2012 IEEE 28th International Conference on Data Engineering

Bi-level Locality Sensitive Hashing for k-Nearest Neighbor Computation

Jia Pan and Dinesh Manocha
{panj, dm}@cs.unc.edu
http://gamma.cs.unc.edu/KNN
University of North Carolina at Chapel Hill

***Abstract*—We present a new Bi-level LSH algorithm to perform approximate k-nearest neighbor (KNN) search in high dimensional spaces. Our formulation is based on a two-level scheme. In the first level, we use a RP-tree that partitions the dataset into sub-groups with bounded aspect ratios and is used to compute well-separated clusters. During the second level, we compute a single LSH hash table for each sub-group along with a hierarchical structure based on space-filling curves. Given a query, we first determine the appropriate sub-group and perform k-nearest neighbor search within the suitable buckets in the LSH hash table corresponding to the sub-group. Our algorithm also maps well to current GPU architectures and can improve the quality of approximate KNN queries as compared to prior LSH-based algorithms. We perform extensive experimental analysis on two large, high-dimensional image datasets. Given a runtime budget, our experimental results indicate that Bi-level LSH can provide better accuracy in terms of recall or error ratios.**

I. Introduction

Nearest neighbor search in high-dimensional space is an important problem in database management, data mining, computer vision and WWW-based search engines. The underlying applications use feature-rich data, such as digital audio, images or video, which are typically represented as high-dimensional feature vectors. One popular way to perform similarity searches on these datasets is via an exact or approximate k-nearest neighbor (KNN) search in a high-dimensional feature space. This problem is well studied in literature, and is regarded as a challenging problem due to its intrinsic complexity and the quality or accuracy issues that arise in terms of computing the appropriate k-nearest neighbors.

In terms of runtime cost, ideal nearest neighbor query should take $\mathcal{O}(1)$ or $\mathcal{O}(\lg n)$ per-query time, because the size of the dataset (i.e. n) can be very large (e.g. > 1 million). The sub-linear timing for each query is important for many applications such as mobile GIS systems [1], where the number of queries per second can be very high. In such cases, a linear per-query timing results in high runtime complexity, which can be rather slow for many applications. Moreover, the space required should be $\mathcal{O}(n)$ in order to handle large datasets. In terms of quality issues, each query should return k-nearest neighbor results that are close enough to the exact k-nearest neighbors computed via a brute-force, linear-scan approach that has a high $O(n)$ per-query complexity.

Most current approaches for k-nearest neighbor computation are unable to satisfy the runtime requirements for high-dimensional datasets. For example, tree-based methods such as cover-tree [2], SR-tree [3] can compute accurate results, but are not time-efficient for high-dimensional datasets. When the dimensionality exceeds 10, these space partitioning-based methods can be slower than the brute-force approach [4]. Approximate nearest neighbor algorithms tend to compute neighbors that are close enough to the queries instead of the exact k-nearest neighbors, and have a lower runtime and memory overhead than the exact algorithms [5]. For high-dimensional k-nearest neighbor search, one of the widely used approximate methods is *locality-sensitive hashing* (LSH) [6], which uses a family of hash functions to group or collect nearby items into the same bucket with a high probability. In order to perform a similarity query, LSH-based algorithms hash the query item into one bucket and use the data items within that bucket as potential candidates for the final results. Moreover, the items in the bucket are ranked according to the exact distance to the query item in order to compute the k-nearest neighbors. The final ranking computation among the candidates is called *short-list search*, which is regarded as the main bottleneck in LSH-based algorithms and takes 95% or more of the overall running time. In order to achieve high search accuracy, LSH-based methods use multiple (e.g. more than 100 [7]) hash tables, which results in high space and time complexity. To reduce the number of hash tables, some LSH variations tend to use more candidates [8], estimate optimal parameters [9], [10], or use improved hash functions [11], [12], [13], [14]. All these methods only consider the average runtime or quality of the search process over randomly sampled hash functions, which may result in large deviations in the runtime cost or the quality of k-nearest neighbor results.

Main Results: In this paper, we present a novel bi-level scheme based on the LSH framework. Our formulation offers improved runtime and quality as compared to prior LSH-based algorithms. We use a two-level scheme: During the first level, we use random projections as a preprocess to divide the given dataset into multiple subgroups with bounded aspect ratios (i.e. roundness) [15] and can distinguish well-separated clusters. In practice, this preprocessing step tends to decrease the deviation in the runtime and quality due to the randomness of LSH framework and the non-uniform distribution of data items. During the second level, we apply standard [6] or multiprobe [8] LSH techniques to each subgroup and compute approximate k-nearest neighbors. In order to reduce the runtime and quality deviation caused by different queries, we construct a hierarchical LSH table based on Morton curves. This hierarchy is also enhanced by lattice techniques to provide better k-nearest neighbor candidates for LSH short-list search. We have applied our algorithm to two large, high-dimensional, widely-used image datasets and compared its performance with prior LSH-based methods. In practice, we observe improvements in selectivity, recall ratio and error ratio. Moreover, our method can reduce the runtime and quality deviation caused by random projection on different queries. Furthermore, our Bi-level LSH algorithm maps well to current GPU architectures [16], and can offer up to 40X speedup over a single-core CPU-based implementation.

The rest of the paper is organized as follows. We survey the background of LSH and related work in Section II. Section III gives an overview of our approach. We present our new Bi-level LSH algorithm in Section IV and describe its parallel implementation on current GPUs in Section V. We highlight the performance on different benchmarks in Section VI.

II. Background and Related Work

In this section, we briefly review related work on LSH based k-nearest neighbor computation.

