# Spherical Hashing

Jae-Pil Heo[1], Youngwoon Lee[1], Junfeng He[2], Shih-Fu Chang[2], Sung-Eui Yoon[1]
[1] KAIST [2] Columbia University

## Abstract

*Many binary code encoding schemes based on hashing have been actively studied recently, since they can provide efficient similarity search, especially nearest neighbor search, and compact data representations suitable for handling large scale image databases in many computer vision problems. Existing hashing techniques encode high-dimensional data points 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. Furthermore, we propose a new binary code distance function,* spherical Hamming distance, *that is tailored to our hypersphere-based binary coding scheme, and design an efficient iterative optimization process to achieve balanced partitioning of data points for each hash function and independence between hashing functions. Our extensive experiments show that our spherical hashing technique significantly outperforms six state-of-the-art hashing techniques based on hyperplanes across various image benchmarks of sizes ranging from one to 75 million of GIST descriptors. The performance gains are consistent and large, up to 100% improvements. The excellent results confirm the unique merits of the proposed idea in using hyperspheres to encode proximity regions in high-dimensional spaces. Finally, our method is intuitive and easy to implement.*

## 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 [6] have been used for low-dimensional data points. Unfortunately, these techniques are not scalable to high-dimensional data points. Hence recently hashing techniques have been actively studied to provide efficient solutions for such high-dimensional data points [11, 23, 26].

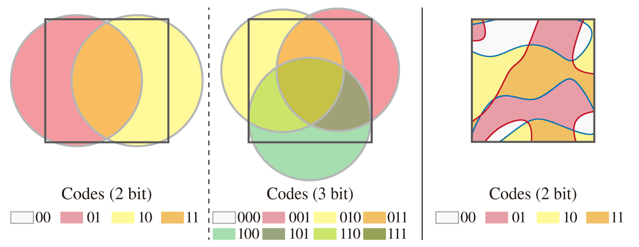

Figure 1. The first two images show partitioning examples of spherical hashing functions in a 2D space for 2 and 3 bit binary codes. The rectangle in the images represents the boundary of data points. The rightmost figure shows partitioning boundaries of non-linear hashing functions used in the GSPICA method [10], which is based on hyperplanes defined in its RBF kernel space. The partitioning boundary is visualized in the 2D original data space.

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 input points. Locality-Sensitive Hashing (LSH) [11] is one of the most widely known techniques in this category. This technique is extended to various hashing functions [3, 1, 13, 2, 20]. 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 [26], semi-supervised hashing [25], iterative quantization [7], joint optimization [10], and random maximum margin hashing [15].

In all of these existing hashing techniques, hyperplanes are used to partition the data points (located in the original data space or a kernel space) 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 in the original data space than hyperplanes. 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 closed region even in an arbitrarily high dimensional space. One can find that hyperplanes in a kernel space are able to map to non-linear hashing functions (Fig. 1). However, we have found that the proposed simple spherical hashing in the original space achieves more spatially coherent partitioning than the non-linear hashing functions used in recent works [10, 15, 20].

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 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).

In order to highlight benefits of our method, we have tested our method against different benchmarks that consists of one to 75 million image feature points 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. 4).

## 2. Related Work

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

### 2.1. Image Representations

To identify visually similar images for a query image, many image representations have been studied [4]. Examples of the most popular schemes include the Bag-of-visual-Words representation [21] and GIST descriptor [19], which have been known to work well in practice. These image descriptors have high dimensionality (e.g. hundreds to thousands) and identifying similar images is typically reduced to finding nearest neighbor points in those high dimensional, image descriptor spaces [17]. Since these image descriptor spaces have high dimensions, finding nearest neighbor image descriptors has been known to be very hard because of the 'curse of dimensionality' [11].

### 2.2. Tree-based Methods

Space partition based tree structures such as kd-trees [6] have been used to find nearest neighbors and optimized for higher performance [16]. It has been widely known, however, that kd-tree based search can run slower than linear scan for high dimensional points. Nistér and Stewénius [18] proposed another tree-based nearest neighbor search scheme based on hierarchical k-means trees.

