Vector and Line Quantization for Billion-scale Similarity Search on GPUs

Wei Chen[a], Jincai Chen[a,b,*], Fuhao Zou[c,*], Yuan-Fang Li[d], Ping Lu[a,b], Qiang Wang[a], Wei Zhao[b]

[a] *Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China*
[b] *Key Laboratory of Information Storage System of Ministry of Education, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*
[c] *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074,China*
[d] *Faculty of Information Technology, Monash University, Clayton 3800, Australia*

Abstract

Billion-scale high-dimensional approximate nearest neighbour (ANN) search has become an important problem for searching similar objects among the vast amount of images and videos available online. The existing ANN methods are usually characterized by their specific indexing structures, including the inverted index and the inverted multi-index structure. The inverted index structure is amenable to GPU-based implementations, and the state-of-the-art systems such as Faiss are able to exploit the massive parallelism offered by GPUs. However, the inverted index requires high memory overhead to index the dataset effectively. The inverted multi-index structure is difficult to implement for GPUs, and also ineffective in dealing with database with

*Corresponding author
Email addresses: jcchen@hust.edu.cn (Jincai Chen), fuhao_zou@hust.edu.cn (Fuhao Zou)

different data distributions. In this paper we propose a novel hierarchical inverted index structure generated by vector and line quantization methods. Our quantization method improves both search efficiency and accuracy, while maintaining comparable memory consumption. This is achieved by reducing search space and increasing the number of indexed regions.

We introduce a new ANN search system, VLQ-ADC, that is based on the proposed inverted index, and perform extensive evaluation on two public billion-scale benchmark datasets SIFT1B and DEEP1B. Our evaluation shows that VLQ-ADC significantly outperforms the state-of-the-art GPU- and CPU-based systems in terms of both accuracy and search speed. The source code of VLQ-ADC is available at `https://github.com/zjuchenwei/vector-line-quantization`.

Keywords: Quantization; Billion-scale similarity search; high dimensional data; Inverted index; GPU

1. Introduction

In the age of the Internet, the amount of images and videos available online increases incredibly fast and has grown to an unprecedented scale. Google processes over 40,000 various queries per second, and handles more than 400 hours of YouTube video uploads every minute [1]. Every day, more than 100 million photos/videos are uploaded to Instagram, more than 300 million uploaded to Facebook, and a total of 50 billion photos have been shared to Instagram[1]. As a result, scalable and efficient search for similar

[1] `https://www.omnicoreagency.com/instagram-statistics/`

images and videos on the billion scale has become an important problem and it has been under intense investigation.

As online images and videos are unstructured and usually unlabeled, it is hard to compare them directly. A feasible solution is to use real-valued, high-dimensional vectors to represent images and videos, and compare the distances between the vectors to find the nearest ones. Due to the curse of dimensionality [2], it is impractical for multimedia applications to perform exhaustive search in billion-scale datasets. Thus, as an alternative, many *approximate nearest neighbor* (ANN) search algorithms are now employed to tackle the billion-scale search problem for high-dimensional data. Recent best-performing billion-scale retrieval systems [3–8] typically utilize two main processes: indexing and encoding.

To avoid expensive exhaustive search, these systems use *index structures* that can partition the dataset space into a large number of disjoint regions, and the search process only collects points from the regions that are closest to the query point. The collected points then form a short list of candidates for each query point. The retrieval system then calculates the distance between each candidate and the query point, and sort them accordingly.

To guarantee query speed, the indexed points need to be loaded into RAM. For large datasets that do not fit in RAM, dataset points are *encoded* into a compressed representation. Encoding has also proven to be critical for memory-limited devices such as GPUs that excel at handling data-parallel tasks. A high-performance CPU like Intel Xeon Platinum 8180 (2.5 GHz, 28 cores) performs 1.12 TFLOP/s single precision peak performance[2]. In

[2]`https://ark.intel.com/content/www/us/en/ark/products/120496/`

contrast, GPUs like NVdia Tesla P100 can provide up to 10T FLOP/s single precision peak performance[3], and are good choices for high performance similarity search systems. Many encoding methods have been proposed, including hashing methods and quantization methods. Hashing methods encode data points to compact binary codes through a hash function [9, 10], and quantization methods, typically product quantization (PQ), map data points to a set of centroids and use the indices of the centroids to encode the data points [11, 12]. By hashing methods, the distance between two data points can be approximated by the Hamming distance between their binary code. By quantization methods, the Euclidean distance between the query and compressed points can be computed efficiently. It has been shown in the literature that quantization encoding can be more accurate than various hashing methods [11, 13, 14].

Jégou et al. [11] first introduced an index structure that is able to handle billion-scale datasets effieciently. It is based on the inverted index structure that partitions the high dimensional vector space into Voronoi regions for a set of centroids obtained by a quantization method called vector quantization (VQ) [15]. This system, called IVFADC, achieves reasonable recall rates in several tens of milliseconds. However, the VQ-based index structure needs to store a large set of full dimensional centroids to produce a huge number of regions, which would require a large amount of memory.

An improved inverted index structure called the inverted multi-index

intel-xeon-platinum-8180-processor-38-5m-cache-2-50-ghz.html

[3]https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf

(IMI) was later proposed by Babenko and Lempitsky [16]. The IMI is based on product quantization (PQ), which divides the point space into several orthogonal subspaces and clusters the subspaces into Voronoi regions independently. The Cartesian product of regions in each subspace forms regions in the global point space. The strength of the IMI is that it can produce a huge number of regions with much smaller codebooks than that of the inverted index. Due to the huge number of indexed regions, the point space is finely partitioned and each regions contains fewer points. Hence the IMI can provide accurate and concise candidate lists with memory and runtime efficiency.

However, it has been observed that for some billion-scale datasets, the majority of the IMI regions contain no points [5], which is a waste of index space and has a negative impact on the final retrieval performance. The reasons for this deficiency is that the IMI learns the centroids independently on the subspaces which are not statistically independent [7]. In fact, some convolutional neural networks (CNN) produce feature vectors with considerable correlations between the subspaces [10, 17, 18].

The high level of parallelism provided by GPUs has recently been leveraged to accelerate similarity search of high-dimensional data, and it has been demonstrated that GPU-based systems are more efficient than CPU-based systems by a large margin [4, 6]. Comparing to IMI structure, the inverted indexing structure proposed by Jégou et al. [11] is more straightforward to parallelize, because the IMI structure depends on a complicated multi-sequence algorithm, which is sequential in nature [4] and hard to parallelize.

To the best of our knowledge, there are two high performance systems

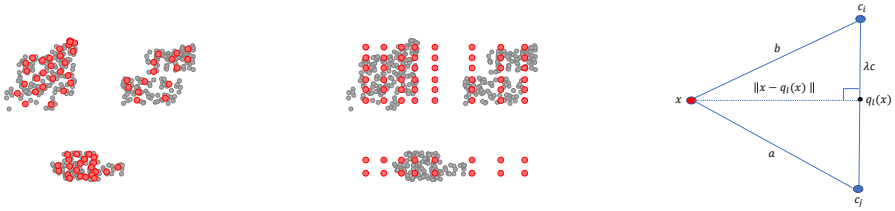
that are able to handle ANN search for billion-scale datasets on the GPU: PQT [4] and Faiss [6]. PQT proposes a novel quantization method call line quantization (LQ) and is the first billion-scale similarity retrieval system on the GPU. Subsequently Faiss implements the idea of IVFADC on GPUs and currently has the state-of-the-art performance on GPUs. We compare our method against Faiss and two other systems in Section 5.

In this paper, we present VLQ-ADC, a novel billion-scale ANN similarity search framework. VLQ-ADC includes a two-level hierarchical inverted indexing structure based on Vector and Line Quantization (VLQ), which can be implemented on GPU efficiently. 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 describe how to encode data points via a novel algorithm for high runtime/memory efficiency.
3. Our evaluation 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 VLQ, our approach for approximate nearest neighbor (ANN)-based similarity search method. Section 4 introduces the details of GPU implementation. Section 5 provides a series of experiments, and compares the results to the state of the

art.



(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.

2. Related work

In this section, we briefly introduce some quantization methods and several retrieval systems related to our approach. Table 1 summarizes the common notations used throughout this paper. 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)

In vector quantization [15] (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

Table 1: Commonly used notations.