A. Basic LSH

Given a metric space $(\mathbb{X}, \|\cdot\|)$ and a database $S \subseteq \mathbb{X}$, for any given query $\mathbf{v} \in \mathbb{X}$, the k-nearest neighbor algorithm computes a set of k points $I(\mathbf{v}) \subseteq S$ that are closest to $\mathbf{v}$. We assume that $\mathbb{X}$ is embedded in a D-dimensional Euclidean space $\mathbb{R}^D$ and each item is represented as a high-dimensional vector, i.e. $\mathbf{v} = (v_1, ..., v_D)$.

The basic LSH algorithm is an approximate method to compute k-nearest neighbors, which uses a M-dimensional hash function $H(\cdot)$ to transform $\mathbb{R}^D$ into a lattice space $\mathbb{Z}^M$ and distributes each data item into one lattice cell:

$$H(\mathbf{v}) = \langle h_1(\mathbf{v}), h_2(\mathbf{v}), ..., h_M(\mathbf{v})\rangle, \quad (1)$$

where $h_i(\cdot)$ is a 1-dimensional hash function. The lattice space is usually implemented as a hash table, since many of the cells may be empty. LSH algorithms have been developed for several distance measures, such as l_p distance. For l_p space, $p \in (0, 2]$ [6],

$$h_i(\mathbf{v}) = \lfloor \frac{\mathbf{a}_i \cdot \mathbf{v} + b_i}{W} \rfloor, \quad (2)$$

where the D-dimensional vector $\mathbf{a}_i$ consists of i.i.d. entries from Gaussian distribution $N(0, 1)$ and b_i is drawn from a uniform distribution $U[0, W)$. M and W control the dimension and size of each lattice cell and therefore control the locality sensitivity of the hash functions. In order to achieve higher accuracy for approximate KNN queries, L hash tables are used and each of them is constructed independently with different dim-M hash functions $H(\cdot)$. Given a query item $\mathbf{v}$, we first compute its hash code using $H(\mathbf{v})$ and locate the hash bucket that contains $\mathbf{v}$. All the points in the bucket will belong to its potential k-nearest neighbor candidate set and we represent that set as $A(\mathbf{v})$. Next, we perform a local scan on $A(\mathbf{v})$ to compute the k-nearest neighbors $I(\mathbf{v})$. This step is called short-list search and is the main bottleneck of the LSH framework.

There are several known metrics used to measure the performance of a k-nearest neighbor search algorithm. First is the *recall ratio*, i.e. the percentage of the exact k-nearest neighbors $N(\mathbf{v})$ in the returned results $I(\mathbf{v})$:

$$\rho(\mathbf{v}) = \frac{|N(\mathbf{v}) \cap I(\mathbf{v})|}{|N(\mathbf{v})|} = \frac{|N(\mathbf{v}) \cap A(\mathbf{v})|}{|N(\mathbf{v})|}, \quad (3)$$

where $N(\mathbf{v})$ can be computed using any exact k-nearest neighbor approach and serves as the ground-truth.

The second metric is the *error ratio* [7], i.e. the relationship between $\mathbf{v}$'s distance to $N(\mathbf{v})$ and $I(\mathbf{v})$:

$$\kappa(\mathbf{v}) = \frac{1}{k}\sum_{i=1}^{k} \frac{\|\mathbf{v} - N(\mathbf{v})_i\|}{\|\mathbf{v} - I(\mathbf{v})_i\|}, \quad (4)$$

where $N(\mathbf{v})_i$ or $I(\mathbf{v})_i$ is $\mathbf{v}$'s i-th nearest neighbor in $N(\mathbf{v})$ or $I(\mathbf{v})$. We use recall and error ratios to measure the quality of the LSH algorithm; our goal is to compute the k-nearest neighbors with large recall and error ratios close to 1.

The final metric is the *selectivity* [10]:

$$\tau(\mathbf{v}) = |A(\mathbf{v})|/|S|, \quad (5)$$

where $|S|$ is the size of the dataset. Selectivity is also a rough measurement for the runtime cost of the short-list search because the time to compute k maximum elements from the set $A(\mathbf{v})$ is a function of the size of the set. According to Knuth [17], the complexity of selecting k maximum among $|A(\mathbf{v})|$ elements is $O(|A(\mathbf{v})| + k)$.

B. Variations of LSH

Many techniques have been proposed to improve the basic LSH algorithm. LSH-forest [9] avoids tuning of the parameter M by representing the hash table as a prefix tree and the parameter M is computed based on the depth of the corresponding prefix-tree leaf node. Multi-probe LSH [8] systematically probes the buckets near the query points in a query-dependent manner, instead of only probing the bucket that contains the query point. It can obtain a higher recall ratio with fewer hash tables, but may result in larger selectivity from additional probes. Dong et al. [10] construct a statistical quality and runtime model using a small sample dataset, and then compute M and W that can result in a good balance between high recall ratio and low selectivity. Joly et al. [18] improve multi-probe LSH by using prior information collected from a sampled dataset.

Many approaches have been proposed to design better hash functions. $\mathbb{Z}^M$ lattice may suffer from the curse of dimensionality: in a high dimensional space, the *density* of $\mathbb{Z}^M$ lattice, i.e. the ratio between the volume of one $\mathbb{Z}^M$ cell and the volume of its inscribed sphere, increases very quickly when the dimensionality increases. In order to overcome these problems, lattices with densities close to one are used as space quantizers, e.g. E_8-lattice [12] and Leech lattice [11] are used for dim-8 and dim-24 data items, respectively.