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.

### 2.3. Binary Hashing Methods

Binary hashing methods aim to embed points in binary codes, while preserving relative distances among them. One of the most popular hashing techniques is LSH [11]. Its hash function is based on projection onto random vectors drawn from specific distribution. Many variations of LSH have been proposed for learned metrics [13], min-hash [2], random Fourier features [20], etc. [3, 1]. Since hashing functions in LSH are drawn independently from the data points, it could be inefficient especially for short lengths of binary codes.

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. [26] have proposed data-dependent, spectral hashing motivated by spectral graph partitioning. It improved performance over LSH especially for compact bit lengths (i.e. 8 and 16 bits). Wang et al. [25] proposed a semi-supervised hashing method to improve image retrieval performance by exploiting label information of the training set. Gong and Lazebnik [7] introduced a procrustean approach that directly minimizes quantization error by rotating zero-centered PCA-projected data. He et al. [10] presented a hashing method that jointly optimizes both search accuracy and search time. Joly and Buisson [15] constructed hash functions by using large margin classifiers with arbitrarily sampled data points that are randomly separated into two sets.

All the mentioned hashing techniques compute binary codes by partitioning data points (located in the original feature space or a kernel space) into two different sets based on hyperplanes. Departing from this conventional approach, we adopt a novel approach of partitioning data points by hyperspheres.

We found a similarly named method, spherical LSH [22]. Our method is totally different from this spherical LSH, which is a specialized technique for data points located on the unit hypersphere.

## 2.4. Distance based Indexing Methods

The database community has been designing efficient techniques for indexing high dimensional points and supporting various proximity queries. Filho et al. [5] index points with distances from fixed *pivot* points. As a result, a region with a same index given a pivot becomes a ring shape. This method reduces the region further by using multiple pivot points. They then built various hierarchical structures (e.g., R-tree) to support various proximity queries. The efficiency of this method highly depends on the locations of pivots. For choosing pivots, Jagadish et al. [12] used k-means clustering and Venkateswaran et al. [24] adopted other heuristics such as maximizing the variance of distances from pivots to data points.

This line of work uses a similar concept to ours in terms of using distances from pivots for indexing high-dimensional points. However, our approach is drastically different from these techniques, since ours aims to compute compact binary codes preserving the original metric spaces of feature points by using hashing, while theirs targets for designing hierarchical indexing structures supporting efficient proximity queries.

## 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 $b_i = \{-1, +1\}^c$, where $c$ is the length of the code.

### 3.1. Binary Code Embedding Function

Our hashing function $H(x) = (h_1(x), ..., h_c(x))$ maps points in $\mathbb{R}^D$ into the binary cube $\{-1, +1\}^c$. We use a hypersphere to define a spherical hashing function. Each spherical hashing function $h_k(x)$ is defined by a pivot $p_k \in \mathbb{R}^D$ and a distance threshold $t_k \in \mathbb{R}^+$ as the following:

$$h_k(x) = \begin{cases} -1 & \text{when } d(p_k, x) > t_k \\ +1 & \text{when } d(p_k, x) \le t_k, \end{cases}$$

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 $h_k(x)$ indicates whether the point $x$ is inside the hypersphere whose center is $p_k$ and radius is $t_k$.

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 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 $c$ hyperspheres goes up to $\binom{c-1}{d} + \sum_{i=0}^{d} \binom{c}{i}$ [27]. In addition, we can approximate a hyperplane with a large hypersphere (e.g. 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 the left figure of Fig. 2, hyperspheres bound regions of binary codes more tightly compared to hyperplanes used in LSH [3]. Across all the tested code lengths, hyperspheres show about two times tighter bounds over the hyperplane-based approach.

