Spherical Hashing: Binary Code Embedding with Hyperspheres

Jae-Pil Heo, *Member, IEEE*, Youngwoon Lee, *Member, IEEE*, Junfeng He, *Member, IEEE*, Shih-Fu Chang, *Fellow, IEEE*, Sung-Eui Yoon, *Senior Member, IEEE*

Abstract—Many binary code embedding schemes have been actively studied recently, since they can provide efficient similarity search, and compact data representations suitable for handling large scale image databases. Existing binary code embedding techniques encode high-dimensional data by using hyperplane-based hashing functions. In this paper we propose a novel hypersphere-based hashing function, *spherical hashing*, to map more spatially coherent data points into a binary code compared to hyperplane-based hashing functions. We also propose a new binary code distance function, *spherical Hamming distance*, tailored for our hypersphere-based binary coding scheme, and design an efficient iterative optimization process to achieve both balanced partitioning for each hash function and independence between hashing functions. Furthermore, we generalize spherical hashing to support various similarity measures defined by kernel functions. Our extensive experiments show that our spherical hashing technique significantly outperforms state-of-the-art techniques based on hyperplanes across various benchmarks with sizes ranging from one to 75 million of GIST, BoW and VLAD descriptors. The performance gains are consistent and large, up to 100% improvements over the second best method among tested methods. These results confirm the unique merits of using hyperspheres to encode proximity regions in high-dimensional spaces. Finally, our method is intuitive and easy to implement.

Index Terms—hashing, binary codes, large-scale image search

1 Introduction

Thanks to rapid advances of digital camera and various image processing tools, we can easily create new pictures and images for various purposes. This in turn results in a huge amount of images available online. These huge image databases pose a significant challenge in terms of scalability to many computer vision applications, especially those applications that require efficient similarity search.

For similarity search, nearest neighbor search techniques have been widely studied and tree-based techniques [2], [3], [4], [5] have been used for low-dimensional data. Unfortunately, these techniques are not scalable to high-dimensional data. Hence recently binary code embedding techniques have been actively studied to provide efficient solutions for such high-dimensional data [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23].

Encoding high-dimensional data points into binary codes based on hashing techniques enables higher scalability thanks to both its compact data representation and efficient indexing mechanism. Similar high-dimensional data points are mapped to similar binary codes and thus by looking into only those similar binary codes (based on the Hamming distance), we can efficiently identify approximate nearest neighbors.

Existing hashing techniques can be broadly categorized as data-independent and data-dependent schemes. In data-independent techniques, hashing functions are chosen independently from the data points. Locality-Sensitive Hashing (LSH) [6] is one of the most widely known techniques in this category. This technique is extended to various hashing functions [7], [8], [11], [12], [13]. Recent research attentions have been shifted to developing data-dependent techniques to consider the distribution of data points and design better hashing functions. Notable examples include spectral hashing [10], semi-supervised hashing [17], iterative quantization [20], joint optimization [21], and random maximum margin hashing [22].

In all of these existing hashing techniques, hyperplanes are used to partition the data points into two sets and assign two different binary codes (e.g., -1 or $+1$) depending on which set each point is assigned to. Departing from this conventional approach, we propose a novel hypersphere-based scheme, *spherical hashing*, for computing binary codes. Intuitively, hyperspheres provide much stronger power in defining a tighter closed region than hyperplanes (See Fig. 1). For example, at least $d + 1$ hyperplanes are needed to define a closed region for a d-dimensional space, while only a single hypersphere can form such a

- *Jae-Pil Heo, Youngwoon Lee, and Sung-Eui Yoon are with the Department of Computer Science, KAIST (Korea Advanced Institute of Science and Technology), South Korea.*
  Sung-Eui Yoon is a corresponding author of the paper
  E-mail: jaepilheo, lywoon89, sungeui@gmail.com
  Code is available at: http://sglab.kaist.ac.kr/Spherical_Hashing
- *Junfeng He is with Facebook.*
  E-mail: hejunf@gmail.com
- *Shih-Fu Chang is with the Department of Electrical Engineering, Columbia University, USA.*
  E-mail: sfchang@ee.columbia.edu

- *This manuscript is extended from the conference paper version [1]*

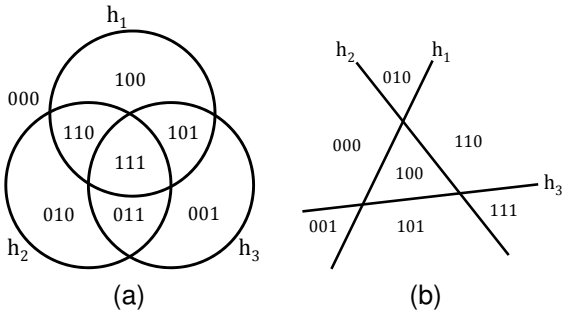


Fig. 1. The difference between our hypersphere-based binary code embedding method and hyperplane-based one. The left and right figures show partitioning examples of hypersphere-based and hyperplane-based methods respectively, for 3 bit binary codes in the 2-D space. Each function h_i determines the value of i-th bit of binary codes. The hypersphere-based binary code embedding scheme gives a higher number of tightly closed regions compared to hyperplane-based one.

closed region even in an arbitrarily high dimensional space.

Our paper has the following contributions:

1) We propose a novel spherical hashing scheme, analyze its ability in terms of similarity search, and compare it against the state-of-the-art hyperplane-based techniques (Sec. 3.1).
2) We develop a new binary code distance function tailored for the spherical hashing method (Sec. 3.2).
3) We formulate an optimization problem that achieves both balanced partitioning for each hashing function and the independence between any two hashing functions (Sec. 3.3). Also, an efficient, iterative process is proposed to construct spherical hashing functions (Sec. 3.4).
4) We design an adaptive scheme to set the distance threshold of the hyperspheres by a maximal margin principle (Sec. 3.5).
5) We generalize spherical hashing to support arbitrary kernel functions, and reformulate the optimization process into a kernelized one (Sec. 4).

In order to highlight benefits of our method, we have tested our method against different benchmarks that consist of one to 75 million image descriptors with varying dimensions. We have also compared our method with many state-of-the-art techniques and found that our method significantly outperforms all the tested techniques, confirming the superior ability of defining closed regions with tighter bounds compared to conventional hyperplane-based hashing functions (Sec. 5).

2 Related Work

In this section we discuss prior work related to nearest neighbor search techniques.

2.1 Hierarchical Methods

Space partitioning based tree structures such as kd-trees [2], [3], R-trees [4], Vantage Point Trees (VPT) [24] have been used to find nearest neighbors. Excellent surveys for such tree-based indexing and nearest neighbor search methods are available [25], [26]. Using kd-trees is one of the most popular approaches, and thus there have been a lot of optimization efforts such as randomized kd-trees [27], relaxed orthogonality of partitioning axes [28], and minimizing probabilistic search cost [29]. It has been widely known, however, that kd-tree based search can run slower even than the linear scan for high dimensional data. Nistér and Stewénius [30] proposed another tree-based nearest neighbor search scheme based on hierarchical k-means trees. Muja and Lowe [5] have proposed an automatic parameter selection algorithm of some of techniques mentioned in above.

Although these techniques achieve reasonably high accuracy and efficiency, they have been demonstrated in small image databases consisting of about one million images. Also, these techniques do not consider compressions of image descriptors to handle large-scale image databases.

2.2 Binary Code Embedding Methods

Binary code embedding methods that embed high dimensional points to compact binary codes have been actively studied recently, since they provide both high compression efficiency and fast similarity computation. Binary code embedding methods aim to embed points in binary codes, while preserving relative distances among them. Most methods compute a binary value using a hash function that preserves distance among data points. Distances among the data points are then approximated by similarity among their binary codes such as Hamming distance.

Binary code embedding methods can be broadly categorized as data-independent and data-dependent schemes. In data-independent methods, the hash functions are defined independently from the data. One of the most popular hashing techniques in this category is Locality Sensitive Hashing (LSH) [6]. Its hash function is based on projection onto random vectors drawn from a specific distribution. Many variations and extensions of LSH have been proposed for L_p norms [8], learned metrics [12], min-hash [11], inner products [7], and multi-probe [31]. Kulis et al. [32] generalized LSH to Kernelized LSH that supports arbitrary kernel functions defining similarity. Raginsky and Lazebnik [13] have proposed a binary code embedding scheme based on random Fourier features for shift-invariant kernels.