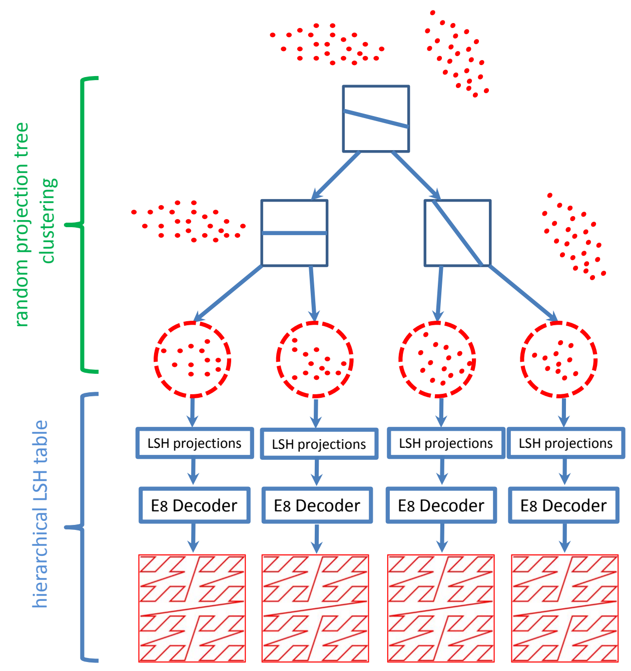


Fig. 1. The framework for Bi-level LSH. The first level is the random projection tree. The second level is the hierarchical lattice for each RP-Tree leaf node.

III. Overview

In this section, we give an overview of our new Bi-level LSH scheme.

The overall pipeline of our algorithm is shown in Figure 1 and includes two levels. In the first level, we construct a *random projection tree* (RP-tree) [19], [20], which is a space-partitioning data structure that is used to partition the points into several subsets. In a RP-tree, each subset is represented as one leaf node. As compared to other methods such as Kd-tree and K-means algorithms, RP-tree has many good properties [20], [15]. These include fast convergence speed, guaranteed 'roundness' of leaf nodes, etc. These properties are useful for generating compact LSH hash code and reduce the algorithm's performance/quality variance.

During the second level, we compute a locality-sensitive hash (LSH) table for each of the subsets generated during the first level. Unlike prior LSH methods, the LSH table in our bi-level algorithm has a hierarchical structure that is computed using a Morton curve. The hierarchal LSH table can reduce the runtime performance and quality variance among different queries: for a query that lies in a region with high data density, our algorithm only needs to search nearby buckets; for a query within a region with low data density, our algorithm can automatically search in far away buckets to compute high quality KNN. We also enhance the hierarchy using a E_8 lattice, which can overcome $\mathbb{Z}^M$ lattice's drawbacks for high-dimensional datasets and can improve the quality of the results.

For each item $\mathbf{v}$ in the dataset, our method computes the RP-tree leaf node RP-tree($\mathbf{v}$) that contains $\mathbf{v}$ and computes its LSH code $H(\mathbf{v})$ using LSH parameters for that subset in the second level. As a result, our Bi-level LSH scheme decomposes the basic LSH code into two parts: the RP-tree leaf node index and the LSH code corresponding to the subset, i.e. $\tilde{H}(\mathbf{v}) = (\text{RP-tree}(\mathbf{v}), H(\mathbf{v}))$. $\tilde{H}(\mathbf{v})$ is stored in a hash table.

Given a KNN query, we first traverse the RP-tree to compute the leaf node that the query belongs to and then compute the LSH code within that subset. Based on the decomposed hash code, we find the buckets in the hash table with the same or similar hash codes as the query, and then the elements in these buckets are used as candidates for k-nearest neighbors in the short-list search.

IV. Bi-level Locality-Sensitive Hashing

In this section, we present the details of our Bi-level LSH algorithm.

A. Level I: RP-tree

During the first level, we use a *random projection tree* (RP-tree) [19], [20] to divide the dataset into several small clusters with good properties for subsequent LSH-based operations. We first give a brief overview of RP-trees and present the details of our first-level algorithm. Finally, we analyze the advantages of our first-level preprocessing scheme within the overall Bi-level LSH framework.

1) Background: The RP-tree construction algorithm is similar to Kd-tree computation: given a data set, we use a split rule to divide it into two subsets, which are split recursively until the resulting tree has a desired depth. Kd-tree chooses a coordinate direction (typically the coordinate with the largest spread) and then splits the data according to the median value for that coordinate. Unlike Kd-tree, RP-tree projects the data onto a randomly chosen unit direction and then splits the set into two roughly equal-sized sets using new split rules. Dasgupta and Freund [20] have proposed two rules for RP-trees: RP-tree max and RP-tree mean. The difference between the two rules is that RP-tree mean occasionally performs a different kind of split based on the distance from the mean of the coordinates. RP-tree has been used in many other applications as a better alternative to K-means [21].

2) Tree Construction: We use RP-tree max or RP-tree mean rule to partition the input large dataset into many small clusters. In practice, we observe that RP-tree mean rule computes better results in terms of recall ratio of the overall bi-level scheme as compared to RP-tree max rule.

In order to apply the RP-tree mean rule, we need to efficiently compute $\Delta(S)$, the diameter for a given point set S in a high-dimensional space. However, the complexity of computing the exact diameter is as high as exact k-nearest neighbor query. Fortunately, there exist algorithms that can approximate the diameter efficiently. We use the iterative method proposed by Egecioglu and Kalantari [22], which converges fast to a diameter approximation with good precision. The diameter computation algorithm uses a series of m values $r_1, ..., r_m$ to approximate the diameter for a point set S, where $m \leq |S|$. It can be shown that $r_1 < r_2 < ... < r_m \leq \Delta(S) \leq \min(\sqrt{3}r_1, \sqrt{5-2\sqrt{3}r_m})$ [22]. In practice, we find that r_m is usually a good enough approximation of $\Delta(S)$ even when m is small (e.g. 40). The time complexity of the resulting approximate diameter algorithm is $O(m|S|)$, and we can construct each level of the RP-tree in linear time for the size of the entire dataset. As a result, the overall complexity to partition the dataset into g groups is $O(\log(g)n)$.