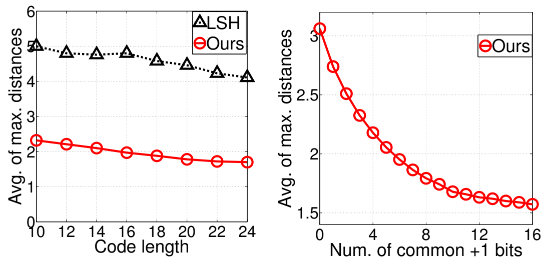

Figure 2. The left figure shows how the avg. of the max. distances among points having the same binary code changes with different code lengths based on hyperspheres or hyperplanes. We randomly sample 1000 different binary codes to compute avg. of the max. distances. The right figure shows how having 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, $(x, y)$, where $H(x) = b_i$ and $H(y) = b_j$. 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. 4.1).

### 3.2. Distance between Binary Codes

Most hyperplane-based binary 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* ($d_{shd}(b_i, b_j)$), between two binary codes $b_i$ and $b_j$ computed by spherical hashing:

$$d_{shd}(b_i, b_j) = \frac{|b_i \oplus b_j|}{|b_i \wedge b_j|},$$

where $|b_i \wedge b_j|$ denotes the number of common $+1$ bits between two binary codes.

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. 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 the right figure of Fig. 2, 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. In implementation, we add a small value to the denominator to avoid the division by zero.

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.

### 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 [26, 10, 15], since independent hashing 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 [10] and improving the accuracy even for longer bit lengths [15]. We also aim to achieve this independence between our spherical hashing functions.

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

$$Pr[\, h_k(x) = +1 \,] = \frac{1}{2}, \quad x \in X, \quad 1 \le k \le c \qquad (1)$$

Let us define a probabilistic event $V_k$ to represent the case of $h_k(x) = +1$. 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. 1), then the independence between two bits can satisfy the following equation given $x \in X$ and $1 \le i, j \le c$:

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

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. 1 and Eq. 2. 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 $c$ different hyperspheres, i.e. their pivots $p_k$ and distance thresholds $t_k$. During this iterative process we construct hyperspheres to satisfy constraints shown in Eq. 1 and Eq. 2.

As the first phase of our iterative process, we sample a subset $S = \{s_1, s_2, ..., s_m\}$ from data points $X$ to approximate its distribution. We then initialize the pivots of $c$ hyperspheres with randomly chosen $c$ 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.

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 $1 \le i, j \le c$:

$$\begin{aligned} o_i &= |\, \{s_k | h_i(s_k) = +1, 1 \le k \le m\} \,|, \\ o_{i,j} &= |\, \{s_k | h_i(s_k) = +1, h_j(s_k) = +1, 1 \le k \le m\} \,|, \end{aligned}$$

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. 1). Also, $o_{i,j}$ measures the number of data points in the subset $S$ 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. First, we adjust the pivot positions of two hyperspheres in a way that $o_{i,j}$ becomes closer to or equal to $\frac{m}{4}$. Intuitively, for each pair of two hyperspheres $i$ and $j$, when $o_{i,j}$ is greater than $\frac{m}{4}$, 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

**Algorithm 1** Our iterative optimization process

```
Input: sample points $S = \{s_1, ..., s_m\}$ , error tolerances $\epsilon_m$, $\epsilon_s$, and the number of hash functions $c$
Output: pivot positions $p_1, ..., p_c$ and distance thresholds $t_1, ..., t_c$ for $c$ hyperespheres
  Initialize $p_1, ..., p_c$ with randomly chosen $c$ data points from the set $S$
  Determine $t_1, ..., t_c$ to satisfy $o_i = \frac{m}{2}$
  Compute $o_{i,j}$ for each pair of hashing functions
  repeat
    for $i = 1$ to $c - 1$ do
      for $j = i + 1$ to $c$ do
        $f_{i \leftarrow j} = \frac{1}{2} \frac{o_{i,j} - m/4}{m/4} (p_i - p_j)$
        $f_{j \leftarrow i} = -f_{i \leftarrow j}$
      end for
    end for
    for $i = 1$ to $c$ do
      $f_i = \frac{1}{c} \sum_{j=1}^{c} f_{i \leftarrow j}$
      $p_i = p_i + f_i$
    end for
    Determine $t_1, ..., t_c$ to satisfy $o_i = \frac{m}{2}$
    Compute $o_{i,j}$ for each pair of hashing functions
  until $avg(| o_{i,j} - \frac{m}{4} |) \leq \epsilon_m \frac{m}{4}$ and $std\text{-}dev(o_{i,j}) \leq \epsilon_s \frac{m}{4}$
```