There have been a number of research efforts to develop data-dependent hashing methods that reflect data distributions to improve the performance. Weiss et al. [10] have proposed spectral hashing motivated

by spectral graph partitioning. Liu et al. [33] applied the graph Laplacian technique by interpreting a nearest neighbor structure as an anchor graph. Strecha et al. [34] used Linear Discriminant Analysis (LDA) for binarization of image descriptors. Wang et al. [17] proposed a semi-supervised hashing method to improve image retrieval performance by exploiting label information of the training set. Gong and Lazebnik [20] introduced a procrustean approach that directly minimizes quantization error by rotating zero-centered PCA-projected data. He et al. [21] presented a hashing method that jointly optimizes both search accuracy and search time by incorporating a similarity preserving term into the Independent Component Analysis (ICA). Joly and Buisson [22] constructed hash functions by using large margin classifiers such as the support vector machine (SVM) with arbitrarily sampled data points that are randomly separated into two sets. In most cases, data-dependent methods outperform data-independent ones with short binary codes.

The efficiency of each hash function in data-dependent methods is, however, getting lower as they allocate longer binary codes. The main cause of this trend is the growing difficulty of defining independent and informative set of projections as the number of hash functions increases. To avoid the issue there have been a few approaches that use a single hash function to determine multiple bits and use less hash functions [33], [35], [36], [37].

All the mentioned techniques compute binary codes by partitioning data points into two different sets based on hyperplanes. Departing from this conventional approach, we adopt a novel approach of partitioning data points by hyperspheres.

There is another category of compact data representation techniques based on the quantization. Jégou et al. have proposed Product Quantization (PQ) [38], which decomposes a high-dimensional space into multiple lower-dimensional spaces and constructs k-means clusters in each subspace separately. They encode a high dimensional data as a concatenation of cluster indices over the subspaces. In [39], PQ is further improved by the dimension reduction and balancing the variance of components.

3 Spherical Hashing

Let us first define notations. Given a set of N data points in a D-dimensional space, we use $X = \{x_1, ..., x_N\}$, $x_i \in \mathbb{R}^D$ to denote those data points. A binary code corresponding to each data point x_i is defined by $b_i = \{-1, +1\}^l$, where l is the length of the code[1].

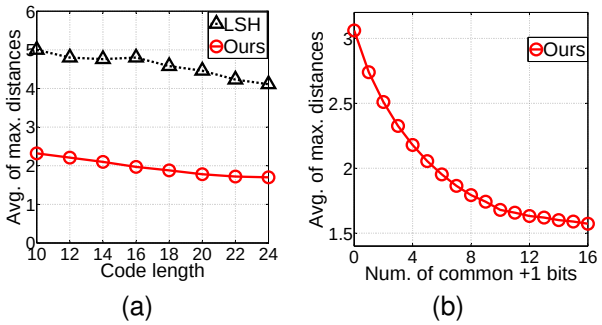


Fig. 2. The left figure shows how the avg. of the max. distances among points having the same binary code changes with different code lengths based on hyperspheres or hyperplanes. We randomly sample 1000 different binary codes to compute avg. of the max. distances. The right figure shows how having more common $+1$ bits in our method effectively forms tighter closed regions. For the right curve we randomly sample one million pairs of binary codes. For each pair of binary codes (b_i, b_j) we compute the max. distance between pairs of points, (x_i, x_j), where $H(x_i) = b_i$ and $H(x_j) = b_j$. We report the avg. of the max. distances as a function of the number of common $+1$ bits, i.e. $|b_i \wedge b_j|$. Both figures are obtained with GIST-1M-960D dataset (Sec. 5.1).

3.1 Binary Code Embedding Function

Our binary code embedding function $H(x) = (h_1(x), ..., h_l(x))$ maps points in $\mathbb{R}^D$ into the binary cube $\{-1, +1\}^l$. We use a hypersphere to define a spherical hashing function. Each spherical hashing function $h_i(x)$ is defined by a pivot $p_i \in \mathbb{R}^D$ and a distance threshold $t_i \in \mathbb{R}^+$ as the following:

$$h_i(x) = \begin{cases} -1 & \text{when } d(p_i, x) > t_i \\ +1 & \text{when } d(p_i, x) \leq t_i, \end{cases} \quad (1)$$

where $d(\cdot, \cdot)$ denotes the Euclidean distance between two points in $\mathbb{R}^D$; various distance metrics (e.g., L_p metrics) can be used instead of the Euclidean distance. The value of each spherical hashing function $h_i(x)$ indicates whether the point x is inside the hypersphere whose center is p_i and radius is t_i. Fig. 1(a) shows an example of a space partitioning and assigned binary codes with three hyperspheres in 2-D space.

The key difference between using hyperplanes and hyperspheres for computing binary codes is their abilities to define a closed region in $\mathbb{R}^D$ that can be indexed by a binary code. To define a closed region in a d-dimensional space, at least $d + 1$ hyperplanes are needed, while only a single hypersphere is sufficient to form such a closed region in an arbitrarily high dimensional space. Furthermore, unlike using multiple hyperplanes a higher number of closed regions

1. $(-1, +1)^*$ codes are conceptual expression. Codes are stored and processed as $(0, 1)^*$ codes in practice.

can be constructed by using multiple hyperspheres, while the distances between points located in each region are bounded. For example, the number of bounded regions by having l hyperspheres goes up to $\binom{l-1}{d} + \sum_{i=0}^{d} \binom{l}{i}$ [40]. In addition, we can approximate a hyperplane with a large hypersphere that has a large radius and a far-away center.

In nearest neighbor search the capability of forming closed regions with tighter distance bounds is very important in terms of effectively locating nearest neighbors from a query point. When we construct such tighter closed regions, a region indexed by the binary code of the query point can contain more promising candidates for the nearest neighbors.

We also empirically measure how tightly hyperspheres and hyperplanes bound regions. For this purpose, we measure the maximum distance between any two points that have the same binary code and take the average of the maximum distances among different binary codes. As can be seen in Fig. 2(a), hyperspheres bound regions of binary codes more tightly compared to hyperplanes used in LSH [8]. Across all the tested code lengths, hyperspheres show about two times tighter bounds over the hyperplane-based approach.

3.2 Distance between Binary Codes

Most hyperplane-based binary code embedding methods use the Hamming distance between two binary codes, which measures the number of different bits, i.e. $|b_i \oplus b_j|$, where $\oplus$ is the XOR bit operation and $|\cdot|$ denotes the number of $+1$ bit in a given binary code. This distance metric measures the number of hyperplanes that two given points reside in the opposing side of them. The Hamming distance, however, does not well reflect the property related to defining closed regions with tighter bounds, which is the core benefit of using our spherical hashing functions.

To fully utilize desirable properties of our spherical hashing function, we propose the following distance metric, *spherical Hamming distance* (SHD) ($d_{SHD}(b_i, b_j)$), between two binary codes b_i and b_j computed by spherical hashing:

$$d_{SHD}(b_i, b_j) = \frac{|b_i \oplus b_j|}{|b_i \wedge b_j|}, \tag{2}$$

where $|b_i \wedge b_j|$ denotes the number of common $+1$ bits between two binary codes which can be easily computed with the AND bit operations.

Having the common $+1$ bits in two binary codes gives us tighter bound information than having the common -1 bits in our spherical hashing functions. This is mainly because each common $+1$ bit indicates that two data points are inside its corresponding hypersphere, giving a stronger cue in terms of distance bounds of those two data points; see Fig. 3 for intuition. In order to see the relationship between the distance bound and the number of the common $+1$ bits, we measure the average distance bounds of data points as a function of the number of the common $+1$ bits. As can be seen in Fig. 2(b), the average distance bound decreases as the number of the common $+1$ bits in two binary codes increases. As a result, we put $|b_i \wedge b_j|$ in the denominator of our spherical Hamming distance.

1000-NN mAP with GIST-1M-960D

# bits	32	64	128	256
RMMH-SHD	0.0279	0.0603	0.0976	0.1466
RMMH-HD	0.0266	0.0576	0.0993	0.1483
ITQ-SHD	0.0385	0.0578	0.0860	0.1060
ITQ-HD	0.0380	0.0620	0.0875	0.1101

TABLE 1
Experimental results of hyperplane based methods combined with SHD.

In implementation we add a small value (e.g. 0.1) to the denominator to avoid the division-by-zero. Also, we can construct a pre-computed SHD table $T(|b_i \wedge b_j|, |b_i \oplus b_j|)$ whose size is $(l+1)^2$ and refer the table, when computing SHD to avoid expensive division operations.

The common $+1$ bits between two binary codes define a closed region with a distance bound as mentioned above. Within this closed region we can further differentiate the distance between two binary codes based on the Hamming distance $|b_i \oplus b_j|$, the numerator of our distance function. The numerator affects our distance function in the same manner to the Hamming distance, since the distance between two binary codes increases as we have more different bits between two binary codes.

