. \tag{18}$$

where $\langle\cdot,\cdot\rangle$ denotes the inner product between two points.

If the anchor point a_{iuv} is the closest anchor point to x, i.e., $q_1(x) = a_{iuv}$, Expression 18 can be transformed in following expression according to Expression 9 :

$$\underbrace{\|q_2(\cdots)\|^2 + 2\langle a_{iuv}, q_2(\cdots)\rangle}_{\text{term1}} + \underbrace{\|y - a_{iuv}\|^2}_{\text{term2}} - \underbrace{2\langle y, q_2(\cdots)\rangle}_{\text{term3}}. \tag{19}$$

According to Equation 3 and 11, term2 in Expression 19 can be computed in the following way:

$$\|y - a_{iuv}\|^2 = (1 - \lambda_{2i})\underbrace{\|y - a_{iu}\|^2}_{\text{term4}} + (\lambda^2_{2i} - \lambda_{2i})\underbrace{\|a_{iu} - s_{iuv}\|^2}_{\text{term5}} + \lambda_{2i}\underbrace{\|y - s_{iuv}\|^2}_{\text{term6}}. \tag{20}$$

According to Equation 3 and 10, term4 and term5 in Expression 20 can be computed in the following expression respectively:

$$(1 - \lambda_{2i})\|y - c_i\|^2 + (\lambda^2_{1i} - \lambda_{1i})\|c_i - s_{iu}\|^2 + \lambda_{1i}\|y - s_{iu}\|^2, \tag{21}$$

and

$$(1 - \lambda_{1i})\|s_{iuv} - c_i\|^2 + (\lambda^2_{1i} - \lambda_{1i})\|c_i - s_{iu}\|^2 + \lambda_{1i}\|s_{iuv} - s_{iu}\|^2. \tag{22}$$

In Expression 19 and Equation 20, some computations can be performed in advance and stored in a lookup table as follows:

- Term1 is independent of the query. It can be precomputed when the data point is encoded. We quantized it into 256 values and explicitly keep its quantized value as an additional byte for each data point. The lookup table just need 256 floating point values.
- Term2 are the distances from the query point to the closest second-level anchor points. They can be precomputed during region traversal.
- Term4 is the scalar product of the PQ sub-centroids and the corresponding query subvectors and can be computed independently before the search. Its computation costs $256 \times D$ multiply-adds [9].

The proposed decomposition is used to simplify the distance computation. According to Faiss, It need to maintain a large lookup table of $2 \cdot K \cdot m \cdot 256$ bytes for Term1. The lookup table grows with the codebook size and have a negative impact on query speed when it grows large. While our decomposition only need $4 \cdot 256$ bytes for Term1. Therefore the proposed decomposition is more scalable. Our decomposition only costs $256 \cdot D$ multiply-adds, while that of Faiss requires $256 \cdot D$ multiply-adds and $w_1 \cdot m \cdot 256$ additions [9]. Hence, the distance computation of RobustiQ is more efficient in terms of memory and runtime.

3.3.3 *Re-ranking.*

Re-ranking is the last step of the query process. It sorts the list of candidates data points by their distances to the query point. We employ the efficient sorting algorithm of [9] in our re-ranking step. Due to the shorter candidate list and the more efficient approximate distance computation, the re-ranking step of our system is both faster and more accurate than that of Faiss.

4 EXPERIMENTS AND EVALUATION

In this section, we evaluate the performance of our system RobustiQ in terms of query speed and query accuracy. We compare

RobustiQ against three state-of-the-art billion-scale ANN search systems, including Faiss [9], the best GPU-based system and two GPU-based retrieval systems Ivf-hnsw [4] and Multi-D-ADC [2]. All experiments are conducted on a machine with two 2.1GHz Intel Xeon E5-2620 v4 CPUs and two NVIDIA GTX Titan X GPUs with 12 GB memory each.

The evaluation is performed on two large public available datasets that are commonly used to evaluate billion-scale ANN search: SIFT1B [8] of 10^9 128-D vectors and DEEP1B [16] of 10^9 96-D vectors. Each dataset has a 10,000 query set with the precomputed ground-truth nearest neighbors. For RobustiQ, we sample 2×10^6 vectors from each dataset for learning all the trainable parameters. We evaluate query accuracy by Recall@k, which is the rate of queries for which the nearest neighbor is in the top k results.