closer. Second, once pivots are computed, we adjust the distance threshold $t_i$ of $i$th hypersphere such that $o_i$ becomes $\frac{m}{2}$ to meet balanced partitioning of the data points for the hypersphere (Eq. 1).

We perform our iterative process until the computed hyperspheres do not make further improvements in terms of satisfying constraints. Specifically, we consider the 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 $\frac{m}{4}$ 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 $\epsilon_m$% and $\epsilon_s$%, error tolerances, of the ideal mean of $o_{i,j}$ respectively; we found that too low error tolerances lead to over-fitting while the values near 10% give reasonable result. In the following experiments, we consistently used fixed values for $\epsilon_m$ and $\epsilon_s$ as 10% and 15% respectively.

**Force computation:** A (repulsive or attractive) force from $p_j$ to $p_i$, $f_{i \leftarrow j}$, is defined as the following (Fig. 3):

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

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

$$f_i = \frac{1}{c} \sum_{j=1}^{c} f_{i \leftarrow j}.$$

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.

The time complexity of our iterative process is $O((c^2 + cD)m)$, which is comparable to those of the state-of-the-art techniques (e.g., $O(D^2 m)$ of spectral hashing [26]). In practice, our iterative process is finished within 10 to 30 iterations. Also, its overall computation time is less than 30 seconds even for 128 code lengths. The convergence rate with respect to the number of iterations is shown in Fig. 4. Note that our iterative optimization process shares similar characteristics of the N-body simulation [9] designed for simulating various dynamic systems of particles (e.g., celestial objects interacting with each other under gravitational forces). Efficient numerical integration methods (e.g., fast multipole method) can be applied to accelerate our iterative optimization process.

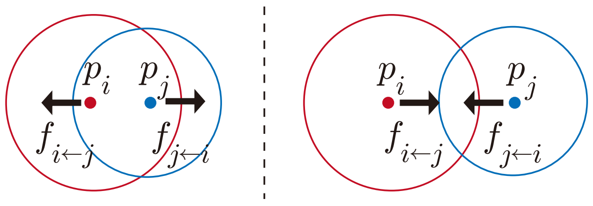

Figure 3. These two images show how a force between two pivots is computed. In the left image a repulsive force is computed since their overlap $o_{i,j}$ is larger than the desired amount. On the other hand, the attractive force is computed in the right image because their overlap is smaller than the desired amount.

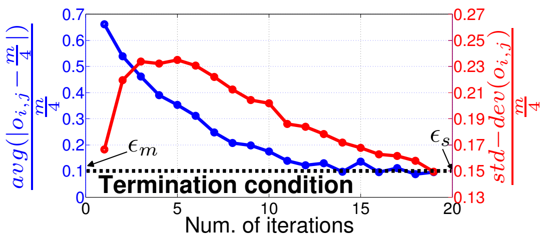

Figure 4. This graph shows a convergence rate of our iterative optimization. The left and right $y$-axises indicate how the average and std. dev. of $o_{i,j}$ approach our termination condition in the scale of our error tolerances, $\epsilon_m$ and $\epsilon_s$ respectively. In this case, we set $\epsilon_m$ as 0.1 and $\epsilon_s$ as 0.15 for terminating our optimization. This result is obtained with the **GIST-1M-384D** dataset at the 64-bit code length.