3) Analysis: We show that our RP-construction algorithm can improve the quality of KNN queries and reduce the performance variation, over the basic LSH algorithm.

One of our main goals to use RP-tree is to enable our LSH scheme to adapt to datasets with different distributions and to obtain higher quality and lower runtime cost. The RP-tree partitions a dataset into several leaf nodes so that each leaf node only contains similar data items, e.g. images for the same or similar objects, which are likely to be drawn from the same distribution. During the second level of our Bi-level LSH scheme, we may choose different LSH parameters (e.g. bucket size W) that are optimal for each cell instead of choosing a single set of parameters for the entire dataset. Such parameter choosing strategy can better capture the interior differences within a large dataset.

Kd-tree and K-means can also perform partitioning on data items, but RP-tree has some advantages over these methods for high-dimensional datasets. An ideal partitioning algorithm should have fast convergence speed and should be able to adapt to data's intrinsic dimension. In our case, the convergence speed is measured by the number of tree levels needed to halve the radius of one RP-tree node. Moreover, the intrinsic dimension represents the fact that usually the data items typically lie on a low-dimensional submanifold embedded in a high dimensional Euclidean space. For a dataset with intrinsic dimension d, which is usually much smaller than D, it is known that in the worst case, the Kd-tree would require $\mathcal{O}(D)$ levels so as to halve the radius of cells in $\mathbb{R}^D$ even when $d \ll D$, which is slow. K-means is another data-partitioning approach, but is sensitive to parameter initialization and may be slow in terms of converging to a good solution.

As compared to these methods, RP-tree has guaranteed and faster convergence speed and can adapt to intrinsic dimension automatically, according to the properties related to RP-tree max and RP-tree mean [20], [15]. Intuitively, RP-tree max property implies faster convergence in terms of worst-case improvement amortized over levels while RP-tree mean property implies quick convergence in terms of average improvement during each level.

Another main advantage of RP-tree is that it can reduce the performance and quality variation of LSH algorithms. Previous statistical models related to LSH, such as [8], [10], [18], are based on optimizing the average performance and quality over all random sampled projections. However, the actual performance and quality of the resulting algorithms may deviate from the mean values according to the magnitude of the variance caused by random projections, which is related to the 'shape' of the dataset. Let us consider the LSH scheme on l_p (i.e. Equation (2)) as an example and we can show that the shape of the dataset will influence the variance in the performance and quality of the LSH algorithm. As shown in Figure 2(a), if the dataset shape is flat, then the projection along the long axis will result in a hash function with small recall ratio but also small selectivity (i.e. small runtime cost) while the projection along the short axis will result in a hash function with large recall ratio but large selectivity (i.e. large runtime cost). It is hard to compute a value of W that is suitable for all the projections in terms of both quality and runtime cost. Therefore, we can have large variance in the performance or quality. Instead, if the shape is nearly 'round', as shown in Figure 2(b), then the recall ratio and the runtime cost would be similar along all the projection directions and the deviation in overall performance or quality is small. As a result, LSH algorithms would work better on datasets with a 'round' shape or with bounded aspect ratio. It has been shown that RP-tree max rule can partition a dataset into subsets with bounded aspect ratios and therefore, reduces the performance variance of LSH algorithms [15].

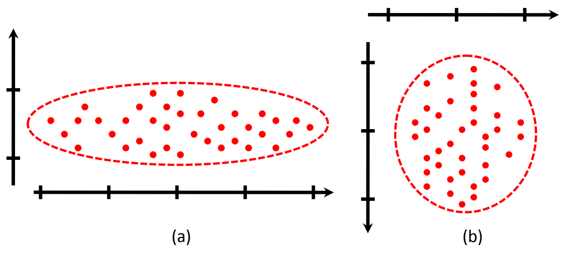


Fig. 2. LSH scheme behaves better on dataset with bounded aspect ratio. (a) The data has a large aspect ratio and there is no bucket size W that is suitable for all random projections. (b) The dataset has a shape close to a sphere and W chosen for one projection is usually suitable for all the projections.

B. Level II: Hierarchical LSH

In the second level of the bi-level algorithm, we construct LSH hash table for each leaf node cell calculated during the first level. Before that, we use an automatic parameter tuning approach [10] to compute the optimal LSH parameters for each cell. The RP-tree computed during the first level has already partitioned the dataset into clusters with nearly homogeneous properties. As a result, the estimated parameters for each cluster can improve the runtime performance and quality of the LSH scheme, as compared to the single set of parameters used for the entire dataset. The LSH code of one item and its leaf node index composes its Bi-level LSH code, i.e. $\tilde{H}(\mathbf{v}) = (\text{RP-tree}(\mathbf{v}), H(\mathbf{v}))$.

1) Basic LSH Table: The LSH table is a hash table storing the LSH hash codes for all items in the database. Each bucket in the hash table stores the database elements with the same LSH hash code. Unlike common hash tables trying to decrease the number of hash collisions, LSH hash table tries to merge nearby elements in the same bucket. A single bucket is usually implemented as a data structure such as a linked list or a self-balancing tree, which is efficient for operations like element lookup. The short-list search then traverses the data structure to find the k elements closest to the given query.