4.1 Recall rate and query time

In experiment 1, we evaluate the recall rates and query time of all the systems. Each system is configured as follows:

1. **Faiss**. We construct three VQ codebooks of different number of centroids using Lloyd iteration: $K = 2^{18}/2^{17}/2^{16}$.
2. **Ivf-hnsw**. We use a codebook of $K = 2^{17}$ centroids as described in Sec. 2.1, and set 64 sub-centroids for each first-level centroid.[1]
3. **Multi-D-ADC**. We construct two codebook with different sizes: $K = 2^{12}/2^{14}$ and use the implementation from the Faiss library.
4. **RobustiQ**. We use VQ (the same codebooks as Faiss) as the first-level indexing structure, a 32-edge k-NN graph as the second-level indexing structure, and a 4-edge graph as the third-level indexing structure.

We report the Recall@k values for different $k = 1/10/100$, average query time on both datasets in milliseconds (ms), and speedup (su) over the slowest system. We compare RobustiQ with the current state-of-the-art GPU-based system Faiss with different codebook sizes in Table 1. For these two GPU-based systems, the experiments are performed on 1 GPU for the 8-byte encoding length, and on 2 GPUs for the 16-byte encoding length. From Table 1, a number of observations can be made.

(1) **Overall superiority.** RobustiQ outperforms Faiss in both accuracy and query time by a large margin, in all experimental settings except for R@100 with 16-byte codes for both datasets.

(2) **Substantial speedup.** For each codebook size, RobustiQ outperforms Faiss in query time. The speedup is especially prominent for the largest codebook ($K = 2^{18}$), where RobustiQ provides a speedup of ~3×.

(3) **Enhanced robustness.** The query speed of Faiss deteriorates considerably when the codebook size increases. For SIFT1B with 8-byte encoding length, the query speed of Faiss with codebook size $K = 2^{17}$ is 3x slower than that of Faiss with codebook size $K = 2^{16}$. In contrast, in the same circumstance, the query speed of RobustiQ with codebook size $K = 2^{17}$ is just 2× slower than $K = 2^{16}$, and the average query time of codebook size $K = 2^{17}$ only increased 0.03ms, comparing to the 1.83ms increase for Faiss.

[1]We use implementation of Ivf-hnsw that is available online (https://github.com/dbaranchuk/ivf-hnsw) for all the experiments.

Table 2 presents the comparison with the two best CPU-based systems, Ivf-hnsw and Multi-D-ADC. The codebook of size 2^{17} for RobustiQ is chosen for comparison as its performance is the most balanced between search quality and query speed. From the table we can make the following important observations.

(1) **Remarkable speedup.** RobustiQ is consistently faster than the two systems by an order of magnitude in all experiments. For all configurations, RobustiQ's query time is around 0.062 milliseconds, while the other systems' query times vary greatly. Compared to Ivf-hnsw, RobustiQ is on average 51.8× faster and up to 62× faster. RobustiQ is also consistently faster than Multi-D-ADC, by an average 11× speedup.

(2) **Competitive accuracy.** At the same time, on DEEP1B, RobustiQ achieves best recall while being also the fastest. On SIFT1B, RobustiQ does not exhibit best recall values, though the difference from the best values are quite small.

4.2 Robustness comparison

Table 1 gives a detailed account of RobustiQ's performance advantage over Faiss with different codebook sizes. In this experiment we compare the robustness of RobustiQ against Faiss. Figure 3 shows, for SIFT1B and DEEP1B, the change in recall and query time for Faiss and RobustiQ for the same codebook sizes and encoding length. As can be seen in the figure, the recall value of both systems naturally increase with the increase in codebook size. More importantly, the query time increase of RobustiQ is much slower than that of Faiss, clearly demonstrating RobustiQ's superior robustness over Faiss.

We also evaluated the performance of RobustiQ with different parameter settings and found that increasing the number of second-level and third-level edges can improve search accuracy, while slightly increasing query time.

5 CONCLUSION

Approximate nearest neighbor (ANN) search has important application in large-scale machine learning and computer vision research, and has recently gained significant attention. GPU-based methods such as Faiss are able to handle billion-scale datasets, and they represent the state-of-the-art in ANN performance, in terms of both search accuracy speed. However, the vector quantization (VQ)-based indexing structure of Faiss is not *robust* enough, so that the increase in search accuracy is achieved only with significantly reduced search speed.

We present RobustiQ, a GPU-based ANN search method that addresses the inherent robustness limitation of VQ. RobustiQ incorporates two congruent components: (1) a novel hierarchical indexing structure comprising one layer of vector quantization and two layers of line quantization that induces smaller quantization errors than VQ, and (2) an indexing and encoding method that uses sampling to improve space efficiency of indexing and time complexity of querying. Together, they allow RobustiQ to optimise both search accuracy and speed.