## 4. Evaluation

In this section we evaluate our method and compare it with the state-of-the-art methods [3, 26, 7, 10, 15, 20].

## 4.1. Datasets and Protocol

We perform various experiments with the following three datasets:

- **GIST-1M-384D**: A set of 384 dimensional, one million GIST descriptors, which consist of a subset of Tiny Images [23].
- **GIST-1M-960D**: A set of 960 dimensional, one million GIST descriptors that are also used in [14].
- **GIST-75M-384D**: A set of 384 dimensional, 75 million GIST descriptors, which consist of a subset of 80 million Tiny Images [23].

Our evaluation protocol follows that of [15]. Specifically, we test with randomly chosen 1000 queries for datasets **GIST-1M-384D** and **GIST-1M-960D**, 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.

For experiments with **GIST-1M-384D** and **GIST-1M-960D**, we use a machine consisting of i7 X990 with 24GB main memory. For **GIST-75M-384D**, we use a machine consisting of Xeon X5690 and 144GB main memory to hold all the data in its main memory.

## 4.2. Compared Methods

- **LSH** and **LSH-ZC**: Locality Sensitive Hashing [3] with/without Zero Centered data points. The projection matrix is a Gaussian random matrix. As discussed in [8, 7], centering the data around the origin (*i.e.* $\sum x_i = 0$) produces much better results over **LSH**. Hence, we transform data points such that their center is located at the origin for **LSH-ZC**.
- **LSBC**: Locality Sensitive Binary Codes [20]. The bandwidth parameter used in experiment is the inverse of the mean distance between the points in the dataset, as suggested in [8].
- **SpecH**: Spectral Hashing [26].
- **PCA-ITQ**: Iterative Quantization [7].
- **RMMH-L2**: Random Maximum Margin Hashing (RMMH) [15] 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 [15].

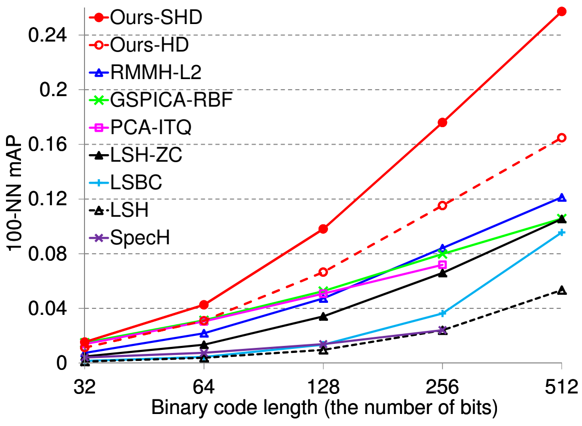

Figure 5. Comparisons between our method and the-state-of-the-art methods with the **GIST-1M-384D** dataset when $k = 100$.

- **GSPICA-RBF**: Generalized Similarity Preserving Independent Component Analysis (GSPICA) [10] with the RBF kernel. We experiment GSPICA with the RBF kernel, since the authors reported the best performance on $k$ nearest neighbor search with this kernel. The parameter used in the RBF kernel is determined by the mean distance of $k$th nearest neighbors within training samples as suggested by [15]. The parameters $\gamma$ and $P$ are 1 and the dimensionality of the dataset respectively, as suggested in [10].
- **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.

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.

## 4.3. Results

Fig. 5 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

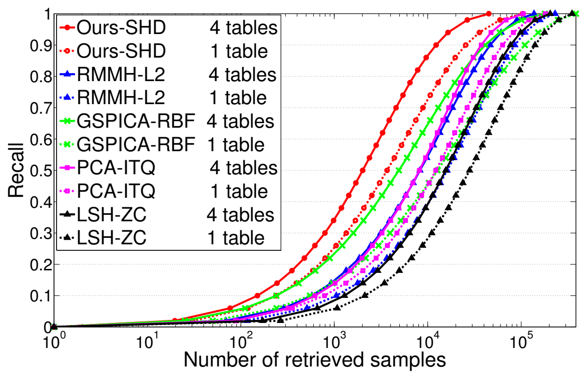

Figure 6. Recall curves of different methods when $k = 100$ for the **GIST-1M-384D** dataset. Each hash table is constructed by 64 bits code lengths. The recall ($y$-axis) represents search accuracy and the number of retrieved samples ($x$-axis) represents search time. Graphs are best viewed with colors.