Notation	Description
x_i, D	data points, their dimension and the number of data points
$\mathcal{X}$, N	a set of data points and its size, $\mathcal{X} = \{x_1, \ldots, x_N\} \subset \mathbb{R}^D$
$c, s, l(c, s)$	centroids, nodes and edges
m	encoding length
k	the number of first-level centroids
n	the number of edges of each first-level centroid
w_1	the number of first-layer nearest regions for a query
α	the portion of the nearest of the $w \cdot n$ second-level regions
w_2	the number of second-level nearest regions for a query, $w_2 = \alpha \cdot w_1 \cdot n$
λ	a scalar parameter for line quantization
r	displacement from data points to the approximate points

a codebook of size k. We can use Lloyd iterations [19] to efficiently obtain a codebook C on a subset of the dataset. For a finite dataset, $\mathcal{X}$, $q_v(x)$ induces quantization error E:

$$E = \sum_{x \in \mathcal{X}} \parallel x - q_v(x) \parallel^2 . \tag{1}$$

According to Lloyd's first condition, to minimize quantization error a quantizer should map vector x to its nearest codebook centroid.

$$q_v(x) = \arg\min_{c \in C} \parallel x - c \parallel . \tag{2}$$

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.

The **inverted index structure** based on VQ [11] can split the dataset space into k regions that correspond to the k centroids of the codebook. Since the ratio of regions to centroids is $1:1$, it requires a large amount of space to store the D-dimensional centroids when k is large. This would give a negative effect on the performance of the retrieval system. Our hierarchical index structure based on VLQ increase the ratio by n times, i.e., n times more regions can be generated by our indexing structure with the same number of centroids as the VQ based indexing structure.

2.2. Product quantization (PQ)

Product quantization (Figure 1 (b)) is an extension of vector quantization. Assuming that the dimension D is a multiple of m, any vector $x \in \mathbb{R}^D$ can be regarded as a concatenation $(x^1, \cdots, x^m)$ of m sub-vectors, each of dimension D/m. Suppose that $C^1, \cdots, C^m$ are m codebooks of subspace $\mathbb{R}^{D/m}$, each owns k D/m-dimensional sub-centroids. A codebook of a product quantizer q_p is thus a Cartesian product of sub-codebooks.

$$C = C^1 \times \cdots \times C^m. \tag{3}$$

Hence the codebook C contains a total of k^m centroids, each is a form of $c = (c^1, \cdots, c^m)$, where each sub-centroid $c^i \in C^i$ for $i \in \mathcal{M} = \{1, \cdots, m\}$. A product quantizer q_p should minimize the quantization error E defined in Formula 1. Hence, for $x \in \mathbb{R}^D$, the nearest centroid in codebook C is

$$q_p(x) = (q_p^1(x^1), \cdots, q_p^m(x^m)), \tag{4}$$

where q^i is a sub-quantizer of q and $q_p^i(x)$ is the nearest sub-centroid for sub-vector x^i, i.e., the nearest centroid $q_p(x)$ for x is the concatenation of the nearest sub-centroids for sub-vector x^i.

The **inverted multi-index structure** (IMI) applies the idea of PQ for indexing and can generate k^m regions with m codebooks of k sub-centroids each. The benefit of inverted multi-index is thus it can easily generate a much larger number of regions than that of VQ-based inverted index structure with moderate values of m and k. The drawback of IMI is that it produces a lot of empty regions when the distributions of subspaces are not independent [5]. This will affect the system's performance when handling datasets which have significant correlations between different subspaces, such as CNN-produced feature point dataset [5].

The PQ-based indexing structure later has been improved by OPQ [20] and LOPQ [12]. OPQ make a rotation on dataset points by a global $D \times D$ rotation matrix and LOPQ rotates the points which belong to the same cell by a same local $D \times D$ rotation matrix to minimize correlations between two subspaces [20]. OPQ and LOPQ can both improve the indexing efficiency of PQ but slow down the query speed by a large margin as well.

Additionally, PQ can also 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 $(c_{j_1}^1, \cdots, c_{j_m}^m)$, 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 [11], the computation is performed based on lookup tables.

$$\| y - x \|^2 \approx \| y - q_p(x) \|^2 = \sum_{i=1}^{m} \| y^i - c^i_{j_i} \|^2 \tag{5}$$

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 $\mathcal{O}(D)$ to $\mathcal{O}(m)$. 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, 7, 12, 14, 20].

2.3. Line quantization (LQ)

Line quantization (LQ) [4] 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 $l(c_i, c_j)$. A line quantizer q_l quantizes a point x to the nearest line as follows:

$$q_l(x) = \arg\min_{l(c_i,c_j)} d(x, l(c_i, c_j)), \tag{6}$$

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

$$\begin{aligned} d(x, l(c_i, c_j))^2 = (1 - \lambda) \| x - c_i \|^2 + (\lambda^2 - \lambda) \| c_j - c_i \|^2 \\ + \lambda \| x - c_j \|^2 \end{aligned} \tag{7}$$

Because the values of $\| x - c_j \|^2, \| x - c_i \|^2, \| c_j - c_i \|^2$ can be pre-computed between x and all centroids, Equation 7 can be calculated efficiently. The

anchor point of x is represented by $(1-\lambda)\cdot c_i + \lambda \cdot c_j$, where λ is a scalar parameter that can be computed as following:

$$\lambda = 0.5 \cdot \frac{(\parallel x - c_i \parallel^2 + \parallel c_j - c_i \parallel^2 - \parallel x - c_j \parallel^2)}{\parallel c_j - c_i \parallel^2}. \tag{8}$$

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

$$r_{ql}(x) = x - ((1-\lambda)\cdot c_i + \lambda \cdot c_j). \tag{9}$$

Here we regard $l(c_i, c_j)$ and $l(c_j, c_i)$ as two different lines. So LQ-based indexing structrue can produce $k \cdot (k-1)$ regions with a codebook of k centroids, The benefit of 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 x when k is large. So we use LQ as an indexing approach with a codebook of a few lines.

Table 2: A summary of current state-of-the-art retrieval systems based on quantization method. N is the size of the dataset $\mathcal{X}$, m is the number of sub-vectors in product quantizatino (PQ), k is the size of the codebook, and n is the number of second-level regions. In the last column of each row, the first term is the complexity for encoding, and the second term is the complexity for indexing.

System	Index structure	Encoding	CPU/GPU	Space complexity
Faiss [6]	VQ	PQ	GPU	$\mathcal{O}(N \cdot m) + \mathcal{O}(k \cdot D)$
Ivf-hnsw [7]	2-level VQ	PQ	CPU	$\mathcal{O}(N \cdot m) + \mathcal{O}(k \cdot (D+n))$
Multi-D-ADC [16]	IMI (PQ)	PQ	CPU	$\mathcal{O}(N \cdot m) + \mathcal{O}(k \cdot (D+k))$
VLQ-ADC (our system)	VLQ	PQ	GPU	$\mathcal{O}(N \cdot m) + \mathcal{O}(k \cdot (D+n))$

2.4. The applications of VQ-based and PQ-based indexing structures for billion-scale dataset

In this subsection we introduce several billion-scale similarity retrieval systems that apply VQ- or PQ-based indexing structure and encoded by PQ, and discuss their strengths and weaknesses.

All the systems discussed below are best-performing, state-of-the-art systems for billion-scale high-dimensional ANN search. Their indexing structure and encoding method are summarized in Table 2. Since all these systems employ the same encoding method based on PQ, we will mainly focus on their indexing structures in the discussions below.

Faiss [6] is a very efficient GPU-based retrieval approach, by realizing the idea of IVFADC [11] on GPUs. Faiss uses the inverted index based on VQ [21] 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

$$r_q(x) = x - q(x), \tag{10}$$

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

The search process of Faiss/IVFADC proceeds as follows:

1. A query point y is quantized to its w nearest regions, extracting a list of candidates $\mathcal{L}_c \subset \mathcal{X}$ which have a high probability of containing the

nearest neighbor.

2. The displacement of the query point y from the centroid of each sub-region is computed as $r_q(y)$.
3. The distances between $r_q(y)$ and PQ-encoded displacements in $\mathcal{L}_c$ are then computed according to Formula 5.
4. Sort the list $\mathcal{L}_c$ to be $\mathcal{L}_s$ based on the distances computed above. The first points in $\mathcal{L}_s$ are returned as the search result for query point y.

Ivf-hnsw [7] is a retrieval system based on a two-level inverted index structure. Ivf-hnsw first splits the data space into k regions like IVFADC. Then each region is further split into several sub-regions that correspond to n sub-centroids. Each sub-centroid of a region can be represented by the centroid of the region and another centroid of a neighbor region. Assume that each region has n neighbor regions, thus each region can be split into n regions. Each data point is first quantized to a region and then further quantized to a sub-region of the region. The displacement of each point from the sub-centroid of a sub-region it belongs to is encoded by PQ. An inverted list of data point is maintained for each sub-regions similar to IVFADC.