In hyperplane based methods, the common $+1$ bits do not give strong cue on estimating the real distance. As a resulot, SHD does not provide any benefit for hyperplane based methods as reported in Table. 1.

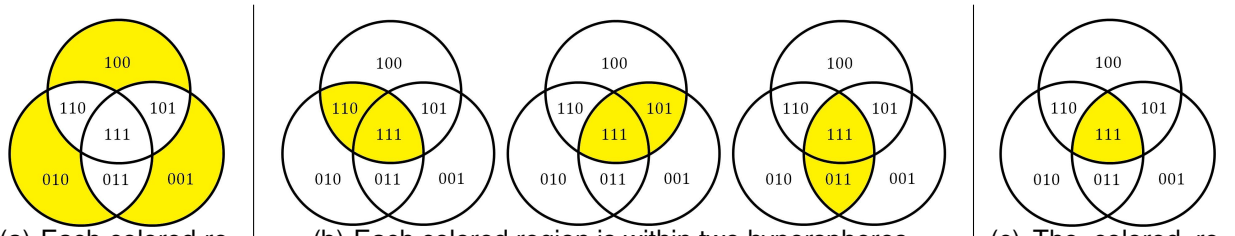
An alternative definition of SHD can be constructed based on the subtraction as following:

$$d_{SHD-SUB}(b_i, b_j) = |b_i \oplus b_j| - |b_i \wedge b_j|. \tag{3}$$

SHD-SUB is intuitive and free from the division by zero. However, the estimated distance is linearly decreasing with respect to $|b_i \wedge b_j|$ and thus a little bit different from our observation in Fig. 2(b). We also provide experimental comparison between SHD and SHD-SUB in Table. 2. Since SHD provides slightly better performance compared to SHD-SUB, we have used SHD in all the experiments in this paper instead of SHD-SUB.

3.3 Independence between Hashing Functions

Achieving balanced partitioning of data points for each hashing function and the independence between hashing functions has been known to be important [10], [21], [22], since independent hashing



(a) Each colored region is within one hypersphere.

(b) Each colored region is within two hyperspheres.

(c) The colored region is within three hyperspheres.

Fig. 3. Intuition of the spherical hamming distance. If both two points x_i and x_j are located in one of colored regions then their binary codes b_i and b_j have at least 1, 2, or 3 common +1 bits, respectively. As the number of common +1 bits of b_i and b_j increases, the size of the region containing both points x_i and x_j are expected to become smaller. As a result, the expected distance between x_i and x_j also gets smaller.

100-NN mAP with GIST-1M-384D

# bits	32	64	128	256	512
SHD	0.0153	0.0426	0.981	0.1760	0.2572
SHD-SUB	0.0139	0.0398	0.0931	0.1656	0.2402

TABLE 2
Comparisons between SHD and SHD-SUB described in Sec. 3.2

functions distribute points in a balanced manner to different binary codes. It has been known that achieving such properties lead to minimizing the search time [21] and improving the accuracy even for longer code lengths [22]. We also aim to achieve this independence between our spherical hashing functions.

We define each hashing function h_i to have the equal probability for $+1$ and -1 bits respectively as the following:

$$Pr[\ h_i(x) = +1\ ] = \frac{1}{2}, \quad x \in X, \quad 1 \leq i \leq l \qquad (4)$$

Let us define a probabilistic event V_i to represent the case of $h_i(x) = +1$. Two events V_i and V_j are independent if and only if $Pr[V_i \cap V_j] = Pr[V_i] \cdot Pr[V_j]$. Once we achieve balanced partitioning of data points for each bit (Eq. 4), then the independence between two bits can satisfy the following equation given $x \in X$ and $1 \leq i < j \leq l$:

$$\begin{aligned} &Pr[h_i(x) = +1,\ h_j(x) = +1] \\ &\quad = Pr[h_i(x) = +1] \cdot Pr[h_j(x) = +1] = \tfrac{1}{2} \cdot \tfrac{1}{2} = \tfrac{1}{4} \end{aligned} \qquad (5)$$

In general the pair-wise independence between hashing functions does not guarantee the higher-order independence among three or more hashing functions. We can also formulate the independences among more than two hashing functions and aim to satisfy them in addition to constraints shown in Eq. 4 and Eq. 5. However we found that considering such higher-order independence hardly improves the search quality.

3.4 Iterative Optimization

We now propose an iterative process for computing l different hyperspheres, i.e. their pivots p_i and distance thresholds t_i. During this iterative process we construct hyperspheres to satisfy constraints shown in Eq. 4 and Eq. 5.

As the first phase of our iterative process, we sample a subset $S = \{s_1, s_2, ..., s_n\}$ from data points X to approximate its distribution. We then initialize the pivots of l hyperspheres with randomly chosen l data points in the subset S; we found that other alternatives of initializing the pivots (e.g., using center points of K-means clustering performed on the subset S) do not affect the results of our optimization process. However, we observe that the optimization process converges slightly quicker, when initial pivots are closely located in the center of the training points. This is mainly because by locating hyperspheres closely to each other, we can initialize hyperspheres to have overlaps. For this acceleration, we set the pivot position of a hypersphere to be the median of randomly chosen multiple samples, i.e. $p_i = \frac{1}{g}\sum_{j=1}^{g} q_j$, where q_j are randomly selected points from S and g is the number of such points. Too small g does not locate pivots closely to the data center, and too large g locates pivots to be in almost similar positions. In practice, $g = 10$ provides a reasonable acceleration rate, given its trade-off space.

As the second phase of our iterative process, we refine pivots of hyperspheres and compute their distance thresholds. To help these computations, we compute the following two variables, o_i and $o_{i,j}$, given $1 \leq i, j \leq l$:

$$\begin{aligned} o_i &= |\ \{s_g | h_i(s_g) = +1, 1 \leq g \leq n\}\ |, \\ o_{i,j} &= |\ \{s_g | h_i(s_g) = +1, h_j(s_g) = +1, 1 \leq g \leq n\}\ |, \end{aligned} \qquad (6)$$

where $|\cdot|$ is the cardinality of the given set. o_i measures how many data points in the subset S have $+1$ bit for i-th hashing function and will be used to satisfy balanced partitioning for each bit (Eq. 4). Also, $o_{i,j}$ measures the number of data points in the subset S

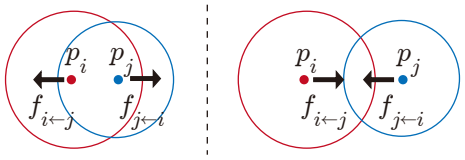


Fig. 4. These two images show how a force between two pivots is computed. In the left image a repulsive force is computed since their overlap $o_{i,j}$ is larger than the desired amount. On the other hand, the attractive force is computed in the right image because their overlap is smaller than the desired amount.

that are contained within both of two hyperspheres corresponding to i-th and j-th hashing functions. $o_{i,j}$ will be used to satisfy the independence between i-th and j-th hashing functions during our iterative optimization process.

Once we compute these two variables with data points in the subset of S, we adopt two alternating steps to refine pivots and distance thresholds for hyperspheres in order to meet our optimization goal:

$$o_i = \frac{n}{2} \text{ and } o_{i,j} = \frac{n}{4}. \tag{7}$$

First, we adjust the pivot positions of two hyperspheres in a way that $o_{i,j}$ becomes closer to or equal to $\frac{n}{4}$. Intuitively, for each pair of two hyperspheres i and j, when $o_{i,j}$ is greater than $\frac{n}{4}$, a repulsive force is applied to both pivots of those two hyperspheres (i.e. p_i and p_j) to place them farther away. Otherwise an attractive force is applied to locate them closer. Second, once pivots are computed, we adjust the distance threshold t_i of ith hypersphere such that o_i becomes $\frac{n}{2}$ to meet balanced partitioning of the data points for the hypersphere (Eq. 4).

We perform our iterative process until the computed hyperspheres do not make further improvements in terms of satisfying constraints. Specifically, we consider the sample mean and standard deviation of $o_{i,j}$ as a measure of the convergence of our iterative process. Ideal values for the mean and standard deviation of $o_{i,j}$ are $\frac{n}{4}$ and zero respectively. However, in order to avoid over-fitting, we stop our iterative process when the mean and standard deviation of $o_{i,j}$ are within ϵ_m% and ϵ_s%, error tolerances, of the ideal mean of $o_{i,j}$ respectively.