2) Hierarchical LSH Table: In order to make LSH approach adaptive to different queries, we further enhance the LSH table with a hierarchical structure. When the given query lies in a region that has low data density (e.g. a region between two clusters), basic LSH or multi-probe techniques may have only a few hits in the LSH hash table. Therefore, there may not be sufficient number of neighborhood candidates for a given k-nearest neighbor query. As a result, the recall ratio could be small. In such cases, the algorithm should automatically search the far away buckets or larger buckets so as to improve multi-probe's efficiency and ensure that there are sufficient number of neighborhood candidates. Our solution is to construct a hierarchy on the buckets. Given a query code, we can determine the hierarchical level that it lies in and use those buckets for the probing algorithm.

a) Hierarchical LSH Table for $\mathbb{Z}^M$ Lattice: For the $\mathbb{Z}^M$ lattice, we implement the hierarchy using a space filling Morton curve, also known as the Lebesgue or Z-order curve. We first compute the Morton code for each unique LSH code by interleaving the binary representations of its coordinate value and then sort the Morton codes to generate a one-dimensional curve. The Morton curve maps the multi-dimensional data to a single dimension and tries to maintain the neighborhood relationship, i.e. nearby points in the high-dimensional space are likely to be close on the one-dimensional curve. Conceptually, the Morton curve can be constructed by recursively dividing dim-D cube into 2^D cubes and then ordering the cubes, until at most one point resides in each cube. Note that the Morton code has one-to-one relationship with the LSH code, so it can also be used as the key to refer to the buckets in the LSH hash table.

Lets assume that we have constructed the LSH Morton curve for the given dataset. Given one query item, we first compute its LSH code and the corresponding Morton code. Next, we search within the Morton curve to find the position where the query code can be inserted without violating the ordering. Finally, we use the Morton codes before and after the insert position in the Morton curve to refer the buckets in the LSH hash table for short-list search. As the Morton curve cannot completely maintain the neighborhood relationship in high-dimensional space, we need to perturb some bits of the query Morton code and repeat this process several times [23]. We can also compute the number of most significant bits (MSB) shared by query Morton code and the Morton codes before and after the insert position. When the shared MSB number is small, we should traverse to a higher level in the hierarchy and use a larger bucket, which is implemented as buckets with the same MSB bits.

To overcome the curse of dimensionality caused by basic $\mathbb{Z}^M$ lattice, we improve the hierarchical LSH table constructing using lattice techniques. The drawback of $\mathbb{Z}^M$ lattice is that it has low density in high-dimensional spaces and cannot provide high quality nearest neighbor candidates for a given query: most candidates inside the same bucket with the query item may be far from it and many of the actual nearest neighbors may be outside the bucket. As a result, we may need to probe many cells to guarantee large enough recall ratio, which implies large selectivity. One solution to this drawback is using E_8 lattice [12] which has the maximum density in dim-8 space. Intuitively, E_8 lattice is closest to dim-8 sphere in shape and thus can provide high quality nearest neighbor candidates.

b) Hierarchical LSH Table for E_8 Lattice: The E_8 lattice is a collection of points in dim-8 space, whose coordinates are all integers or are all half-integers and the sum of the eight coordinates is an even integer, e.g., $(1)^8 \in E_8$, $(0.5)^8 \in E_8$ but $(0,1,1,1,1,1,1,1) \notin E_8$. The collection of all integer points, whose sums are even, is called the D_8 lattice and we have $E_8 = D_8 \cup (D_8 + (\frac{1}{2})^8)$. One main difference between $\mathbb{Z}^8$ and E_8 lattice is that each E_8 lattice node has 240 neighbors with the same distance to it and requires a different multi-probe and hierarchy construction strategy as compared to $\mathbb{Z}^M$ lattice.

For convenience, we first assume the space dimension to be 8. Then given a point $\mathbf{v} \in \mathbb{R}^8$, its E_8 LSH code is

$$H_{E_8}(\mathbf{v}) = \textsc{Decode}\big([\frac{\mathbf{a}_1 \cdot \mathbf{v} + b_1}{W}, \frac{\mathbf{a}_2 \cdot \mathbf{v} + b_2}{W} ..., \frac{\mathbf{a}_8 \cdot \mathbf{v} + b_8}{W}]\big). \tag{6}$$

That is, we replace the floor function in basic LSH hash function by the E_8 decode function $\textsc{Decode}(\cdot)$, which maps the vector in $\mathbb{R}^8$ onto E_8 lattice points. There exists an efficient implementation of the E_8 decoder [12] with only 104 operations.

The multi-probe computation for the E_8 LSH algorithm is performed in the following manner. Given a query, we first probe the bucket that it lies in and then compute the 240 buckets closest to it. The probe sequence is decided by the distance of query item to the 240 E_8 lattice nodes, similar to the case in $\mathbb{Z}^8$ [8]. If the number of candidates computed is not enough, we recursively probe the adjacent buckets of the 240 probed buckets.

The 240 neighborhood property makes it more complicated to construct a hierarchy for the E_8 lattice. Morton code is not suitable for the E_8 lattice because it needs orthogonal lattices like $\mathbb{Z}^M$. However, Morton curve can also be viewed as an extension of octree in $\mathbb{Z}^M$: partition a cube by recursively subdividing into 2^M smaller cubes. The subdivision is feasible because $\mathbb{Z}^M$ has the *scaling* property: the integer scaling of $\mathbb{Z}^M$ lattice is still a valid $\mathbb{Z}^M$ lattice. Suppose $\mathbf{c} = (c_1, ..., c_M) = H(\mathbf{v}) \in \mathbb{R}^M$ is the traditional $\mathbb{Z}^M$ LSH code for a given point $\mathbf{v}$, then its k-th ancestor in the LSH code hierarchy is

$$H^k(\mathbf{v}) = 2^k\big(\underbrace{\lfloor\frac{1}{2}\cdots\lfloor\frac{1}{2}\lfloor}_{k}\frac{c_1}{2}\rfloor\rfloor\ldots\rfloor, ..., \underbrace{\lfloor\frac{1}{2}\cdots\lfloor\frac{1}{2}\lfloor}_{k}\frac{c_M}{2}\rfloor\rfloor\ldots\rfloor\big) \tag{7}$$