The search process of Ivf-hnsw proceeds as follows:

1. A query point y is quantized to its w first-level nearest regions, giving $w \cdot n$ sub-regions.
2. Among the $w \cdot n$ sub-regions, y is secondly quantized to $0.5 \cdot w \cdot n$ nearest sub-regions, generating a list of candidates $\mathcal{L}_c \subset \mathcal{X}$.
3. The displacement of the query point y from the sub-centroid of each sub-region is computed as $r_q(y)$.

4. The distances between $r_q(y)$ and PQ-encoded displacements in $\mathcal{L}_c$ are then computed according to Formula 5.
5. The re-ordering process of Ivf-hnsw is similar to IVFADC/Faiss.

Multi-D-ADC [16] is based on the inverted multi-index which is currently the state-of-the-art indexing method for high-dimensional large-scale datasets. An inverted multi-index of Multi-D-ADC usually owns a product quantizer with two sub-quantizers q^1, q^2 for subspace $\mathbb{R}^{D/2}$, each of k sub-centroids. A region in the D-dimensional space is now a Cartesian product of two corresponding subspace regions. So the IMI can produce k^2 regions. For each point $x = (x^1, x^2) \in \mathcal{X}$, sub-vectors $x^1, x^2 \in \mathbb{R}^{D/2}$ are separately quantized to subspace regions of $q^1(x^1), q^2(x^2)$ respectively, and x is then quantized to the region of $(q^1(x^1), q^2(x^2))$. The displacement of each point x from the centroid $(q^1(x^1), q^2(x^2))$ is also encoded by PQ, and an inverted list of points is again maintained for each region.

The search process of Multi-D-ADC proceeds as follows:

1. For a query point $y = (y^1, y^2)$, The Euclidean distances of each of sub-vectors y^1, y^2 to all sub-centroids of q^1, q^2 are computed respectively. The distance of y to a region can be computed according to Formula 5 for $m = 2$.
2. Regions are traversed in ascending order of distance to y by the multi-sequence algorithm [16] to generate a list of candidates $\mathcal{L}_c \subset \mathcal{X}$.
3. The displacement of the query point y from the centroid (c^1, c^2) of each region is computed as $r_q(y)$ as well.
4. The re-ordering process of Multi-D-ADC is similar to IVFADC/Faiss.

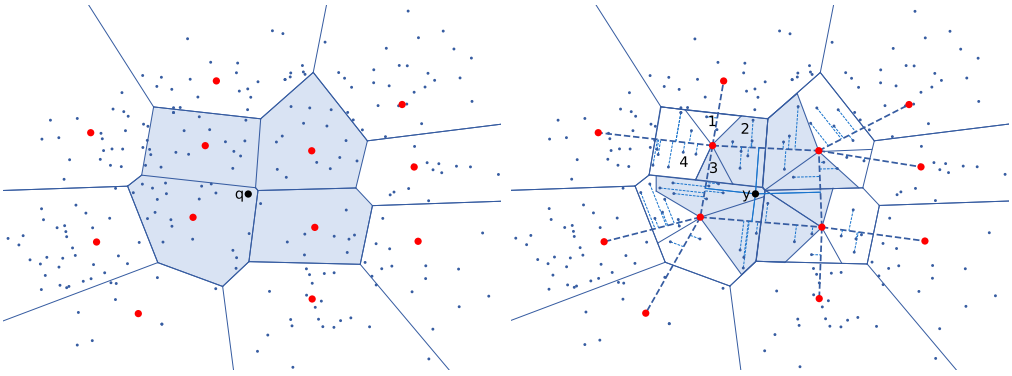
The VQ-based indexing structure requires a large full-dimensional codebook to produce regions when k is large. The PQ-based indexing structure are not suitable for all datasets, especially for those produced by convolutional neural networks (CNN) [7]. The novel VQ-based indexing structure proposed by Ivf-hnsw can produce more regions than the prior VQ-based indexing structure. However its performance on the codebook of small size is not good enough. We will discuss that in Sec.5. In comparison, our indexing structure is efficient with a small size of codebook which can accelerate query speed and at the same time is suitable for any dataset irrespective of the presence/absence of correlations between subspaces.

3. The VLQ-ADC System

In this section we introduce our GPU-based similarity retrieval system, VLQ-ADC, that contains a two-layer hierarchical indexing structure based on vector and line quantization and an asymmetric distance computation method. VLQ-ADC incorporates a novel index structure that can index the dataset points efficiently (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 existing systems above, One major advantage of our system is that our indexing structure can generate shorter and more accurate candidate list for the query point, which will accelerate query speed by a large margin. Another advantage of our system is that the improved asymmetric distance computation method base on PQ encoding method provide a higher search accuracy. In the remainder of this section we will use Figure 2 to illustrate our framework. We recall that commonly used notations are

summarized in Table 1.



VQ-based indexing structure VLQ-based indexing structure

Figure 2: A comparison of the indexing structure and search process of the VQ-based indexing structure (**left**) and our VLQ-based indexing structure (**right**) on data points (small blue dots) of dimension 2 ($D = 2$). The large red dots denote the (first-level) same cell centroids in both figures. **Left:** The 4 shaded areas in the left figure represent the first-level regions, one for each centroid, and they make up the areas that need to be traversed for the query point q. **Right:** For each centroid in the right figure, $n = 4$ nearest neighboring centroids are found. Thus the n-NN graph consists of all the centroids and the edges (thick dashed lines) between them. Each first-level region in the right figure consists of 4 second-level regions, each of which represent the data points closet to the corresponding edge in the n-NN graph as denoted by the line quantizer q_l. Given the query point q and parameter $\alpha = 0.5$, only half of the second-level subregions (shaded in blue) need to be traversed. As can be seen, VLQ allows search to process substantially smaller regions in the dataset than a VQ-based approach.

3.1. The VLQ-based index structure

For billion-scale datasets with a moderate number of regions (e.g., 2^{16}) produced by vector quantization (VQ), the number of data points in most regions is too large, which negatively affects search accuracy. To alleviate

this problem, we propose a hierarchical indexing structure. In our structure, each list is split into several shorter lists, i.e., each region is divided into several subregions, using line quantization (LQ).

Our indexing structure is a two-layer hierarchical structure which consists of two levels 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. The second level contains a line quantizer q_l with 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. In each first-level region, the line quantizer q_l then quantizes each data point to the closest edge in the n-NN graph, thus splitting the region into n second-level regions.

As an example, in the right side of Figure 2, given $n = 4$, the top left first-level region is further divided into 4 subregions by q_l, enclosed by solid lines and denoted 1, 2, 3, and 4. Each subregion contains all the data points that are closest to a given edge of the n-NN graph, as calculated by the line quantization q_l.

Training the codebook. We use Lloyd iteration in the fashion of the Linde-Buzo-Gray algorithm [15] to obtain the codebook of the VQ quantizer q_v. The n-NN graph is then built on the centroids of the codebook.

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-NN graph only needs to store (1)

the indices of its n nearest neighbors and (2) the distances to its n nearest neighbors, which amounts to $k \cdot n \cdot (sizeof(\texttt{int}) + sizeof(\texttt{float}))$ bytes. Note we do not need to store the full-dimensional points. For a typical values of $k = 2^{16}$ centroids and $n = 32$ subcentroids, the additional memory overhead for storing the graph is $2^{16} \cdot 32 \cdot (32 + 32)$ bits (16 MB), which is acceptable for billion-scale datasets.

One way to produce the subregions is by utilizing vector quantization (VQ) again in each region. However, that would require storing full-dimensional subcentroids and thus consume too much additional memory. For the same configuration ($k = 2^{16}$ centroids and $n = 32$ subcentroids) and a dimension of $D = 128$, the additional memory overhead for a VQ-based hierarchical indexing structure would be $2^{16} \cdot 32 \cdot 128 \cdot sizeof(\texttt{float})$ additional bits (1,024 MB). As can be seen, our VLQ-based hierarchical indexing structure is substantially more compact, only consuming 1/64 of the memory required by a VQ-based approach for the second-level codebook.

We note that the PQ-based indexing structure requires $\mathcal{O}(k \cdot (D + k))$ memory to maintain the indexing structure (Table 2), 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 $\mathcal{O}(k \cdot (D + n))$, where typically $n \ll k$ (n is much smaller than k), hence making our index much more memory efficient.

3.2. Indexing and encoding

In this subsection, we will describe the indexing and encoding process and summarize both processes in Algorithm 1 and 2 respectively.