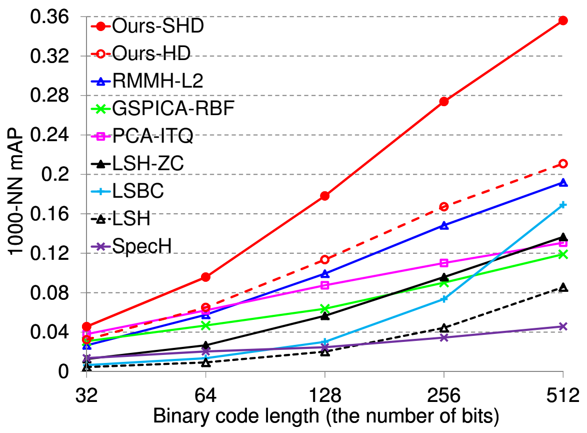

Figure 7. Comparison between our method and the-state-of-the-art methods with the **GIST-1M-960D** dataset when $k = 1,000$.

can effectively create closed regions with tighter distance bounds compared to hyperplanes.

Given 0.1 mAP in Fig. 5, 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 [15]. Once we identify nearest neighbor images based on binary codes, we can employ additional re-ranking processes on those images. As pointed out in [15], 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. 5. 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 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 dataset. Fig. 6 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. For example, when we aim to achieve 0.5 recall rate, our method with one and four hash tables needs 3674 and 2013 images on average respectively. However, **GSPICA-RBF** [10] with four hash tables, the second-best method, needs to identify 4909 images, which are 33% and 143% more number of images than our method with one and four hash tables respectively.

We have also performed all the tests against the 960 dimensional, one million GIST dataset **GIST-1M-960D** with $k = 1000$ (Fig. 7). We have found that our method shows similar trends even with this dataset, compared to what we have achieved in **GIST-1M-384D**. The precision-recall curves corresponding to Fig. 5 and Fig. 7 are available in the supplementary report.

We have also performed all the tests against the 384 dimensional, 75 million GIST dataset **GIST-75M-384D** with $k = 10,000$ (Fig. 8). 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.

In order to see how each component of our method affects the accuracy, we measure mAP by disabling the spherical Hamming distance, the independent constraint, and balanced partitioning in our method. 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.

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 technique generating binary codes based on hyperplanes.

## 4.4. Discussions

As we briefly pointed out in Sec. 1, hyperplanes in a kernel space can map to non-linear hashing functions. However, previous techniques using kernel spaces [17, 10, 15] define their kernels with randomly chosen multiple landmarks from the input data points. In other words, those multiple landmarks affect to determine each bit in their binary codes. As a result, their kernels map to very complicated

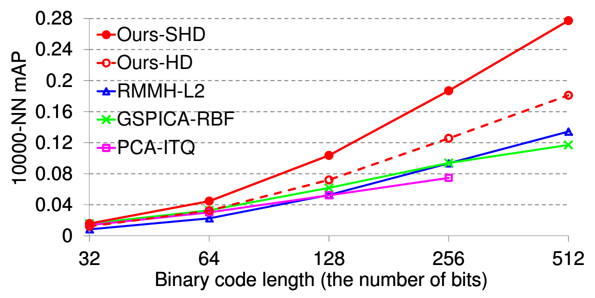

Figure 8. Comparison between our method and the-state-of-the-art methods with the **GIST-75M-384D** dataset when $k = 10,000$.

non-linear functions, which are drastically different from hyperspheres (see the rightmost image of Fig. 1). Instead our method is intuitive and simple, as it uses only the single pivot of a hypersphere for each bit.