$$= 2^k\big(\lfloor\frac{c_1}{2^k}\rfloor, ..., \lfloor\frac{c_M}{2^k}\rfloor\big), \tag{8}$$

where $k \in \mathbb{Z}^+$ and $H^0(\mathbf{v}) = H(\mathbf{v})$. The second equality is due to

$$\lfloor\frac{1}{n}\lfloor\frac{x}{m}\rfloor\rfloor = \lfloor\frac{x}{mn}\rfloor \tag{9}$$

for all $m, n \in \mathbb{Z}$ and $x \in \mathbb{R}$. E_8 lattice also has the scaling property, so we can construct the hierarchy in E_8 by defining the k-th ancestor of a point $\mathbf{v} \in \mathbb{R}^8$ as

$$\begin{aligned} H^k_{E_8}(\mathbf{v}) =& 2^k\big(\underbrace{\textsc{Decode}(\frac{1}{2}\ldots\textsc{Decode}(\frac{1}{2}\textsc{Decode}(\frac{c_1}{2}))\ldots)}_{k}, \\ & \ldots, \underbrace{\textsc{Decode}(\frac{1}{2}\ldots\textsc{Decode}(\frac{1}{2}\textsc{Decode}(\frac{c_M}{2}))\ldots)}_{k}\big) \end{aligned} \tag{10}$$

where $k \in \mathbb{Z}^+$ and $\mathbf{c} = H^0_{E_8}(\mathbf{v}) = H_{E_8}(\mathbf{v})$. The decode function does not satisfy the same property as Equation (9) for the floor function, so we cannot simplify it to have a form similar to Equation (8).

There is no compact representation of E_8 LSH hierarchy similar to the Morton curve for $\mathbb{Z}^M$ lattice. We implement the E_8 hierarchy as a linear array along with an index hierarchy:

1) We generate a set by selecting one point from each E_8 bucket. The linear array starts with the E_8 LSH codes $H^m_{E_8}(\mathbf{v})$ for all the points in the set, where m is the smallest integer that ensures all items have the same code.
2) Next we compute $H^{m-1}_{E_8}(\mathbf{v})$ for all points and use sorting algorithm to group items with the same hash code together. Each hierarchy tree node stores the start and end position for one subgroup and its common E_8 code for the corresponding hierarchy level.

The process is repeated until $m = 0$ and the linear array contains the common original E_8 code for each bucket. Given a query, we traverse the hierarchy in a recursive manner by visiting the child node whose E_8 code is the same as the query at that level. The traversal stops until such a child node does not exist and all the buckets rooted from current node need to be searched for the multi-probe process.

If the dimension of the dataset is $M > 8$, we use the combination of $\lceil\frac{M}{8}\rceil$ E_8 lattices to represent the LSH table.

V. GPU-based Parallel Bi-level LSH

The parallelism of commodity GPUs can be used to accelerate k-nearest neighbor computation. Kato et al. [24] solve the problem on multiple GPUs using N-body algorithm and partial sorting. Brown et al. [25] use Kd-tree computation on GPUs to perform nearest-neighbor search. Pan et al. [26] use a GPU-based LSH algorithm to perform k-nearest neighbor search for motion planning. However, this approach is limited to the basic LSH algorithm, and short-list search computation is a bottleneck in the resulting computation. In this section, we briefly describe the parallel acceleration for the Bi-level LSH algorithm on current GPU architectures. For more details, please refer to [16].

A. LSH Hash Table on GPUs

As mentioned in Section IV-B1, in the second level of the bi-level algorithm, we need to construct LSH hash table for each leaf node cell calculated during the first level. The LSH code of one item and its leaf node index composes its Bi-level LSH code, i.e. $\tilde{H}(\mathbf{v}) = (\text{RP-tree}(\mathbf{v}), H(\mathbf{v}))$. We now further construct the hash table in a manner that maps well to current GPU architectures.

The GPU-based LSH table is implemented as a linear array along with an indexing table. The linear array contains the Bi-level LSH codes of all the items in the dataset, which have been sorted to collect items with same LSH codes together. All the data items with the same LSH code constitute a bucket which is described by the start and end positions of the bucket (i.e. the bucket interval) in the sorted linear array. As each bucket uniquely corresponds to one LSH code, we can use the terminology ‘bucket’ and ‘unique LSH code’ in the same sense. The indexing table corresponds to a cuckoo hash table with each key as one LSH code and the value associated with the key is the corresponding bucket interval for the LSH code in the linear array. In practice, the key, a dim-M LSH code, is compressed to a dim-1 key by using another hash function. The parallel LSH hash table is implemented as a cuckoo hashing table on GPUs [27]. However, we do not construct a hash table for each data group. Instead, we store all the Bi-level LSH codes in one hash table, because the group index (i.e. the output from RP-tree) can distinguish codes from different groups. The hierarchical LSH structure can also be constructed and searched efficiently using GPU-based Morton curves [16].

B. Short-list Search

Short-list search ranks the distances of neighborhood candidates to the query and chooses the k candidates that are closest to the query item. It is usually implemented by inserting the candidates sequentially into a max-heap with the maximum size k, which maintains the k best candidates up to this point. Short-list search is the main bottleneck in LSH algorithm and can take more than 95% of the overall running time.

We use a GPU-based algorithm to accelerate short-list searches for multiple queries. One naive way is to let each GPU thread handle the short-list search for a single query [26]. This is simple to implement but may not be efficient in practice. First, different queries may consider different number of candidates. Such imbalance of tasks will make some threads busy for heap operations, while some other threads are idle and the speed of the overall algorithm is limited by the slowest thread. Second, the max-heap computation is usually implemented on the slower global memory instead of the high-speed shared memory, because the size of max-heap is usually too large for shared memory. Therefore this method can not benefit from shared memory to accelerate data access. Finally, the heap operation (insert, heapify, etc) is related to tree traversal, and is performed in an independent manner for different queries. As a result, instruction divergence and non-coalesced memory accesses will occur among various threads and thereby reduce the overall performance on GPUs.