For our two-level index structure, the indexing process comprises two

Algorithm 1 VLQ-ADC batch indexing process

function INDEX($[x_1, \ldots, x_N]$)
  for $t \leftarrow 1 : N$ **do**
    $x_t \mapsto q_v(x) = \arg\min_{c \in C} \parallel x_t - c \parallel^2$ `// VQ`
    $S_i = n\text{-}\arg\min_{c \in C} \parallel c - c_i \parallel^2$ `// Construct the` n`-NN graph`
    $x_t \mapsto q_l(x) = \arg\min_{l(c_i, s_{ij}), s_{ij} \in S_i} d(x, l(c_i, s_{ij}))$ `// LQ`
  end for
end function

different quantization procedures, one for each layer. Similar to the IVFADC scheme, each dataset point is quantized by the vector quantizer q_v to the first-level regions surrounded by the dotted lines in Figure 2. These regions form a set of inverted lists as search candidates.

We describe the second-level indexing process as follows. Let $\mathcal{X}^i$ be a region of $\{x_1, \ldots, x_l\}$ that corresponds to a centroid c_i, for $i \in \{1, \ldots, k\}$. In constructing the n-NN graph, let $S_i = \{s_{i1}, \ldots, s_{in}\}$ denote the set of the n centroids closest to c_i and $l(c_i, s_{ij})$ denote an edge between c_i and s_{ij}, for $j \in \{1, \ldots, n\}$. The points in $\mathcal{X}^i$ are quantized to the subregions by a line quantizer q_l with a codebook $\mathcal{E}_i$ of n edges $\{l(c_i, s_{i1}), \ldots, l(c_i, s_{in})\}$. Thus the region $\mathcal{X}^i$ is split into n subregions $\{\mathcal{X}^i_1, \ldots, \mathcal{X}^i_n\}$ and each point $x \in \mathcal{X}^i$ is quantized to a second-level subregion $\mathcal{X}^i_j$. So the entire space $\mathcal{X}$ are divided into $k \times n$ second-level subregions.

$$\mathcal{X}^i_j = \{x \in \mathcal{X}^i \mid q_l(x) = l(c_i, s_{ij})\}, \text{ for all } i \in \{1 \ldots k\} \tag{11}$$

Each data point in the dataset $\mathcal{X}$ is assigned to one of the $k \cdot n$ cells. When the data point x is quantized to the sub-region of edge $l(c_i, s_{ij})$, according to

Algorithm 2 VLQ-ADC batch encoding process

function ENCODE($[x_1, \ldots, x_N]$)
  for $t \leftarrow 1 : N$ **do**
    $r_{q_l}(x_t) = x_t - ((1 - \lambda_{ij}) \cdot c_i + \lambda_{ij} \cdot s_{ij})$ `// Equation 12`
    let $r_t = r_{q_l}(x_t)$ `// displacement`
    $r_t = [r_t^1, \ldots, r_t^m]$ `// divide` r_t `into` m `subvectors`
    for $p \leftarrow 1 : m$ **do**
      $r_t^p \mapsto c_{j_p} = \arg\min_{c_{j_p} \in C^p} \parallel r_t^p - c_p \parallel^2$
    end for
    $Code_t = (j_1, \ldots, j_m)$
  end for
end function

the Equation 9 and 8 the displacement of x from the corresponding anchor point can be computed as following:

$$r_{q_l}(x) = x - ((1 - \lambda_{ij}) \cdot c_i + \lambda_{ij} \cdot s_{ij}), \text{ where} \tag{12}$$

$$\lambda_{ij} = -0.5 \cdot \frac{(\parallel x - s_{ij} \parallel^2 - \parallel x - c_i \parallel^2 - \parallel s_{ij} - c_i \parallel^2)}{\parallel s_{ij} - c_i \parallel^2}. \tag{13}$$

.

As shown in Algorithm 2, the value of $r_{q_l}(x)$ is first computed by Equation 12 and encoded into m bytes using PQ [11]. 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 $(c_{j_1}^1, \cdots, c_{j_m}^m)$, 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), we assume that c_i is the closest centroid to x and can observe that the anchor point of each

point x is closer to x 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 5.

From Equation 13, the value of λ_{ij} for each point can be computed. it is a `float` type value and requires 4 bytes for each data point. To further improve memory efficiency, we quantize it into 256 values and encode it by a byte. Empirically we find that the encoded λ_{ij} still exhibits high recall rates.

3.3. Query

One important advantage of our indexing structure is that at query time, a specific query point only needs to traverse a small number of cells whose edges are closest to the query point, as shown in the right side of Figure 2. 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, which correspond to $w_1 \cdot n$ second-level regions produced by quantizer q_v. The subregions traversal is performed within only the $w_1 \cdot n$ second-level regions. Moreover, y is quantized again to w_2 nearest second-level regions by quantizer q_l. Then the candidate list of y is formed by the data points only within the w_2 nearest second-level regions. Because the w_2 second-level regions is obviously smaller than the w_1 first-level regions, the candidate list produced by our VLQ-based indexing structure is shorter than that produced by the

VQ-based indexing structure. This will result in a faster query speed.

We use parameter α to determine the percentage of $w_1 \cdot n$ second-level regions to be traversed give a query, such that $w_2 = \alpha \cdot w_1 \cdot n$. We conduct a series of experiments in Section 5 to discuss the performance of our system with different values of α.

3.3.2. Distance computation

Distance computation is a prerequisite condition for re-ranking. In this section, we describe how to compute the approximate distance between a query point y to a candidate point x. According to [11], the distance from y to x can be evaluated by asymmetric distance computation (ADC) as follows:

$$\| y - q_1(x) - q_2(x - q_1(x)) \|^2 \tag{14}$$

where $q_1(x) = (1 - \lambda_{ij}) \cdot c_i + \lambda_{ij} \cdot s_{ij}$ and $q_2(\cdots)$ is the PQ approximation of the x_i displacement.

Expression 14 can be further decomposed as follows [3]:

$$\begin{aligned} \| y - q_1(x) \|^2 + \| q_2(\cdots) \|^2 + 2\langle q_1(x), q_2(\cdots)\rangle - \\ 2\langle y, q_2(\cdots)\rangle. \end{aligned} \tag{15}$$

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

If $l(c_i, s_{ij})$ is the closest edge to x, i.e., $q_1(x) = (1 - \lambda_{ij})c_i + \lambda_{ij}s_{ij}$, Expression 15 can be transformed in the following way:

$$\begin{aligned} &\underbrace{\| y - ((1 - \lambda_{ij})c_i + \lambda_{ij}s_{ij}) \|^2}_{\text{term1}} + \underbrace{\| q_2(\cdots) \|^2}_{\text{term2}} + \\ &2(1 - \lambda_{ij})\underbrace{\langle c_i, q_2(\cdots)\rangle}_{\text{term3}} + 2\lambda_{ij}\underbrace{\langle s_{ij}, q_2(\cdots)\rangle}_{\text{term4}} - \underbrace{2\langle y, q_2(\cdots)\rangle}_{\text{term5}}. \end{aligned} \tag{16}$$

According to Equation 7, term1 in Expression 16 can be computed in the following way:

$$\begin{aligned} \| y-((1-\lambda_{ij})c_i+\lambda_{ij}s_{ij}) \|^2 &= (1-\lambda_{ij}) \underbrace{\| y-c_i \|^2}_{\text{term6}} + \\ (\lambda_{ij}^2-\lambda_{ij}) &\underbrace{\| c_i-s_{ij} \|^2}_{\text{term7}} + \lambda_{ij} \underbrace{\| y-s_{ij} \|^2}_{\text{term8}}. \end{aligned} \tag{17}$$

In Expression 16 and Equation 17, some computations can be done in advance and stored in lookup table as follows:

- All of term2, term3, term4 and term7 are independent of the query. They can be precomputed from the codebooks. Term2 is the squared norm of the displacement approximation and can be stored in a table of size $256 \times m$. Term7 is the square of the length of the edge that the point x belongs to and is already computed in the codebook learning process. Term3 and term4 are scalar products of the PQ sub-centroids and the corresponding first-level centroid subvectors and can be stored in a table of size $k \times 256 \times m$.

- Term6 and term8 are the distances from the query point to the first-layer centroids. They are the by-product of first-layer traversal.

- Term5 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 [6].

The proposed decomposition is used to simplify the distance computation. With the lookup tables, the distance computation only requires $256 \times D$ multiply-adds and $2 \times m$ lookup-adds. In comparison, the classic IVFADC

distance computation requires $256 \times D$ multiply-adds and m lookup-adds [6]. The additional m lookup-adds in our framework improves the distance computation accuracy with a moderate increase of time overhead. We will discuss this trade-off in detail in Section 5.

3.3.3. Re-ranking

Re-ranking is a step of re-sorting the candidate list of data points according to the distances from candidate points to the query point. It is the last step of the query process. The purpose of re-ranking is to find out the nearest neighbours to the query point among the candidate points by distance comparing.We apply the fast sorting algorithm of [6] to our re-ranking step. Due to the shorter candidate list and more accurate approximate distances, the re-ranking step of our system is both faster and more accurate than that of Faiss.