Furthermore, we choose pivots carefully to satisfy the independence constraint, while other methods [10, 15] use randomly chosen landmarks. Also, to define binary codes, **LSBC** [20] uses shift-invariant kernels. However, **LSBC** uses kernels in a different way from our approach and does not use hyperspheres defined in the original data space to encode binary codes. In summary, according to the best of our knowledge, using hyperspheres in the original data space has never been employed to encode binary codes for computer vision applications.

## 5. Conclusion

We have proposed a novel hypersphere-based binary embedding technique for providing compact data representation and highly scalable nearest neighbor search with high accuracy. Our method significantly outperformed the tested six state-of-the-art hashing techniques based on hyperplanes with one and 75 million GIST descriptors that have 384 or 960 dimensions. In our future work, we would like to incorporate the quantization error that is considered in the iterative quantization method [7] into our optimization process. We expect that by doing so, we can improve the search accuracy further.

**Acknowledgements.** This work was supported in part by MCST/KOCCA/CT/R&D 2011, MKE/KEIT [KI001810035261], MKE/MCST/IITA [2008-F-033-02], BK, DAPA/ADD (UD110006MD), MEST/NRF/WCU (R31-2010-000-30007-0), KMCC, MSRA, and MEST/NRF (2011-0030822).

## References

[1] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[2] O. Chum, J. Philbin, and A. Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *BMVC*, 2008.

[3] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, 2004.

[4] R. Datta, D. Joshi, J. Li, and J. Z. Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Survey*, 40(2):1–60, 2008.

[5] R. F. S. Filho, A. Traina, C. J. Traina., and C. Faloutsos. Similarity search without tears: the omni-family of all-purpose access methods. In *Int'l Conf. on Data Engineering*, 2001.

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

[7] Y. Gong and S. Lazebnik. Iterative quantization: a procrustean approach to learning binary codes. In *CVPR*, 2011.

[8] A. Gordo and F. Perronnin. Asymmetric distances for binary embeddings. In *CVPR*, 2011.

[9] L. Greengard. *The rapid evaluation of potential fields in particle systems*. MIT Press, Cambridge, 1988.

[10] J. He, R. Radhakrishnan, S.-F. Chang, and C. Bauer. Compact hashing with joint optimization of search accuracy and time. In *CVPR*, 2011.

[11] P. Indyk and R. Motwani. Approximate nearest neighbors: toward removing the curse of dimensionality. In *STOC*, 1998.

[12] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive $b^+$-tree based indexing method for nearest neighbor search. *ACM T. on Database Systems*, 2005.

[13] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *CVPR*, 2008.

[14] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE TPAMI*, 2011.

[15] A. Joly and O. Buisson. Random maximum margin hashing. In *CVPR*, 2011.

[16] 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. Technical report, KAIST, 2012.

[17] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, 2009.

[18] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *CVPR*, 2006.

[19] A. Oliva and A. Torralba. Modeling the shape of the scene: a holistic representation of the spatial envelope. *IJCV*, 2001.

[20] M. Raginsky and S. Lazebnik. Locality sensitive binary codes from shift-invariant kernels. In *NIPS*, 2009.

[21] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.

[22] K. Terasawa and Y. Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *Algorithms and Data Structures*, volume 4619, pages 27–38, 2007.

[23] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *CVPR*, 2008.

[24] J. Venkateswaran, D. Lachwani, T. Kahveci, and C. Jermaine. Reference-based indexing of sequence databases. In *VLDB*, 2006.

[25] J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for scalable image retrieval. In *CVPR*, 2010.

[26] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, 2008.

[27] A. M. Yaglom and I. M. Yaglom. *Challenging Mathematical Problems with Elementary Solutions*. New York: Dover Pub., Inc., 1987.