For these parameters, we conducted the following experimental tests to find suitable values. We compute mean Average Precisions (mAPs) of k-nearest neighbor search with various experiment settings, and they are shown in Fig. 5. According to the experimental results, we pick ϵ_m and ϵ_s that provide the empirical maximum. Based on these experimental tests, we have chosen (ϵ_m=10%, ϵ_s=15%) for GIST-1M-384D, GIST-1M-960D, and 1000 dimensional BoW descriptors. We have, however, found that we need stricter termination conditions of the optimization process for higher

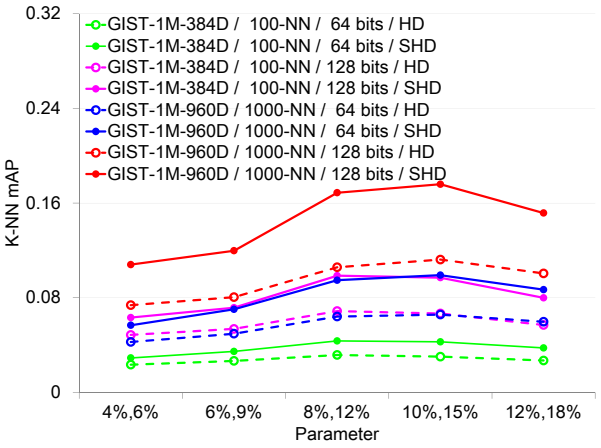


Fig. 5. mAP curves for k-nearest neighbor search with respect to various parameters. A pair of values in x-axis are used for two parameters of ϵ_m and ϵ_s, and y-axis represents their corresponding mAP values. Each legend consists of four experiment settings 'dataset / k / binary code length / distance metric type (HD: Hamming distance, SHD: spherical Hamming distance)'.

dimensional data. The convergence rate of the objective functions is much faster in higher dimensional space, since we have more degrees of freedom of pivot positions, and this can cause an undesired underfitting. We have therefore chosen (ϵ_m=4%, ϵ_s=6%) for 8192 dimensional VLAD descriptors (Sec. 5.1).

Force computation: A (repulsive or attractive) force from p_j to p_i, $f_{i\leftarrow j}$, is defined as the following (Fig. 4):

$$f_{i\leftarrow j} = \frac{1}{2}\frac{o_{i,j} - n/4}{n/4}(p_i - p_j). \tag{8}$$

An accumulated force, f_i, is then the average of all the forces computed from all the other pivots as the following:

$$f_i = \frac{1}{l}\sum_{j=1}^{l} f_{i\leftarrow j}. \tag{9}$$

Once we apply the accumulated force f_i to p_i, then p_i is updated simply as $p_i + f_i$. Our iterative optimization process is shown in Algorithm 1. A simple example of the optimization process in the 2-D space is presented in Fig. 7. The value of $\frac{o_{i,j}-n/4}{n/4}$ is getting smaller during the iterative optimization process, and it has a similar role to the learning rate.

The time complexity of our iterative process is $O((l^2 + lD)n)$, which is comparable to those of the state-of-the-art techniques (e.g., $O(D^2n)$ of spectral hashing [10]). In practice, our iterative process is finished within 30 iterations. Also, its overall computation time is less than 30 seconds even for 128 bits code lengths. The convergence rate with respect to the number of iterations is shown in Fig. 6.

Algorithm 1 Our iterative optimization process

Input: sample points $S = \{s_1, ..., s_n\}$, error tolerances ϵ_m and ϵ_s, and the number of hash functions l

Output: pivot positions $p_1, ..., p_l$ and distance thresholds $t_1, ..., t_l$ for l hyperespheres

Initialize $p_1, ..., p_l$ with randomly chosen l data points from the set S
(It can be replaced with $p_i = \frac{1}{g}\sum_{j=1}^{g} q_j$ for quicker convergence, where q_j are randomly selected points from S)
Determine $t_1, ..., t_l$ to satisfy $o_i = \frac{n}{2}$ (Sec. 3.5)
Compute $o_{i,j}$ for each pair of hashing functions
repeat
  for $i = 1$ to $l - 1$ **do**
    for $j = i + 1$ to l **do**
      $f_{i \leftarrow j} = \frac{1}{2}\frac{o_{i,j} - n/4}{n/4}(p_i - p_j)$
      $f_{j \leftarrow i} = -f_{i \leftarrow j}$
    end for
  end for
  for $i = 1$ to l **do**
    $f_i = \frac{1}{l}\sum_{j=1}^{l} f_{i \leftarrow j}$
    $p_i = p_i + f_i$
  end for
  Determine $t_1, ..., t_l$ to satisfy $o_i = \frac{n}{2}$ (Sec. 3.5)
  Compute $o_{i,j}$ for each pair of hashing functions
until $avg(\mid o_{i,j} - \frac{n}{4} \mid) \leq \epsilon_m \frac{n}{4}$ and $std\text{-}dev(o_{i,j}) \leq \epsilon_s \frac{n}{4}$

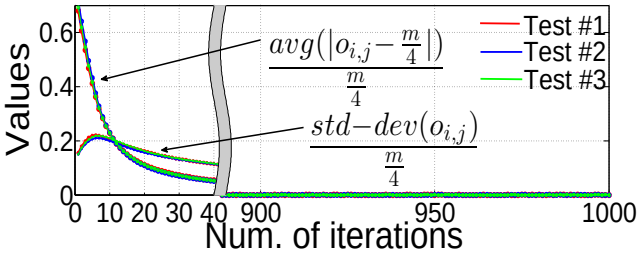


Fig. 6. Convergence rates of our iterative optimization with three individual trials. The optimization processes are finished within 30 iterations when we se we set ϵ_m as 0.1 and ϵ_s as 0.15. This graph also shows that both objectives converge to 0 when we increase the number of iterations. This result is obtained with the **GIST-1M-384D** dataset at the 64-bit code length.

One may wonder why we do not use k-means to compute centers of hyperspheres. Using k-means clustering to obtain centers of hyperspheres is very intuitive, since k-means locates the centers in dense regions and assigning the same hash value to those dense regions seems an appropriate direction. However, this alternative does not ensure the independence between hashing functions. The cluster centers obtained by k-means clustering in a high dimensional space are highly likely to be close to the data mean. It leads that hyperspheres are highly overlapped, and a high portion of regions are not covered by any hypersphere. As a result, the alternative optimization scheme does not meet our independence criteria, since hashing functions corresponding to highly overlapped hyperspheres will generated correlated hash values.

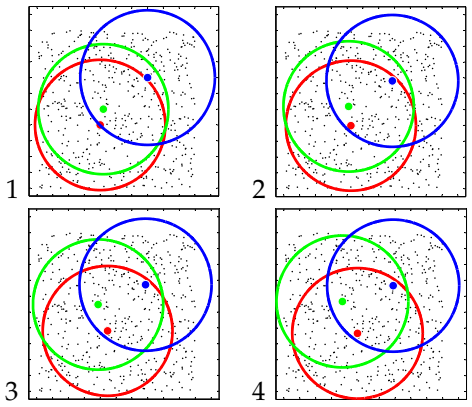


Fig. 7. Visualization of our optimization process with three hyperspheres and 500 points in 2D space.

100-NN mAP with GIST-1M-384D

# bits	32	64	128	256	512
SHD+M	0.0153	0.0426	0.0981	0.1760	0.2572
SHD	0.0147	0.0409	0.0938	0.1678	0.2434
HD+M	0.0113	0.0310	0.0665	0.1152	0.1648
HD	0.0107	0.0290	0.0653	0.1113	0.1583

1000-NN mAP with GIST-1M-960D

# bits	32	64	128	256	512
SHD+M	0.0460	0.0982	0.1782	0.2738	0.3560
SHD	0.0439	0.0945	0.1756	0.2641	0.3398
HD+M	0.0322	0.0660	0.1132	0.1669	0.2103
HD	0.0310	0.0636	0.1126	0.1644	0.2058

TABLE 3
The effect of our max-margin based distance thresholding (Sec. 3.5) indicated by **M**. The max-margin based distance thresholding improves mAPs 3.8% on average over the median based distance thresholding across various settings of experiments.