One possibility is to let each thread perform the heap-insert operation for one candidate for one of the queries. This strategy has been used before to handle multiple collision queries between 3D objects on GPUs [28]. However, in collision detection queries, each thread only traverses one binary tree, but does not update the tree so there are no conflicts among the queries that are performed in parallel. In our case, each thread may need to add one candidate to the max-heap and may remove another item. Different threads may operate on the same max-heap simultaneously. To avoid any conflicts between multiple threads, we need to use a heap representation that allows concurrent access of different threads. Most concurrent heap approaches are based on mutual exclusion, locking part of a heap when inserting or deleting the nodes so that other threads would not access the currently updated element. However, this blocking-based algorithm limits the potential performance to a certain degree, since it has several drawbacks such as deadlock and starvation. The lock-free approach [29] avoids blocking by using atomic synchronization primitives and guarantees that at least one active operation can be processed. However, lock-free methods can be inefficient on current GPUs architectures.

Our solution is shown in Figure 3, which uses work queues allocated on the global memory to accelerate short-list search. Given multiple queries and their candidate sets, we first compute the number of queries that can fit into the global memory, i.e. have enough memory for the candidate sets and the initial k-nearest neighbors. The initial k-nearest neighbors are empty or are the results from previous LSH tables. We copy the initial sets and the candidate sets to the work queue and perform *clustered-sort* on the distances between the points in the work queue and the query points. Here *clustered-sort* is a sorting algorithm which also maintains the relative orders between the clusters. Finally, we perform a compact operation to obtain updated k-nearest neighbor results. This process is repeated until all the candidates have been processed. In order to implement this approach on current GPUs, we also use many new GPU-based primitives [16] for efficient parallel computation.

Our work queue-based parallel method has a time complexity $T_P(n) = 40n/p$, where p is the number of GPU cores and n is the number of elements in the database [16]. It is *work efficient*, i.e. its complexity is bounded both above and below asymptotically by the complexity of the most efficient serial algorithm [16]. Moreover, when k increases, the complexity of work queue-based parallel method increases very slowly while the complexity of per-thread per short-list method increases linearly with k. In practice, the work queue-based method can be 2-5 times faster than the naive parallel short-list search [16].

VI. Implementation and Results

In this section, we compare the performance of our Bi-level LSH algorithm with prior LSH methods. All the experiments are performed on a PC with an Intel Core i7 3.2GHz CPU and NVIDIA GTX 480 GPU. The system has 2GB memory and 1GB video memory.

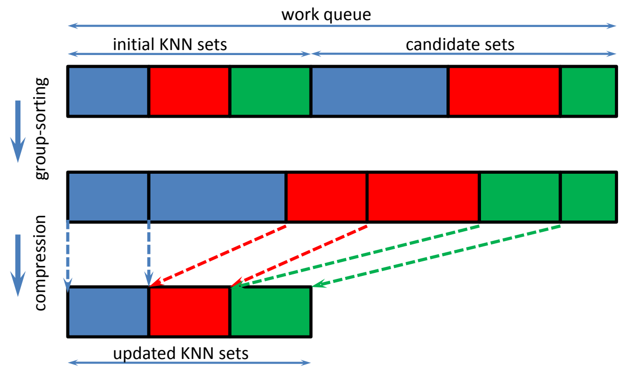


Fig. 3. Work queue based parallel short-list search: different colors correspond to different queries. *Clustered-sort* sorts the candidates for the same query together in an ascending order of distance. Next we collect the first k elements from the reordered candidates. These elements are of the smallest distance to the query and are used as the initial k-nearest neighbor results in the next iteration.

A. Datasets

We employ two datasets to evaluate our method. One benchmark is the LabelMe image dataset (http://labelme.csail.mit.edu), which includes nearly 200,000 images. Each image is represented as a GIST feature of dimension 512. The other benchmark is the Tiny Image database (http://horatio.cs.nyu.edu/mit/tiny/data/index.html), which has 80 million images. Each image is represented as a GIST feature of dimension 384. These datasets have been used to evaluate the performance of other LSH based algorithms [13],

B. Quality Comparison

1) Comparison Metrics: We compare the quality of our Bi-level LSH scheme with prior LSH methods in different ways. First, given the same selectivity (Equation (5)), we compare the recall ratios (Equation (3)) and error ratios (Equation (4)) of different methods. The selectivity measures the number of potential candidates for short-list search and is proportional to the algorithm's runtime cost. Recall and error ratios measure how close the approximate result is to the exact query result. Given a fixed runtime budget, these criteria are used to compare the quality of two k-nearest neighbor algorithms. Second, we compare the standard deviations due to randomly selected projections between prior LSH approaches and our bi-level scheme. The LSH-based methods are randomized algorithms whose performance may change when the projection vectors are chosen randomly. In our case, we tend to ensure that the algorithm gives high quality results with low runtime cost even in the worst case, and not just the average case over different random projections. Finally, we compare the standard deviations of different methods with our approach over different queries. Our goal is to ensure that the algorithm computes k-nearest neighbor results with similar quality and runtime cost for all different queries.