4. GPU Implementation

One advantage of our VLQ-ADC framework is that it is amenable to implementations on GPUs. It is mainly because our searching and distance computing algorithm that applied during query can be efficiently parallelized on GPUs. In this work we have implemented our framework in CUDA.

There are three different levels of granularity of parallelism on GPU: threads, blocks and grids. A block is composed of multiple threads, and a grid is composed of multiple blocks. Furthermore, there are three memory types on GPU. Global memory is typically 4–32 GB in size with 5–10$\times$ higher bandwidth than CPU main memory [6], and can be shared by different blocks. Shared memory is similar to CPU L1 cache in terms of speed and is

Algorithm 3 VLQ-ADC batch search process

function SEARCH($[y_1, \cdots, y_{n_q}], \mathcal{L}_1, \cdots, \mathcal{L}_{k \times n}$)
  for $t \leftarrow 1 : n_q$ **do**
    $C_t \leftarrow w_1\text{-arg}\min_{c \in C} \parallel y_t - c \parallel^2$
    $L_{LQ}^t \leftarrow w_2\text{-arg}\min_{c_i \in C_t, s_{ij} \in S_i} \parallel y - (1 - \lambda_{ij}) \cdot c_i - \lambda_{ij} \cdot s_{ij} \parallel^2$ `// described in Sec. 3.3.1`
    Store values of $\parallel y_t - c \parallel^2$
  end for
  for $t \leftarrow 1 : n_q$ **do**
    $L_t \leftarrow []$
    Compute $\langle y_t, q_2(\cdots) \rangle$ `// See term5 in Equation 16`
    for L in L_{LQ}^t **do**
      for t' in $\mathcal{L}_L$ **do**
        `// distance evaluation described in Sec. 3.3.2`
        $d \leftarrow \parallel y_t - q_1(x_{t'}) - q_2(x_{t'} - q_1(x_{t'})) \parallel^2$
        Append $(d; L; j)$ to L_t
      end for
    end for
  end for
  $R_t \leftarrow$ K-smallest distance-index pairs (d, t') from L_t `// Re-ranking`
  return R_t
end function

only shared by threads within the same block. GPU register file memory has the highest bandwidth and the size of register file memory on GPU is much larger than that on CPU [6].

VLQ-ADC is able to utilize GPU efficiently for indexing and search. For example, we use blocks to process D-dimensional query points and the threads of a block to traverse the inverted lists. We use global memory to store the indexing structures and compressed dataset that is shared by all blocks and grids, and load part of lookup tables in the shared memory to accelerate distance computation. As the GPU register file memory is very large, we store structured data in the register file memory to increase the performance of the sorting algorithm.

Algorithm 3 summarizes our search process that is implemented on GPU. We use four arrays to store the information of the inverted index lists. The first array stores the length of each index list, the second one stores the sorted vector IDs of each list, and the third the fourth store the corresponding codes and λ values of each list respectively. For an NVIDIA GTX Titan X GPU with a 12GB of RAM, we load part of the dataset indexing structure in the global memory for different kernels, i.e., region size, data points compressed codes and λ values of each list. A kernel is the unit of work (instruction stream with arguments) scheduled by the host CPU and executed by GPUs [6]. We load the vector IDs on the CPU side, because vector IDs are resolved only if re-ranking step determines K-nearest membership. This lookup produces a few sparse memory reads in a large array, thus the IDs stored on CPU can only cause a tiny performance cost.

Our implementation makes use of some basic functions from the Faiss library, including matrix multiplication and the K-selection algorithm to im-

prove the performance of our approach[4].

K-selection. The K-selection algorithm is a high-performance GPU-based sorting method proposed by Faiss [6] and GSKNN [8]. The K-selection keep intermediate data in the register file memory. It exchanges register data using the warp shuffle instruction, enabling warp-wide parallelism and storage. The warp is a 32-wide vector of GPU threads, each thread in the warp has up to 255 32-bit registers in a shared register file. All the threads in the same warp can exchange register data using the warp shuffle instruction.

List search. We use two kernels for inverted list search. The first kernel is responsible for quantizing each query point to w_1 nearest first-level regions (line 3 in Algorithm 3). The second kernel is responsible for finding out the w_2 nearest second-level regions for the query point (line 4 in Algorithm 3). The distances between each query point and its w nearest centroids are stored for further calculation. In the two kernels, we use a block of threads to process one query point, thus a batch of n_q query points can be processed concurrently.

Distance computation and re-ranking. After the inverted lists $\mathcal{L}_i$ of each query point are collected, there are up to $n_q \times w_2 \times \max|\mathcal{L}_i|$ candidate points to process. During the distance computation and re-ranking process, processing all the query points in a batch yields high parallelism, but can exceed available GPU global memory. Hence, we choose a tile size $t_q < n_q$ based on amount of available memory to reduce memory overhead, bounding its

[4]The source code will be released upon publication.

complexity by $\mathcal{O}(t_q \times w_2 \times max|\mathcal{L}_i|)$.

We use one kernel to compute the distances from each query point to the candidate points according to Expression 16, and sort the distances via the K-selection algorithm in a separate kernel. The lookup tables are stored in the global memory. In the distance computation kernel, we use a block to scan all w_1 inverted lists for a single query point, and the significant portion of the runtime is the $2 \times w_2 \times m$ lookups in the lookup tables and the linear scanning of the $\mathcal{L}_i$ from global memory.

In the re-ranking kernel, we refer to Faiss by using a two-pass K-selection. First reduce $t_q \times w_2 \times max|\mathcal{L}_i|)$ to $t_q \times \tau \times K$ partial results, where τ is some subdivision factor, then the partial results are reduced again via k-selection to the final $t_q \times K$ results.

Due to the limited amount of GPU's memory, if an index instance with long encoding length cannot fit in the memory of a single GPU, it cannot be processed one the GPU efficiently. Our framework supports multi-GPU parallelism to process indexing instance of a long encoding length. For b GPUs, we split the index instance into b parts, each of which can fit in the memory of a single GPU. We then process the local search of n_q queries on each GPU, and finally join the partial results on one GPU. Our multi-GPU system is based on MPI, which can be easily extended to multiple GPUs on multiple servers.

5. Experiments and Evaluation

In this section, we evaluate the performance of our system VLQ-ADC and compare it to three state-of-the-art billion-scale retrieval systems that are

based on different indexing structures and implemented on CPUs or GPUs: Faiss [6], Ivf-hnsw [7] and Multi-D-ADC [16]. All the systems are evaluated on the standard metrics: accuracy and query time, with different code lengths. All the 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 public benchmark datasets that are commonly used to evaluate billion-scale ANN search: SIFT1B [22] of 10^9 128-D vectors and DEEP1B [5] of 10^9 96-D vectors. Each dataset has a 10,000 query set with the precomputed ground-truth nearest neighbors. For our system, we sample 2×10^6 vectors from each dataset for learning all the trainable parameters. We evaluate the search accuracy by the test result Recall@K, which is the rate of queries for which the nearest neighbors is in the top K results.

Here we choose nprobe =64 for all the inverted indexing systems (Faiss, Ivf-hnsw and VLQ-ADC), as 64 is a typical value for nprobe in the Faiss system. The parameter max_codes that means the max number of candidate data points for a query is only useful to CPU-based system (max_codes is set to 100,000), hence for GPU-based systems like Faiss and VLQ-ADC, max_codes parameter is not configured. In fact, we compute the distances of query point to all the data points that are contained in the neighbor regions.

5.1. Evaluation without re-ranking

In experiment 1, we evaluate the index quality of each retrieval system. We compare three different inverted index structures and two inverted multi-index schemes with different codebooks sizes without the re-ranking step.

1. **Faiss**. We build a codebook of $k = 2^{18}$ centroids by k-means, and find proposed inverted lists of each query by Faiss.
2. **Ivf-hnsw**. We use a codebook of $k = 2^{16}$ centroids by k-means, and set 64 sub-centroids for each first-level centroid.[5]
3. **Multi-D-ADC**. We use two IMI schemes with two codebook sizes $k = 2^{10}$ and $k = 2^{12}$ and choose the implementation from the Faiss library for all the experiments.
4. **VLQ-ADC**. For our approach, we use the same codebook as Ivf-hnsw, and a 64-edge k-NN graph with indexing and querying as described in Section 3.2 and 3.3.[6]

The recall curves of each indexing approach are presented in Figure 3. On both datasets, our proposed system VLQ-ADC (blue curve) outperforms the other two inverted index systems and the Multi-D-ADC scheme with small codebooks ($k = 2^{10}$) for all the reasonable range of X. Compared with the Multi-D-ADC scheme with a larger codebook ($k = 2^{12}$), our system performs better on DEEP1B, and almost equally well on SIFT1B.