3.5 Max-Margin based Distance Thresholding

In each iteration step, we need to determine distance thresholds $t_1, ..., t_l$ to satisfy $o_i = \frac{n}{2}$ for the balanced partitioning. For this we could simply set each t_i as $d(p_i, s_{n/2})$ the distance from p_i to $s_{n/2}$, when samples of S are sorted into $s_1, ..., s_n$ in terms of distance from p_i. However, this simple approach could lead to undesirable partitioning, especially when $s_{n/2}$ is located in a dense region. To alleviate this concern, we set the distance threshold t_i to maximize the margin from points to to the hypersphere without severely comprising the balance partition criterion. For our max-margin based threshold optimization, we first sort samples of S into $s_1^s, ..., s_n^s$ according to $d(p_i, s_j^s)$

the distance to the pivot. Instead of simply using the median point $s^s_{n/2}$ with its index, $n/2$, indicating a sample in the ordered list, we compute a set J containing candidate indices near the median $\frac{n}{2}$ for the optimization:

$$J = \{j|(\frac{1}{2} - \beta)n \leq j \leq (\frac{1}{2} + \beta)n, j \in \mathbb{Z}^+\}, \quad (10)$$

where β is a parameter that controls the degree of tolerance for breaking the balance partition criterion. We set $\beta = 0.05$ in practice. We then compute an index $\hat{j}$ of a sample among the sorted list that maximizes the margin to the hypersphere as the following:

$$\hat{j} = \arg\max_{j \in J} d(t_i, s^s_{j+1}) - d(t_i, s^s_j) \quad (11)$$

The distance threshold t_i is finally determined such that the hypersphere partitions $s^s_{\hat{j}}$ and $s^s_{\hat{j}+1}$ as the following:

$$t_i = \frac{1}{2}(d(t_i, s^s_{\hat{j}}) + d(t_i, s^s_{\hat{j}+1})) \quad (12)$$

Table. 3 shows that the max-margin based distance thresholding can lead to performance improvement of 3.8% mAPs in average.

4 Generalized Spherical Hashing

Many applications benefit from the use of domain-specific kernels that define data similarities [41], [42]. In this section we generalize our basic spherical hashing (Sec. 3) to a kernelized one.

Let us first define notations. Given a set of N data elements in an input space $\mathcal{X}$, we use $X = \{x_1, x_2, ..., x_N\} \in \mathcal{X}$ to denote those data elements. We use a non-linear map $\Phi : \mathcal{X} \rightarrow \mathcal{F}$ from the input space $\mathcal{X}$ to a *kernel space* $\mathcal{F}$. We denote $k(x, y) = \langle \Phi(x), \Phi(y) \rangle$ as a kernel function corresponding to the map Φ, where $\langle \cdot, \cdot \rangle$ is the inner product operator.

4.1 Kernelized Binary Code Embedding Function

The squared distance between two points $\Phi(x)$ and $\Phi(y)$ in the kernel space $\mathcal{F}$ can be expressed with the kernel function as the following:

$$\begin{aligned} &\| \Phi(x) - \Phi(y) \|^2 \\ &= \langle \Phi(x), \Phi(x) \rangle - 2\langle \Phi(x), \Phi(y) \rangle + \langle \Phi(y), \Phi(y) \rangle \\ &= k(x, x) - 2k(x, y) + k(y, y). \end{aligned} \quad (13)$$

Our binary code construction function $H(x) = (h_1(x), ..., h_l(x))$ maps a data element in the input space into the Hamming space $\{-1, +1\}^l$. Each kernelized spherical hashing function $h_i(x)$ is defined with the pivot point p_i in the kernel space and distance threshold t_i as the following:

$$h_i(x) = \begin{cases} -1 & \text{when } \| \Phi(x) - p_i \|^2 > t_i^2 \\ +1 & \text{when } \| \Phi(x) - p_i \|^2 \leq t_i^2 \end{cases}. \quad (14)$$

Intuitively, each kernelized spherical hashing function $h_i(x)$ determines whether $\Phi(x)$, the point x mapped into the kernel space, is inside the hypersphere defined by its center p_i and radius t_i.

To represent the center of a hypersphere in the kernel space, we use a set of m landmark samples $Z = \{z_1, ..., z_m\} \in X$, where $m \ll n$. We now express the center p_i by a linear combination of $\{\Phi(z_1), ..., \Phi(z_m)\}$ as the following:

$$p_i = \sum_{j=1}^{m} w^i_j \Phi(z_j), \quad (15)$$

where $w^i_j \in \mathbb{R}$ denotes a weight of $\Phi(z_j)$ for p_i.

The squared distance between a point $\Phi(x)$ and the pivot p_i used in our kernelized spherical hashing function is computed as the following:

$$\begin{aligned} &\| \Phi(x) - p_i \|^2 \\ &= \langle \Phi(x), \Phi(x) \rangle - 2\langle \Phi(x), p_i \rangle + \langle p_i, p_i \rangle \text{ [by Eq. 13]} \\ &= \langle \Phi(x), \Phi(x) \rangle - 2\langle \Phi(x), \sum_{j=1}^{m} w^i_j \Phi(z_j) \rangle \\ &\quad + \langle \sum_{j=1}^{m} w^i_j \Phi(z_j), \sum_{j=1}^{m} w^i_j \Phi(z_j) \rangle \quad \text{[by Eq. 15]} \\ &= \langle \Phi(x), \Phi(x) \rangle - 2\sum_{j=1}^{m} w^i_j \langle \Phi(x), \Phi(z_j) \rangle \\ &\quad + \sum_{j=1}^{m}\sum_{g=1}^{m} w^i_j w^i_g \langle \Phi(z_j), \Phi(z_g) \rangle \quad (16) \\ &= k(x, x) - 2\sum_{j=1}^{m} w^i_j k(x, z_j) + \sum_{j=1}^{m}\sum_{g=1}^{m} w^i_j w^i_g k(z_j, z_g) \end{aligned}$$

Note that the last term $\sum_{j=1}^{m}\sum_{g=1}^{m} w^i_j w^i_g k(z_j, z_g)$ can be pre-computed for each hypersphere, since it is independent of x [43].

4.2 Kernelized Iterative Optimization

We first sample a training set $S = \{s_1, ..., s_n\}$ from X to approximate its distribution, and also sample a subset $Z = \{z_1, ..., z_m\}$ from S as landmarks that are used for defining center positions of hyperspheres, as described in Eq. 15.

As an initial step of our optimization process, we sample a training set $T = \{t_1, ..., t_n\}$ from the dataset X to approximate its distribution, and also sample a subset $Z = \{z_1, ...z_m\}$ from T as landmarks that are used for defining center positions of hyperspheres, as described in Eq. 15.

Initial center positions p_i of hyperspheres are chosen randomly. Specifically we initialize each element of the weight vectors w^i that define centers p_i of hyperspheres with randomly drawn values from the uniform distribution $U(-1, 1)$ and normalize the weight vectors according to L_2 norm.

To express the constraints specified in Eq. 4 and Eq. 5 with the training set X, we recall the following two variables o_i and $o_{i,j}$ defined in Eq. 7.

We perform the procedure described in Sec. 3.4 and Algorithm. 1 with the following kernelized force model:

$$\begin{aligned} f_{i \leftarrow j} &= \frac{1}{2}\frac{o_{i,j} - n/4}{n/4}(p_i - p_j) \\ &= \frac{1}{2}\frac{o_{i,j} - n/4}{n/4}\left(\sum_{g=1}^{m} w_g^i \Phi(z_g) - \sum_{g=1}^{m} w_g^j \Phi(z_g)\right) \\ &= \frac{1}{2}\frac{o_{i,j} - n/4}{n/4}\sum_{g=1}^{m}(w_g^i - w_g^j)\Phi(z_g). \end{aligned} \tag{17}$$

5 Evaluation

In this section we evaluate our method and compare it with the state-of-the-art methods [8], [10], [13], [20], [21], [22]. All experiments are conducted on a single Xeon X5690 machine with 144GB memory where the complete data set can be stored.

5.1 Datasets

We perform various experiments with the following four datasets:

- **GIST-1M-384D**: A set of 384 dimensional, one million GIST descriptors, which consist of a subset of Tiny Images [9].
- **GIST-1M-960D**: A set of 960 dimensional, one million GIST descriptors that are also used in [38].
- **GIST-75M-384D**: A set of 384 dimensional, 75 million GIST descriptors, which consist of a subset of 80 million Tiny Images [9].
- **ILSVRC**: One million of 1000 dimensional BoW descriptors which is a subset of the ImageNet database [44].
- **VLAD-1M-8192D**: One million of 8192 dimensional VLAD [39] descriptors (128 dimensional SIFT features and 64 codebook vectors).

5.2 Evaluation on Euclidean Space

We first present results with the Euclidean space, followed by ones with the kernel space.

5.2.1 Protocol

We tested with randomly chosen 1000 queries for datasets **GIST-1M-384D**, **GIST-1M-960D**, and **VLAD-1M-8192D**, and 500 queries for **GIST-75M-384D** that do not have any overlap with data points. The performance is measured by mean Average Precision (mAP). The ground truth is defined by k nearest neighbors that are computed by the exhaustive, linear scan based on the Euclidean distance. When calculating precisions, we consider all the items having lower or the equal Hamming distance (or spherical Hamming distance) from given queries.