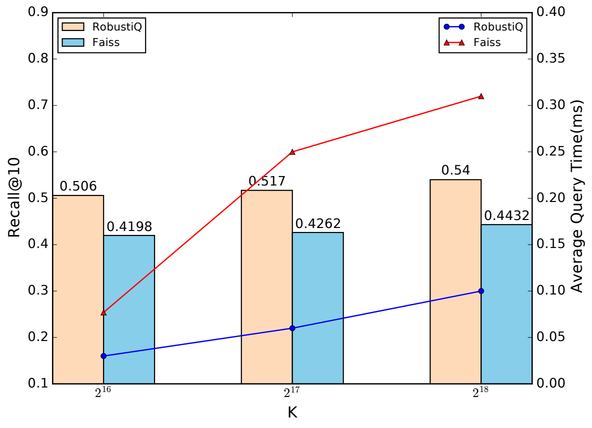
Our comprehensive evaluation on two public billion-scale benchmarks, SIFT1B and DEEP1B, show the overall superior performance of RobustiQ over Faiss in both search accuracy and speed, achieving an average speedup of 2–3×. We also demonstrate the improved

Table 1: Performance comparison between RobustiQ and Faiss of recall@1/10/100, query time (ms) and speedup (su) on SIFT1B and DEEP1B. We compare the two systems under three codebooks of different sizes (K). Speedup is computed against the slowest system. Best result in each column is bolded, and second best is underlined.

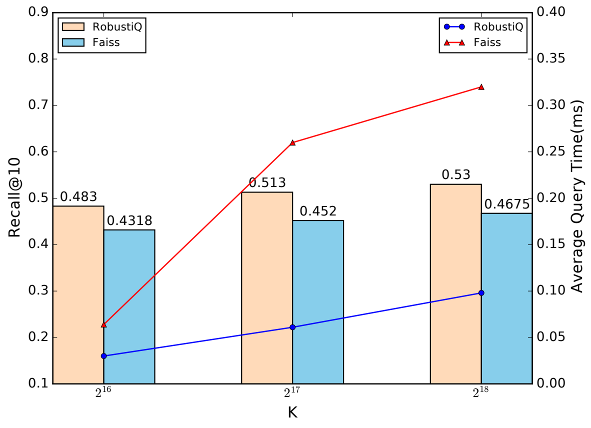
System (K)	SIFT1B										DEEP1B									
	8 bytes					16 bytes(2 GPUs)					8 bytes					16 bytes(2 GPUs)				
	R@1	R@10	R@100	t (ms)	su	R@1	R@10	R@100	t (ms)	su	R@1	R@10	R@100	t (ms)	su	R@1	R@10	R@100	t (ms)	su
Faiss (2^{18})	0.1383	0.4432	0.7978	0.31	1×	0.3180	0.7825	**0.9618**	0.280	1×	0.2101	0.4675	0.7438	0.32	1×	0.3793	0.7650	**0.9509**	0.33	1×
Faiss (2^{17})	0.1310	0.4262	0.7833	0.25	1.24×	0.3077	0.7590	0.9445	0.10	2.8×	0.2038	0.4520	0.7330	0.26	1.2×	0.3715	0.7517	0.9361	0.18	1.8×
Faiss (2^{16})	0.1231	0.4198	0.7763	0.077	4×	0. 3018	0.7569	0.9556	0.064	4.37×	0.1976	0.4318	0.7173	0.064	5×	0.3658	0.7415	0.9411	0.063	5.2×
RobustiQ (2^{18})	**0.173**	**0.540**	**0.860**	0.10	3.1×	**0.3535**	**0.8115**	0.9538	0.11	2.54×	**0.240**	**0.530**	**0.817**	0.098	3.3×	**0.4047**	**0.7967**	0.9469	0.10	3.3×
RobustiQ (2^{17})	0.165	0.517	0.840	0.060	5×	0.3488	0.7925	0.9355	0.062	4.5×	0.231	0.513	0.782	0.061	5.2×	0.395	0.7676	0.925	0.060	5.5×
RobustiQ (2^{16})	0.1623	0.5063	0.819	**0.030**	**10×**	0.3319	0.7640	0.9004	**0.033**	**8.5×**	0.217	0.483	0.76	**0.030**	**10.7×**	0.3800	0.7511	0.8964	**0.030**	**11×**

Table 2: Performance comparison between RobustiQ and two CPU-based systems of recall@1/10/100, query time (ms) and speedup (su) on SIFT1B and DEEP1B. For each system the codebook size is specified beneath each system's name. Speedup is computed against the slowest system. Best result in each column is bolded, and second best is underlined.