On the DEEP1B dataset, the recall rate of our system is consistently higher than that of all the other indexing structures. With a codebook that is only 1/4 the size of Faiss' codebook, the recall rate of our inverted index is higher than Faiss. This demonstrates that the line quantization procedure can further improve the index quality than the previous inverted

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

[6]The VLQ-ADC source code is available at `https://github.com/zjuchenwei/vector-line-quantization`.

index methods.

Even on the SIFT1B dataset, the performance of our indexing structure is almost the same as that of IMI with much larger codebook $k = 10^{12}$ and much better than other inverted index structures.

As shown in Figure 3, for the SIFT1B dataset, the IMI with $k = 2^{12}$ can generate better candidate list than the inverted indexing structures. While for the DEEP1B dataset, the performance of the IMI falls behind that of the inverted indexing structures. The reasons are that SIFT vectors are histogram-based and the subvectors are corresponding to the different subspaces, which describe disjoint image parts that have weak correlations in the subspace distributions. On the other hand, the DEEP vectors are produced by CNN that have a lot of correlations between the subspaces. It can be observed that the performances of our indexing structure is consistent across the two datasets. This demonstrates that our indexing structure's suitability for different data distributions.

5.2. Evaluation with re-ranking

In experiment 2, we evaluate the recall rates with the re-ranking step. In all systems the dataset points are encoded in the same way: indexing and encoding. (1) **Indexing**: displacements from data points to the nearest cell centroids are calculated. For VLQ-ADC the displacements are calculated from data points to the nearest anchor points on the line. (2) **Encoding**: the residual values are encoded into 8 or 16 bytes by PQ with the same codebooks shared by all the cells. Here we compare the same four retrieval systems as in experiments 1. All the configurations for the retrieval systems are the same as in experiment 1. For the GPU-based systems, we evaluate

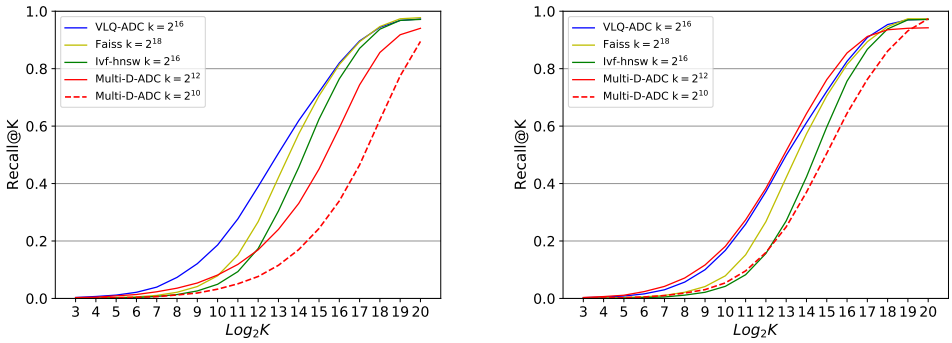


Figure 3: Recall rate comparison of our system, VLQ-ADC, without the re-ranking step, against two inverted index systems, Faiss, Ivf-hnsw, and one inverted multi-index scheme, Multi-D-ADC (with two different codebook sizes: $k = 2^{10}$ and $k = 2^{12}$).

performance with 8-byte codes on 1 GPU and 16-byte codes on 2 GPUs.

The Recall@K values for different values $K = 1/10/100$ and the average query times on both datasets in milliseconds (ms) are presented in Table 3. From Table 3 we can make the following important observations.

Overall best recall performance. Our system VLQ-ADC achieves best recall performance for both datasets and the two codebooks (8-byte and 16-byte) in most cases. For the twelve recall values (Recall@1/10/100 $\times$ two codebooks $\times$ two datasets), VLQ-ADC achieves best values in nine cases and second best in two cases. The second-best system is Faiss, obtaining best results in two cases. Multi-D-ADC (with $k = 2^{12} \times 2^{12}$ regions) obtains best results in one case.

Substantial speedup. VLQ-ADC is consistently and significantly faster than all the other systems in all experiments. For all configurations,

VLQ-ADC's query time is within 0.054–0.068 milliseconds, while the other systems' query times vary greatly. In the most extreme case, VLQ-ADC is 125× faster than Multi-D-ADC (0.068 vs 8.54). At the same time, VLQ-ADC is also consistently faster than the second fastest system, the GPU-based Faiss, by an average 5× speedup.

Comparison with Faiss. VLQ-ADC outperforms the current state-of-the-art GPU-based system Faiss in terms of both accuracy and query time by a large margin, except for only three out of sixteen cases (R@10 with 16-byte codes for SIFT1B, and R@100 with 16-byte codes for SIFT1B and DEEP1B). E.g., as a GPU-based system, VLQ-ADC outperforms Faiss in terms of accuracy by 17%, 14%, 4% of R@1, R@10 and R@100 respectively on the SIFT1B dataset and 8-byte codes. At the same time, the query time is consistently and significantly faster than Faiss, with a speedup of up to 5.7×. Faiss outperforms VLQ-ADC in recall values in three cases, all with 16-byte codes. However, the difference is negligible (∼0.02%). Similarly, though less pronounced, characteristics can be observed on DEEP1B.

The main reason for this improvement is that the index quality and encoding precision in VLQ-ADC is better than those of Faiss. Due to the better indexing quality, the inverted list of our system is much shorter than that of Faiss, which results in a much shorter query time.

Additionally, although the codebook size of our system ($k = 2^{16}$) is only 1/4 of that of Faiss ($k = 2^{18}$), our system produces more regions (2^{22}) than Faiss (2^{18}). Therefore, our system achieves better accuracy

as well as memory and runtime efficiency than Faiss.

Comparison with Multi-D-ADC. The proposed system also outperforms the IMI based system Multi-D-ADC both in terms of accuracy and query time on both datasets. For example, VLQ-ADC leads Multi-D-ADC with codebooks $k = 2^{12}$ by 14.2%, 7.4%, 1.3% of R@1, R@10 and R@100 respectively on the SIFT1B dataset and 8-byte codes with up to 6.8× speedup. On the DEEP1B dataset, the advantage of our system is even more pronounced. Similarly, VLQ-ADC outperforms Multi-D-ADC scheme with smaller codebooks $k = 2^{10}$ even more significantly, especially in terms of query time, where VLQ-ADC consistently achieves speedups of at least one order of magnitude while obtaining better recall values.

Comparison with Ivf-hnsw. Similarly, VLQ-ADC outperforms Ivf-hnsw, another CPU-based retrieval system in both recall and query time. Although Ivf-hnsw can also produce more regions with a small codebook, it still cannot outperform the VQ-based indexing structure with larger size of codebook.

Effects on recall of indexing and encoding. The improvement of R@10 and R@100 shows that the second-level line quantization provides more accurate short-list of candidates than the previous inverted index structure, and the improvement of R@1 shows that it can also improves encoding accuracy.

Multi-D-ADC. From Table 3, we can also observe that Multi-D-ADC scheme with $k = 2^{12}$ outperforms the scheme with $k = 2^{10}$ in query

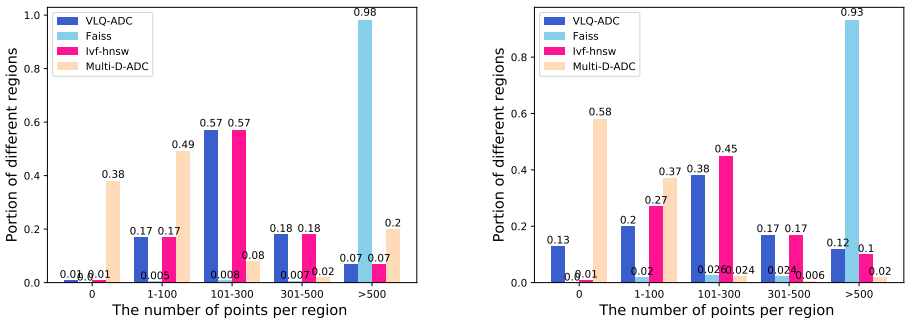
time by a large margin. It is mainly because Multi-D-ADC with larger codebooks can produce more regions, which can extract more concise and accurate short-lists of candidates.

Table 3: Performance comparison between VLQ-ADC (with the re-ranking step) against three other state-of-the-art retrieval systems of recall@1/10/100 and retrieval time on two public datasets. For each system the number of total regions is specified beneath each system's name. VLQ-ADC consistently achieves higher recall values and significantly lower query time than all other systems. Best result in each column is **bolded**, and second best is underlined. For the two GPU-based systems, Faiss and VLQ-ADC, we experiments are performed on 1 GPU for 8-byte encoding length, and on 2 GPUs for 16-byte encoding length.