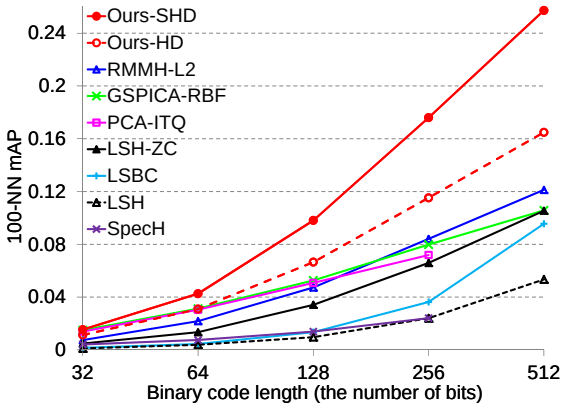


Fig. 8. Comparison between our method and the-state-of-the-art methods with the **GIST-1M-384D** dataset when $k = 100$.

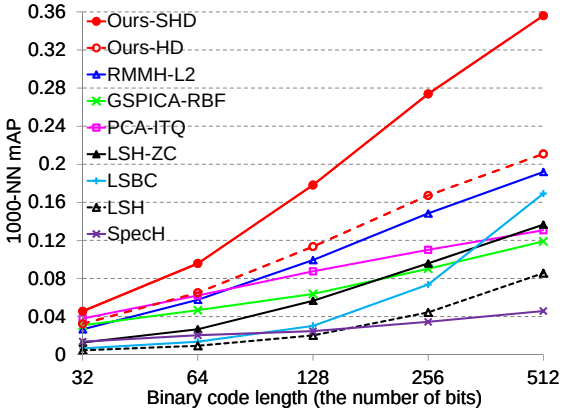


Fig. 9. Comparison between our method and the-state-of-the-art methods with the **GIST-1M-960D** dataset when $k = 1,000$.

5.2.2 Compared Methods

- **LSH** and **LSH-ZC**: Locality Sensitive Hashing [8] with/without Zero Centered data points.
- **LSBC**: Locality Sensitive Binary Codes [13]. The bandwidth parameter used in experiment is the inverse of the mean distance between the points in the dataset, as suggested in [45].
- **SpecH**: Spectral Hashing [10].
- **PCA-ITQ**: Iterative Quantization [20].
- **RMMH-L2**: Random Maximum Margin Hashing (RMMH) [22] with the triangular L2 kernel. We experiment RMMH with the triangular L2 kernel since the authors reported the best performance on k nearest neighbor search with this kernel. We use 32 for the parameter M that is the number of samples for each hash function, as suggested by [22].
- **GSPICA-RBF**: Generalized Similarity Preserving Independent Component Analysis (GSPICA) [21] with the RBF kernel. We experiment GSPICA with the RBF kernel, since the authors reported the best performance on k nearest neighbor

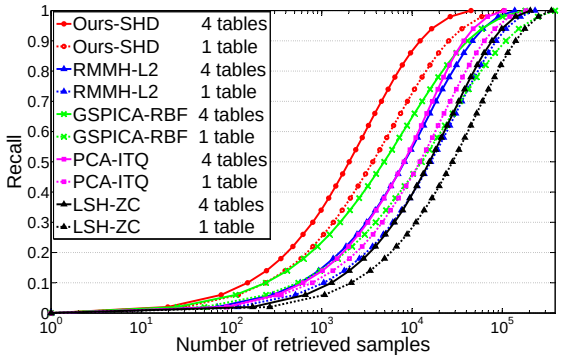


Fig. 10. Recall curves of different methods when $k = 100$ for the **GIST-1M-384D** dataset. Each hash table is constructed by 64 bits code lengths. The recall (y-axis) represents search accuracy and the number of retrieved samples (x-axis) represents search time.

search with this kernel. The parameter used in the RBF kernel is determined by the mean distance of kth nearest neighbors within training samples as suggested by [22]. The parameters γ and P are 1 and the dimensionality of the dataset respectively, as suggested in [21].

- **Ours-HD** and **Ours-SHD**: We have tested two different versions of our method. **Ours-HD** represents our method with the common Hamming distance, while **Ours-SHD** uses our spherical Hamming distance (Sec. 3.2). Max-margin based distance thresholding scheme (Sec. 3.5) is also applied to both versions of our method.

For all the data-dependent hashing methods, we randomly choose 100K data points from the original dataset as a training set. We also use the same training set to estimate parameters of each method. We report the average mAP and recall values by repeating all the experiments five times, in order to gain statistically meaningful values; for **GIST-75M-384D** benchmark, we repeat experiments only three times because of its long experimentation time. Note that we do not report results of two PCA-based methods **SpecH** and **PCA-ITQ** for 512 hash bits at 384 dimensional datasets, since they do not support bit lengths larger than the dimension of the data space.

5.2.3 Results

Fig. 8 shows the mAP of k nearest neighbor search of all the tested methods when $k = 100$. Our method with the spherical Hamming distance, **Ours-SHD**, shows better results over all the tested methods across all the tested bit lengths ranging from 32 bits to 512 bits. Furthermore, our method shows increasingly higher benefits over all the other tested methods as we allocate more bits. This increasing improvement is mainly because using multiple hyperspheres can effectively create closed regions with tighter distance bounds compared to hyperplanes.

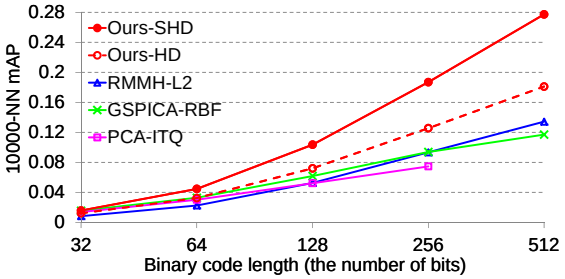


Fig. 11. Comparison between our method and the-state-of-the-art methods with the **GIST-75M-384D** dataset when $k = 10,000$.

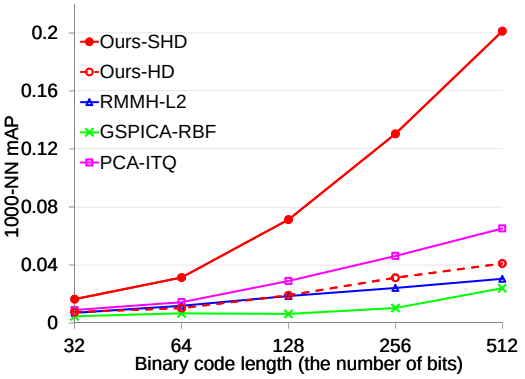
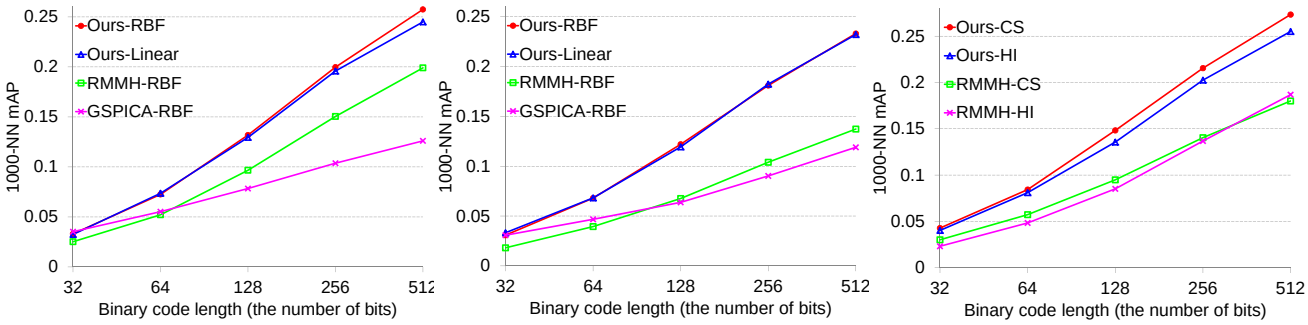


Fig. 12. Comparison between our method and the-state-of-the-art methods with the **VLAD-1M-8192D** dataset when $k = 1,000$.

Given 0.1 mAP in Fig. 8, our method needs to use 128 bits to encode each image. On the other hand, other tested methods should use more than 256 bits. As a result, our method provides over two times more compact data representations than other methods. We would like to point out that low mAP values of our method are still very meaningful, as discussed in [22]. Once we identify nearest neighbor images based on binary codes, we can employ additional re-ranking processes on those images. As pointed out in [22], 0.1 mAP given $k = 100$ nearest neighbors, for example, indicates that 1000 images on average need to be re-ranked.