System (K)	SIFT1B										DEEP1B									
	8 bytes					16 bytes					8 bytes					16 bytes				
	R@1	R@10	R@100	t (ms)	su	R@1	R@10	R@100	t (ms)	su	R@1	R@10	R@100	t (ms)	su	R@1	R@10	R@100	t (ms)	su
Ivf-hnsw (2^{17})	**0.1708**	0.5247	0.8217	2.57	1×	**0.3575**	0.7796	0.8855	3.55	1×	0.230	0.506	0.7724	2.82	1×	0.3646	0.7491	0.8804	3.72	1×
Multi-D-ADC ($2^{14} \times 2^{14}$)	0.1703	**0.5270**	**0.8530**	0.423	6×	0.3572	**0.8048**	0.9314	0.587	6×	0.2034	0.4543	0.7389	0.632	4.5×	0.3693	0.7405	0.9190	1.065	3.5×
Multi-D-ADC ($2^{12} \times 2^{12}$)	0.1420	0.4720	0.8183	0.367	7×	0.3324	0.8029	**0.9752**	1.603	2.2×	0.1874	0.4240	0.6979	0.839	3.4×	0.3557	0.7087	0.9059	1.52	2.4×
RobustiQ (2^{17})	0.165	0.517	0.840	**0.060**	42.1×	0.3488	0.7925	0.9355	**0.062**	57.2×	**0.231**	**0.513**	**0.782**	**0.061**	46.2×	**0.395**	**0.7676**	**0.925**	**0.060**	62×



SIFT1B



DEEP1B

Figure 3: Comparison of recall@10 and average query time between RobustiQ and Faiss under the same codebook sizes. The two systems are compared with 8-byte encoding length. The x axis indicates the 3 different codebook sizes ($K = 2^{16}/2^{17}/2^{18}$). The left y axis is the recall@10 value and the right y axis is the average query time (ms).

robustness over Faiss, as RobustiQ is much less susceptible to the negative impact on query speed from increase in search accuracy. Compared with the two best CPU-based ANN systems, RobustiQ achieves up to 58.3× speedup .

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China under Grant No.61672254.No.61672246 and No.61272068.

REFERENCES

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2017. Accelerated Nearest Neighbor Search with Quick ADC. In *Proceedings of the 2017 ACM on*

International Conference on Multimedia Retrieval, ICMR 2017, Bucharest, Romania, June 6-9, 2017. 159–166. https://doi.org/10.1145/3078971.3078992

[2] Artem Babenko and Victor Lempitsky. 2012. The inverted multi-index. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 3069–3076.

[3] Artem Babenko and Victor Lempitsky. 2014. Improving Bilayer Product Quantization for Billion-Scale Approximate Nearest Neighbors in High Dimensions. *Eprint Arxiv* (2014).

[4] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 202–216.

[5] Sik Kim Dong and Ness B. Shroff. 1999. Quantization based on a novel sample-adaptive product quantizer (SAPQ). *IEEE Transactions on Information Theory* 45, 7 (1999), 2306–2320.

[6] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2946–2953.

[7] H Jégou, M Douze, and C Schmid. 2011. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117.

[8] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*. IEEE, 861–864.

[9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734* (2017).

[10] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *Computer Vision and Pattern Recognition*. 2329–2336.

[11] Yoseph Linde, Andres Buzo, and Robert Gray. 1980. An algorithm for vector quantizer design. *IEEE Transactions on communications* 28, 1 (1980), 84–95.

[12] Mohammad Norouzi and David J. Fleet. 2013. Cartesian K-Means. In *IEEE Conference on Computer Vision and Pattern Recognition*. 3017–3024.

[13] Marco Patella and Paolo Ciaccia. 2009. Approximate similarity search: A multi-faceted problem. *Journal of Discrete Algorithms* 7, 1 (2009), 36–48.

[14] Josef Sivic and Andrew Zisserman. 2003. Video Google: a text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*. 1470–1477.

[15] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik Lensch. 2016. Efficient large-scale approximate nearest neighbor search on the GPU. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2027–2035.

[16] Artem Babenko Yandex and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *Computer Vision and Pattern Recognition*. 2055–2063.