System	SIFT1B								DEEP1B							
	8 bytes				16 bytes				8 bytes				16 bytes			
	R@1	R@10	R@100	t (ms)	R@1	R@10	R@100	t (ms)	R@1	R@10	R@100	t (ms)	R@1	R@10	R@100	t (ms)
Faiss 2^{18}	0.1383	0.4432	0.7978	0.31	0.3180	0.7825	0.9618	0.280	0.2101	0.4675	0.7438	0.32	0.3793	**0.7650**	**0.9509**	0.33
Ivf-hnsw 2^{16}	0.1599	0.496	0.778	2.35	0.331	0.737	0.8367	2.77	0.217	0.467	0.7195	2.30	0.3646	0.7096	0.828	3.07
Multi-D-ADC $2^{10} \times 2^{10}$	0.1255	0.4191	0.7843	1.65	0.3064	0.7716	**0.9782**	8.54	0.1716	0.3834	0.6527	3.28	0.324	0.6918	0.9258	6.152
Multi-D-ADC $2^{12} \times 2^{12}$	0.1420	0.4720	0.8183	0.367	0.3324	0.8029	0.9752	1.603	0.1874	0.4240	0.6979	0.839	0.3557	0.7087	0.9059	1.52
VLQ-ADC 2^{16}	**0.1620**	**0.507**	**0.829**	**0.054**	**0.345**	**0.8033**	0.9400	**0.068**	**0.2227**	**0.4855**	**0.7559**	**0.059**	**0.394**	0.7644	0.9272	**0.067**

5.3. Data point distributions of different indexing structures

The space and time efficiency of an indexing structure is impacted by the distribution of data points produced by the structure. To analyse the distribution produced by the structures studied in this paper, we plot in Figure 4 the percentages of regions by the discretized number of data points in each region.



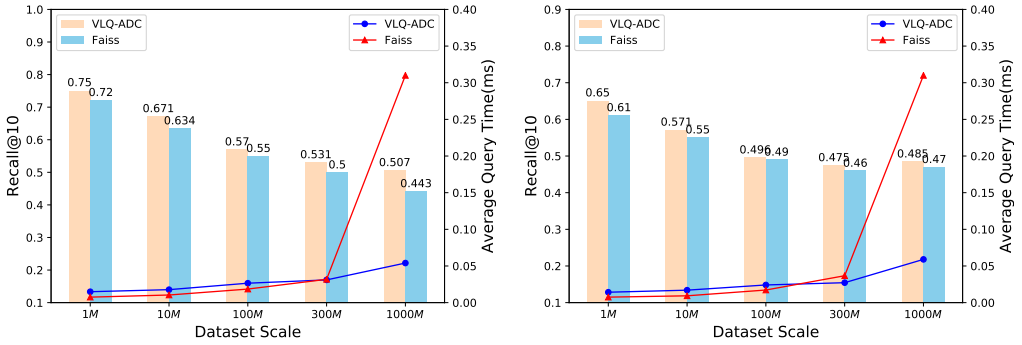
SIFT1B DEEP1B

Figure 4: The distributions of data points in regions produced by different indexing structures. The x axis is five categories representing the discretized numbers of data points in each region (0, 1–100, 101–300, 301–500 and > 500). The y axis is the percentage of regions in each different categories.

As shown in Figure 4, the portion of empty regions produced by the inverted indexing structures (Faiss, Ivf-hnsw and VLQ-ADC) is much less than that produced by the inverted multi-index structure (Multi-D-ADC). For Multi-D-ADC, there are 38% empty regions for SIFT1B and 58% empty regions for DEEP1B (left most group in each plot). This result empirically validates the space inefficiency of inverted multi-index structure [16].

For Faiss, which is based on the inverted indexing structure using VQ, over 98% and 93% of regions contain more than 500 data points for SIFT1B and DEEP1B respectively. This will possibly produce long candidate lists for queries, thus negatively impacting query speed. For VLQ-ADC (and Ivf-hnsw), the regions are much more evenly distributed. The majority of the regions on both datasets contain less than 500 data points, and more regions contain 101–300 data points than others. This is a main reason why

VLQ-ADC can provide shorter candidate lists and thus a faster query speed.



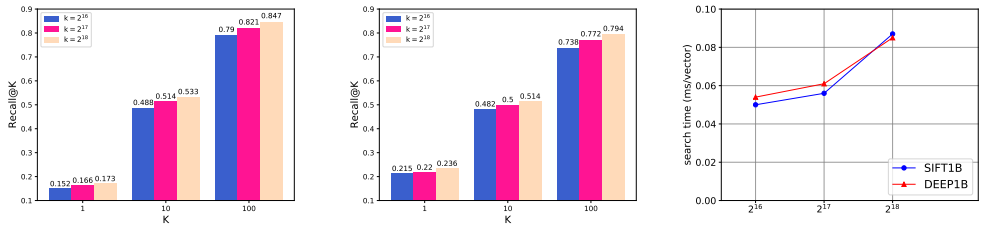
SIFT1B DEEP1B

Figure 5: Comparison of recall@10 and average query time between VLQ-ADC and Faiss under different dataset scales. The two systems are compared with an 8-byte encoding length. The x axis indicates the five data scales (1M/10M/100M/300M/1000M). The left y axis is the recall@10 value (represented by the bars) and the right y axis is the average query time (in ms, represented by the lines).

5.4. Performance comparison under different dataset scales

In this section we evaluate the performance of our system under different dataset scales. Figure 5 shows, for SIFT1B and DEEP1B, the recall and query time values for Faiss and VLQ-ADC for subsets of SIFT and DEEP1B of different sizes: 1M, 10M, 100M, 300M and 1000M (full dataset) respectively. As can be seen in the figure, the recall values of VLQ-ADC is always higher than that of Faiss under all dataset scales. When the scale of dataset is under 300M, the query speed of Faiss is slightly faster than that of VLQ-ADC. When the scale of the datasets is over 300M, the query speed of VLQ-ADC matches that of Faiss.

It can also been observed from the figure that for the full datasets of SIFT1B and DEEP1B (1000M), Faiss takes 0.31ms and 0.32ms respectively (see Table 3 too). Compared to the 100M subsets of these two datasets, Faiss suffers an approx. 15× slowdown when data scale grows 10×. On the other hand, for these two datasets, VLQ-ADC takes 0.054ms and 0.059ms respectively, representing only an approx. 2× slowdown when data scale grows 10×. The superior scalability and robustness of VLQ-ADC over Faiss is evident from this experiment.

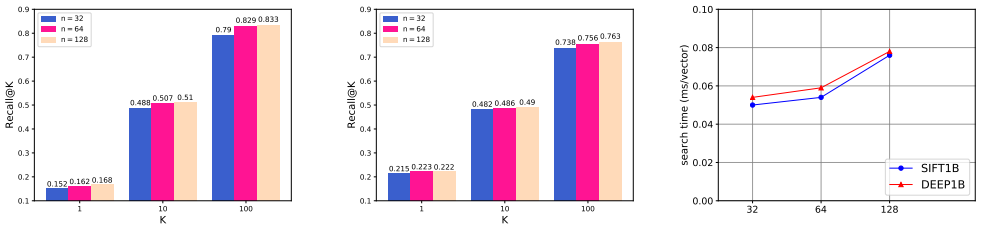


SIFT1B DEEP1B Search time

Figure 6: The performance of VLQ-ADC on different numbers of centroids $k = 2^{16}/2^{17}/2^{18}$. The results are collected on the same two datasets with an 8-byte encoding length and $n = 32$ edges of each centroids. The right plot shows the average search time with different values of k.

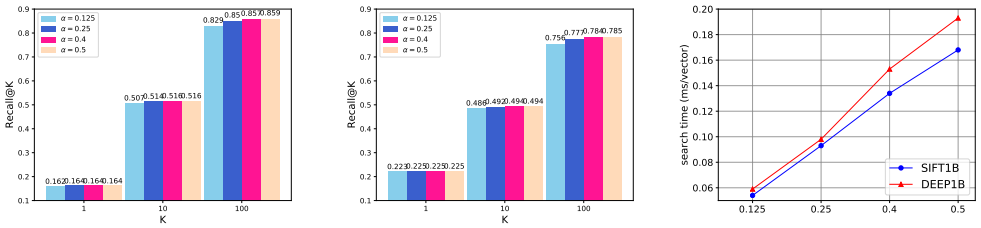
5.5. Evaluation on impact of parameter values

Number of centroids k and edges n. We evaluate the performance of VLQ-ADC on different k and n values with 8-byte codes. We first fix the value of n to 64 and compare the performance of our system for different k centroids. In Figure 6, we present the evaluation of VLQ-ADC for $k = 2^{16}/2^{17}/2^{18}$. Then we fix $k = 2^{16}$ and increase the number of edge n from 32



SIFT1B DEEP1B Search time

Figure 7: The performance of VLQ-ADC on different numbers of graph edges $n = 32/64/128$. The results are collected on the same two datasets with an 8-byte encoding length and $k = 2^{16}$ number of centroids. The right plot shows the average search time with different values of n.