Performances of our methods with two different binary code distance functions are also shown in Fig. 8. Our method with the Hamming distance **Ours-HD** shows better results than most of other methods across different bits, especially higher bits. Furthermore, the spherical Hamming distance **Ours-SHD** shows significantly improved results even than **Ours-HD**. The spherical Hamming distance function also shows increasingly higher improvement over the Hamming distance, as we add more bits for encoding images.

Our technique can be easily extended to use multiple hash tables; for example, we can construct a new hash table by recomputing S, the subset of the original



(a) GIST-1M-384D (b) GIST-1M-960D (c) ILSVRC

Fig. 15. k-nearest neighbor search performances on three different datasets when $k = 1000$.

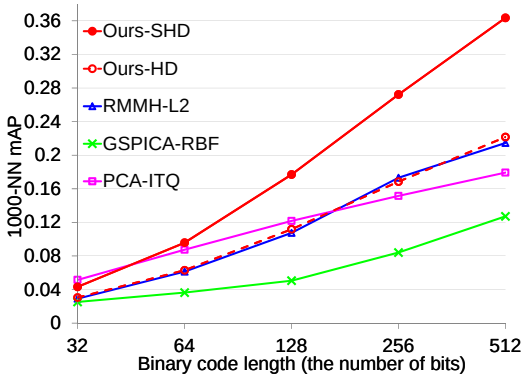


Fig. 13. Comparison between our method and the-state-of-the-art methods with the **L2-normalized GIST-1M-960D** dataset when $k = 1,000$.

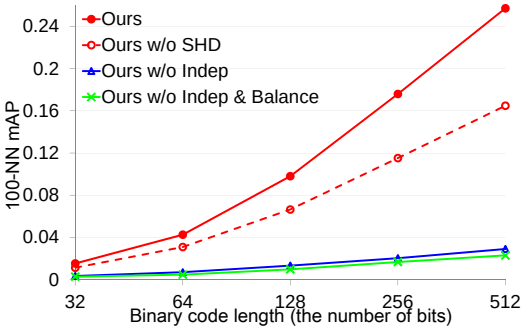


Fig. 14. This figure shows how each component of our method affects the accuracy. The mAP curves are obtained with GIST-1M-384D dataset when $k = 100$.

dataset. Fig. 10 shows recall curves of different methods with varying numbers of hash tables, when we allocate 64 bits for encoding each image. Our method (with our spherical Hamming distance) improves the accuracy as we use more tables. More importantly, our method only with a single table shows significantly improved results over all the other tested methods that use four hash tables.

We have also performed all the tests against the 960 dimensional, one million GIST dataset **GIST-1M-960D** with $k = 1000$ (Fig. 9). We have found that our method shows similar trends even with this dataset, compared to what we have achieved in **GIST-1M-384D**.

We have performed each test multiple times, since our method can be impacted by different initializations. However, our method shows robust results against different initializations. For example, the standard deviation of mAPs of five experiments with **GIST-1M-960D** when the code length is 64 bits is only 0.0017, while the average mAP is 0.0982. The standard deviation with 256 bits is 0.0035, while the average is 0.2738.

In order to evaluate our method with very high dimensional data, we have performed the tests with 8192 dimensional, one million VLAD dataset **VLAD-1M-8192D** with $k = 1000$ (Fig. 12). **Ours-SHD** consistently provides the best performance among the tested techniques.

We have also performed all the tests against the 384 dimensional, 75 million GIST dataset **GIST-75M-384D** with $k = 10,000$ (Fig. 11). We have found that our method shows significantly higher results than all the other tested methods across all the tested bit lengths even with this large-scale dataset.

Since many applications use L2-normalized data, we have performed an experiment with **L2-normalized GIST-1M-960D** with $k = 1,000$ (Fig. 13). **Ours-HD** provides similar accuracy to **RMMH**, but lower accuracy with short code lengths and higher accuracy with long code lengths compared to **PCA-ITQ**. It implies that minimizing quantization error is important in the short code lengths, but independence among hash functions is more important in the long code lengths. Nevertheless, **Ours-SHD** outperformed the tested state-of-the-art methods in most configurations. Note that hyperplane based hashing techniques do not receive any benefit by using **SHD** even in the normalized data. This result confirms that we can still exploit the benefit of using hyperspheres over hyperplanes even with the L2-normalized dataset.

In order to see how each component of our method

affects the accuracy, we measure mAP by disabling the spherical Hamming distance, the independence constraint, and the balanced partitioning criterion in our method (Fig. 14). In the case of using 64 bits, mAP of our method goes down 28%, 83%, and 90% by disabling the spherical Hamming distance, the independence constraint, and the balanced partitioning/independence constraints respectively. From these, we can see all proposed ideas are critical for the effectiveness of the proposed method, while the criteria of hash code independence and balance partition are most important.

Finally, we also measure how efficiently our hypersphere-based hashing method generates binary codes given a query image. Our method takes 0.08 ms for generating a 256 bit-long binary code. This cost is same to that of the LSH.

5.3 Evaluation on Kernel Space

5.3.1 Datasets

We normalized **GIST-1M-384D** and **GIST-1M-960D** datasets according to L_2-norm to make exact k-nearest neighbors with the linear kernel to be equivalent to the k-nearest neighbors with the RBF kernel as did in [22]. As a result, we can compare results acquired by using two different kernels in the same ground. We also normalized **ILSVRC** dataset according to L_1-norm as suggested in [46]. For all the experiments, we tested with randomly chosen one thousand queries that do not overlap with data elements. We report the average result by repeating all the experiments three times.

5.3.2 Compared Methods

- **RMMH**: Random Maximum Margin Hashing [22]. We used 32 for the parameter M that is the number of samples for each hash function as suggested by [22].
- **GSPICA**: Generalized Similarity Preserving Independent Component Analysis [21]. We set the parameter P as the dimensionality of the dataset and γ to 1 as suggested in the paper.

For our method **Ours**, we set the number of landmarks m as the dimensionality of input data.

5.3.3 Used Kernels

We have tested our method with the following four popular kernels:

- **Linear**: Linear kernel, $k(x, y) = \langle x, y \rangle$.
- **RBF**: RBF kernel, $k(x, y) = exp(-\gamma \parallel x - y \parallel^2)$. We set the bandwidth parameter γ as an inverse of the mean distance between randomly sampled points as suggested by [45].
- **HI**: Histogram Intersection kernel, $k(x, y) = \sum_{i=1}^{D} min(x_i, y_i)$, where D is the dimensionality of x and y.
- **CS**: Chi Square kernel, $k(x, y) = 2 \sum_{i=1}^{D} \frac{x_i y_i}{x_i + y_i}$.

5.3.4 Results

We evaluated our method with k-nearest neighbor search. The ground truth is defined by top k data elements based on each tested kernel function. The performance is measured by mAP. When calculating precisions, we consider all the items having lower or equal Hamming distance from given queries.

Fig. 15 shows the mAP of k-nearest neighbor search of all the tested methods when $k = 1000$; other cases (e.g. $k = 50, 100, 500, 2000$) show similar trends. We evaluated the performance of our method with **Linear** and **RBF** in **GIST384D** and **GIST960D** datasets (Fig. 15-(a) and -(b)); we report results of compared methods only with **RBF**, since **RBF** gives better results than **Linear**.

We also experimented with **CS** and **HI** in **ILSVRC** dataset (Fig. 15-(c)). Since **RMMH** performed better than **GSPICA** in this experiment, we report results of **RMMH** in this graph. We have found that our method consistently shows higher performance than the state-of-the-art methods in all the tested benchmarks with various kernels.

5.4 Evaluation on Image Retrieval

We evaluated image retrieval performance of our method by using the **ILSVRC**, which has 1000 different classes. We followed the evaluation protocol of [22]. For each query we run a k-nearest neighbor classifier on the top 1000 results retrieved by each method. As suggested in ILSVRC, we evaluated tested methods with the five best retrieved classes (i.e. recognition rate@5). Specifically we first perform 1000-nearest neighbor search for a gien query with binary codes. We then evaluate the correctness of the five most frequent classes within the 1000-nearest neighbor images contain the ground truth class.

Fig. 16 shows the recognition rates of our method compared to **RMMH** with **CS** and **HI** kernels. Our method consistently gives better retrieval performance with both kernels over the other tested methods.

5.5 Discussion

SHD (Sec. 3.2) drastically improves mAPs in the Euclidean space (Sec. 5.2). However, we observed that **SHD** does not provide significant accuracy improvements with the generalized spherical hashing. Table 4 shows how much the **SHD** improves mAPs of the generalized spherical hashing over **HD** with two popular kernels, and **SHD** provides 3.5% benefits on the mAPs over the Hamming distance. The reason why **SHD** shows a relatively small benefit for generalized spherical hashing is that **SHD** does not directly reflect inner product, since it is designed to better reflect the Euclidean distance. Nonetheless, generalized spherical hashing with both of **HD** and **SHD** outperforms state-of-the-art kernelized binary code embedding methods.

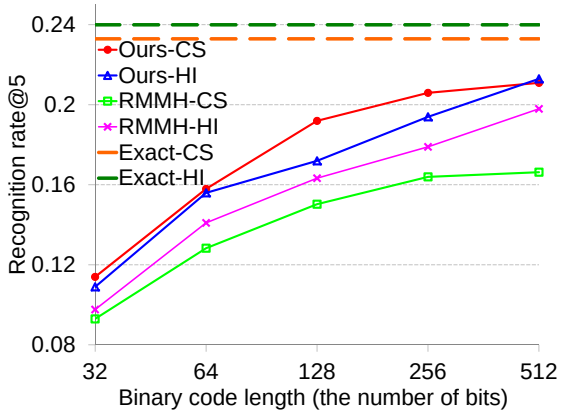


Fig. 16. Image retrieval performances over the **ILSVRC** dataset. Exact-CS and Exact-HI are the upper bound of recognition rates obtained by the exact 1000-nearest neighbor classifier based on the corresponding kernel functions.

1000-NN mAP with GIST-1M-384D

# bits	32	64	128	256	512
RBF-SHD	0.0358	0.0789	0.1336	0.1992	0.2587
RBF-HD	0.0330	0.0725	0.1318	0.1997	0.2575
Linear-SHD	0.0345	0.0671	0.1385	0.1974	0.2424
Linear-HD	0.0325	0.0736	0.1295	0.1958	0.2450

1000-NN mAP with GIST-1M-960D

# bits	32	64	128	256	512
RBF-SHD	0.0316	0.0672	0.1198	0.1757	0.2451
RBF-HD	0.0308	0.0680	0.1220	0.1811	0.2331
Linear-SHD	0.0335	0.0733	0.1270	0.1932	0.2517
Linear-HD	0.0332	0.0682	0.1194	0.1824	0.2321

TABLE 4
The effect of **SHD** with generalized spherical hashing.

6 Conclusion

In this work we have proposed a novel hypersphere-based binary embedding technique, spherical hashing, for providing a compact data representation and highly scalable nearest neighbor search with high accuracy. We have found that spherical hashing significantly outperforms the tested six state-of-the-art binary code embedding techniques based on hyperplanes with one and 75 million high-dimensional image descriptors. We have also proposed generalized spherical hashing to support various similarity metrics defined by arbitrary kernel functions, and we have demonstrated on three datasets with four popular kernels that generalized spherical hashing improves the state-of-the-art techniques.

References

[1] J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon, "Spherical hashing," in *CVPR*, 2012.
[2] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, pp. 509–517, September 1975.
[3] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM TOMS*, vol. 3, no. 3, pp. 209–226, 1977.
[4] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984.
[5] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISAPP*, 2009, pp. 331–340.
[6] P. Indyk and R. Motwani, "Approximate nearest neighbors: toward removing the curse of dimensionality," in *STOC*, 1998.
[7] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *STOC*, 2002.
[8] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *SoCG*, 2004.
[9] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *CVPR*, 2008.
[10] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *NIPS*, 2008.
[11] O. Chum, J. Philbin, and A. Zisserman, "Near duplicate image detection: min-hash and tf-idf weighting," in *BMVC*, 2008.
[12] P. Jain, B. Kulis, and K. Grauman, "Fast image search for learned metrics," in *CVPR*, 2008.
[13] M. Raginsky and S. Lazebnik, "Locality sensitive binary codes from shift-invariant kernels," in *NIPS*, 2009.
[14] R. Salakhutdinov and G. Hinton, "Semantic hashing," *International Journal of Approximate Reasoning*, 2009.
[15] J. Wang, S. Kumar, and S.-F. Chang, "Sequential projection learning for hashing with compact codes," in *ICML*, 2010.
[16] L. Pauleve, H. Jégou, and L. Amsaleg, "Locality sensitive hashing: a comparison of hash function types and querying mechanisms," *Pattern Recognition Letters*, 2010.
[17] J. Wang, S. Kumar, and S.-F. Chang, "Semi-supervised hashing for scalable image retrieval," in *CVPR*, 2010.
[18] R.-S. Lin, D. Ross, and J. Yangik, "Spec hashing: Similarity preserving algorithm for entropy-based coding," in *CVPR*, 2010.
[19] M. Norouzi and D. J. Fleet, "Minimal loss hashing for compact binary codes," in *ICML*, 2011.
[20] Y. Gong and S. Lazebnik, "Iterative quantization: a procrustean approach to learning binary codes," in *CVPR*, 2011.
[21] J. He, R. Radhakrishnan, S.-F. Chang, and C. Bauer, "Compact hashing with joint optimization of search accuracy and time," in *CVPR*, 2011.
[22] A. Joly and O. Buisson, "Random maximum margin hashing," in *CVPR*, 2011.
[23] W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang, "Supervised hashing with kernels," in *CVPR*, 2012.
[24] T. cker Chiueh, "Content-based image indexing," in *In Proceedings of the 20th VLDB Conference*, 1994, pp. 582–593.
[25] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 322–373, Sep. 2001.
[26] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, Sep. 2001.
[27] C. Silpa-Anan, R. Hartley, S. Machines, and A. Canberra, "Optimised kd-trees for fast image descriptor matching," in *CVPR*, 2008.
[28] Y. Jia, J. Wang, G. Zeng, H. Zha, and X.-S. Hua, "Optimizing kd-trees for scalable visual descriptor indexing," in *CVPR*, 2010.
[29] K. Kim, M. K. Hasan, J.-P. Heo, Y.-W. Tai, and S.-E. Yoon, "Probabilistic cost model for nearest neighbor search in image retrieval," *Computer Vision and Image Understanding (CVIU)*, 2012.
[30] D. Nistér and H. Stewénius, "Scalable recognition with a vocabulary tree," in *CVPR*, 2006.
[31] A. Joly and O. Buisson, "A posteriori multi-probe locality sensitive hashing," in *Proceedings of the 16th ACM international conference on Multimedia*, ser. MM '08. New York, NY, USA: ACM, 2008, pp. 209–218.
[32] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *ICCV*, 2009.
[33] W. Liu, J. Wang, S. Kumar, and S.-F. Chang, "Hashing with graphs," in *ICML*, 2011.

[34] C. Strecha, A. M. Bronstein, M. M. Bronstein, and P. Fua, "Lda-hash: Improved matching with smaller descriptors," *PAMI*, 2010.
[35] W. Kong and W.-J. Li, "Double-bit quantization for hashing," in *AAAI*, 2012.
[36] Y. Lee, J.-P. Heo, and S.-E. Yoon, "Quadra-embedding: Binary code embedding with low quantization error," in *ACCV*, 2012.
[37] W. Kong, W.-J. Li, and M. Guo, "Manhattan hashing for large-scale image retrieval," in *SIGIR*, 2012.
[38] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE TPAMI*, 2011.
[39] H. Jégou, M. Douze, C. Schmid, and P. Pérez, "Aggregating local descriptors into a compact image representation," in *CVPR*, 2010, pp. 3304 –3311.
[40] A. M. Yaglom and I. M. Yaglom, *Challenging Mathematical Problems with Elementary Solutions*. New York: Dover Pub., Inc., 1987.
[41] K. Grauman and T. Darrell, "The pyramid match kernel: discriminative classification with sets of image features," in *ICCV*, 2005.
[42] J. Zhang, M. Marszalek, S. Lazebnic, and C. Schmid, "Local features and kernels for classification of texture and object categories: A comprehensive study," *IJCV*, 2007.
[43] I. S. Dhillon, Y. Guan, and B. Kulis, "Kernel k-means: spectral clustering and normalized cuts," in *KDD*, 2004.
[44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, june 2009, pp. 248 –255.
[45] A. Gordo and F. Perronnin, "Asymmetric distances for binary embeddings," in *CVPR*, 2011.
[46] F. Perronnin, J. Sánchez, and Y. Liu, "Large-scale image categorization with explicit data embedding," in *CVPR*, 2010.