SIFT1B DEEP1B Search time

Figure 8: The performance of VLQ-ADC on different values of parameter $\alpha = 0.25/0.4/0.5$, with values of k, n and w fixed at $k = 2^{16}, n = 64, w = 64$. The result are collected on the same two datasets with an 8-byte encoding length and 64 edges of each centroids. The right plot shows the average search time with different values of α.

to 64 and 128. In Figure 7, we present the evaluation of the VLQ-ADC for different edge numbers.

From Figure 6 and 7 we can observe that the increase in the number of centroids and edges can improve search accuracy, while slightly increasing query time. This is because the indexing scheme with more centroids

and more edges can represent the dataset points more accurately and hence provide more accurate short inverted lists.

Value of portion α. Now, we discuss how to determine the value of parameter α for subregions pruning, as described in Section 3.3.1. As shown in Figure 8, we test several values of α on both datasets. A lower α value means fewer subregions will be traversed, hence lower query time. At the same time, we can observe that higher α values only moderately increase recall values, while significantly increases query time (up to 3.7× times). Hence we choose $\alpha = 0.25$.

Time and memory consumption. Because the billion-scale dataset do not fit on the GPU, the database is built in batches of 2M vectors, then aggregating the information on the CPU. With file I/O, it takes about 150 minutes to build the whole database on a single GPU.

Here we analyze the memory consumption of each system. As shown in Table 2, for a database of $N = 10^9$ points, the basic memory consumption for all systems is $4 \cdot N$ bytes for point IDs that are Integer type and $m \cdot N$ bytes for point codes. In addition to that, Multi-D-ADC consumes $4 \cdot k^2$ bytes to store the region boundaries. Faiss consumes $4 \cdot k \cdot D$ bytes for the codebooks and $4 \cdot k \cdot m \cdot 256$ bytes for the lookup tables. Ivf-hnsw requires N bytes for quantized norm items $4 \cdot k \cdot (D + n)$ bytes for its indexing structure[7]. For our system, we require N bytes for quantized λ values and $4 \cdot k \cdot (D + 2n + m \cdot 256)$ bytes for the codebook, the n-NN graph and the lookup tables. We summarize the total memory consumption for all systems in Table 4 with 8-byte encoding length on both datasets.

As presented in Table 4, the memory consumption of our system is less

than that of Faiss, and about 10% more than that of Multi-D-ADC with 2^{12} codebook, which is acceptable for most realistic setups.

Table 4: The memory consumption of all systems for SIFT1B of 10^9 128-dimensional data points.

System (codebook size)	Memory consumption (GB)
Faiss (2^{18})	14
Ivf-hnsw (2^{16})	13.04
Multi-D-ADC ($2^{12} \times 2^{12}$)	12.25
VLQ-ADC (2^{16})	13.55

6. Conclusion

Billion-scale approximate nearest neighbor (ANN) search has become an important task as massive amounts of visual data becomes available online. In this work, we proposed VLQ-ADC, a simple yet scalable indexing structure and a retrieval system that is capable of handling billion-scale datasets. VLQ-ADC combines line quantization with vector quantization to create a hierarchical indexing structure. Search space is further pruned to a portion of the closest regions, further improving ANN search performance. The proposed indexing structure can partition the billion-scale database in large number of regions with a moderate size of codebook, which solved the drawback of prior VQ-based indexing structures.

We performed comprehensive evaluation on two billion-scale benchmark datasets: SIFT1B and DEEP1B and three state-of-the-art ANN search sys-

tems: Multi-D-ADC, Ivf-hnsw, and Faiss. Our evaluation shows that VLQ-ADC consistently outperforms all three systems on both recall and query time. VLQ-ADC achieves a recall improvement over Faiss, the state-of-the-art GPU-based system, of up to 17% and a query time speedup of up to 5× times.

Moreover, VLQ-ADC takes the data distribution into account in the indexing structure. As a result, it performs well on datasets with different distributions. Our evaluation shows that VLQ-ADC is the best performer on both SIFT1B and DEEP1B, demonstrating its robustness with respect to data with different distributions.

We conclude by pointing out a number of future work directions. We plan to investigate further improvements to the indexing structure. Moreover, a more systematic and principled method for hyperparameter selection is worthy investigation.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China under Grant No.61672246, No.61272068, No.61672254 and the Fundamental Research Funds for the Central Universities under Grant HUST:2016YXMS018. In addition, we gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPUs used for this research. The authors appreciate the valuable suggestions from the anonymous reviewers and the Editors.

References

[1] T. H. H. Chan, A. Guerqin, M. Sozio, Fully dynamic k-center clustering, in: World Wide Web Conference, 2018, pp. 579–587.

[2] R. Weber, H.-J. Schek, S. Blott, A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces, in: VLDB, volume 98, 1998, pp. 194–205.

[3] A. Babenko, V. Lempitsky, Improving bilayer product quantization for billion-scale approximate nearest neighbors in high dimensions, arXiv preprint arXiv:1404.1831 (2014).

[4] P. Wieschollek, O. Wang, A. Sorkine-Hornung, H. Lensch, Efficient large-scale approximate nearest neighbor search on the GPU, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 2027–2035.

[5] A. B. Yandex, V. Lempitsky, Efficient indexing of billion-scale datasets of deep descriptors, in: Computer Vision and Pattern Recognition, 2016, pp. 2055–2063.

[6] J. Johnson, M. Douze, H. Jégou, Billion-scale similarity search with GPUs, arXiv preprint arXiv:1702.08734 (2017).

[7] D. Baranchuk, A. Babenko, Y. Malkov, Revisiting the inverted indices for billion-scale approximate nearest neighbors, in: Proceedings of the European Conference on Computer Vision (ECCV), 2018, pp. 202–216.

[8] C. D. Yu, J. Huang, W. Austin, B. Xiao, G. Biros, Performance optimization for the k-nearest neighbors kernel on x86 architectures, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 7.

[9] Y. Gong, S. Lazebnik, A. Gordo, F. Perronnin, Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval, IEEE Transactions on Pattern Analysis and Machine Intelligence 35 (2013) 2916–2929.

[10] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

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

[12] Y. Kalantidis, Y. Avrithis, Locally optimized product quantization for approximate nearest neighbor search, in: Computer Vision and Pattern Recognition, 2014, pp. 2329–2336.

[13] M. Muja, D. G. Lowe, Scalable nearest neighbor algorithms for high dimensional data, IEEE Transactions on Pattern Analysis and Machine Intelligence 36 (2014) 2227–2240.

[14] M. Norouzi, D. J. Fleet, Cartesian k-means, in: IEEE Conference on Computer Vision and Pattern Recognition, 2013, pp. 3017–3024.

[15] Y. Linde, A. Buzo, R. Gray, An algorithm for vector quantizer design, IEEE Transactions on communications 28 (1980) 84–95.

[16] A. Babenko, V. Lempitsky, The inverted multi-index, in: Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on, IEEE, 2012, pp. 3069–3076.

[17] A. S. Razavian, H. Azizpour, J. Sullivan, S. Carlsson, CNN features off-the-shelf: An astounding baseline for recognition, in: 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2014, pp. 512–519.

[18] Y. Gong, L. Wang, R. Guo, S. Lazebnik, Multi-scale orderless pooling of deep convolutional activation features, in: European conference on computer vision, Springer, 2014, pp. 392–407.

[19] S. Lloyd, Least squares quantization in pcm, IEEE transactions on information theory 28 (1982) 129–137.

[20] T. Ge, K. He, Q. Ke, J. Sun, Optimized product quantization for approximate nearest neighbor search, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2013, pp. 2946–2953.

[21] J. Sivic, A. Zisserman, Video Google: a text retrieval approach to object matching in videos, in: Proceedings Ninth IEEE International Conference on Computer Vision, 2003, pp. 1470–1477.

[22] H. Jégou, R. Tavenard, M. Douze, L. Amsaleg, Searching in one billion vectors: re-rank with source coding, in: Acoustics, Speech and Signal

785 Processing (ICASSP), 2011 IEEE International Conference on, IEEE,
786 2011, pp. 861–864.