2) Metric Evaluation: In terms of the comparison, we use 100,000 items in the dataset to calculate the LSH hash table and then use another 100,000 items in the same dataset as the queries to perform k-nearest neighbor search using different LSH algorithms. For one specific LSH algorithm, we use three different L values (10, 20, 30) and for each L, we increase the bucket size W gradually which will result in selectivities in an ascending order. We keep all other parameters fixed ($M = 8$, $k = 500$). For each W, we execute the 100,000 k-nearest neighbor queries 10 times with different random projections. For each execution of the LSH algorithm, we compute its recall ratio (ρ), error ratio (κ) and selectivity (τ) as the measurements for quality and runtime cost, which change with the bucket size W. According to our previous analysis in Section IV-A3, the three measurements are random variables and we have $\rho = \rho(W, r_1, r_2)$, $\kappa = \kappa(W, r_1, r_2)$ and $\tau = \tau(W, r_1, r_2)$, where r_1 and r_2 are two random variables representing various projections and queries, respectively.

In order to compare the quality of different methods, given the same selectivity, we need to compute two curves: $\rho(W) = f(\tau(W))$, which describes the relationship between selectivity and recall ratio and $\kappa(W) = g(\tau(W))$, which describes the relationship between selectivity and error ratio. We estimate the two curves based on expected measurements over all LSH executions with the same W but different projection directions and queries

$$\mathbb{E}_{r_1,r_2}(\rho(W)) = \tilde{f}(\mathbb{E}_{r_1,r_2}(\tau(W))), \tag{11}$$

and

$$\mathbb{E}_{r_1,r_2}(\kappa(W)) = \tilde{g}(\mathbb{E}_{r_1,r_2}(\tau(W))), \tag{12}$$

where $\tilde{f}$ and $\tilde{g}$ are conservative estimates of quality metric because

$$\tilde{f}(\mathbb{E}_{r_1,r_2}(\tau)) = \mathbb{E}_{r_1,r_2}(\rho) = \mathbb{E}_{r_1,r_2}(f(\tau)) \leq f(\mathbb{E}_{r_1,r_2}(\tau))$$

and therefore $\tilde{f} \leq f$. Here $\leq$ relation is used because f is usually a concave function. Similarily, we have $\tilde{g} \leq g$. Moreover, we use standard deviations $\mathrm{Std}_{r_1}(\mathbb{E}_{r_2}(\tau))$, $\mathrm{Std}_{r_1}(\mathbb{E}_{r_2}(\rho))$, $\mathrm{Std}_{r_1}(\mathbb{E}_{r_2}(\kappa))$ to measure the variation of selectivity, recall ratio and error ratio caused by random projections. Moreover, we use the standard deviations $\mathrm{Std}_{r_2}(\mathbb{E}_{r_1}(\tau))$, $\mathrm{Std}_{r_2}(\mathbb{E}_{r_1}(\rho))$, $\mathrm{Std}_{r_2}(\mathbb{E}_{r_1}(\kappa))$ to measure the variation of selectivity, recall ratio and error ratio due to different queries. In practice, we use arithmetic mean of the measurements to estimate these terms.

3) Performance Analysis: We perform slightly different performance analysis as compared to prior approaches [8] and [10]. First, we take into account the variances due to randomness of LSH scheme as one of our main goals is to reduce these variances. Secondly, we search a neighborhood with larger size (k = 500) than k = 20 or 50 used in previous works [8], [10], which makes the LSH framework more difficult to obtain high recall ratio. We find that our Bi-level LSH algorithm gives similar performance improvement when k = 500 and k = 20, 50. Due to space limitations, we only show the performance results for k = 500 in Figure 5–Figure 12. Finally, we use a large query set $(100,000)$ instead of the several hundred queries used in previous methods, which results in a better estimate of the quality and runtime cost of LSH algorithms. The large neighborhood size, query number and the repetition with different random projections require more computations in our experiments, as compared to the others reported in prior work.

4) Quality Comparisons: The LSH algorithms that we compared include standard LSH and its two variants based on multiprobe or hierarchy technique (i.e. multiprobed standard LSH and hierarchical standard LSH), Bi-level LSH and its two variants using multiprobe or hierarchy technique (i.e. multiprobed Bi-level LSH and hierarchical Bi-level LSH). For each LSH algorithm, we further test its performance by using $\mathbb{Z}^M$ and E_8 lattice, respectively.

a) Standard LSH vs. Bi-level LSH: First, we compare the standard LSH algorithm with our Bi-level LSH algorithm while using $\mathbb{Z}^M$ lattice as the space quantizer, as shown in Figure 5. We notice the following facts from the results: 1) Given the same selectivity, our bi-level scheme can usually provide higher recall ratio and error ratio than standard LSH algorithm. The reason is that the RP-tree partition in the first level clusters similar features together, which are more likely to be close to each other in the feature space. This partition provides better locality coding than standard LSH hash function. However, we also observe that when the selectivity is large (0.6 in Figure 5(b) and 0.5 in Figure 5(c)), the limit recall ratio of Bi-level LSH can be smaller than that of standard LSH. The reason is that given the same bucket size W, Bi-level LSH always has more buckets than standard LSH due to the first level partition in bi-level scheme. However, in practice we are only interested in selectivity less than 0.4, otherwise the efficiency benefit of LSH over brute-force method is quite limited. We also observe that Bi-level LSH always provides results that are better than standard LSH. 2) We observe that when the selectivity increases, the error ratio increases much faster than the recall ratio. This is due to the concentration effect mentioned earlier. 3) Our Bi-level LSH has smaller standard deviation due to random projections than standard LSH (i.e. smaller ellipse in the figures), for both quality and selectivity. 4) When selectivity is small (i.e. small W), the standard deviation of quality is large while the standard deviation of selectivity is small. When selectivity is large (i.e. large W), the standard deviation of selectivity is large while the standard deviation of quality is small. The reason is that for small W, different projections will produce results with different quality, while for large W, the quality (i.e. recall and error ratio) of the k-nearest neighbor results will converge to one for most projections. 5) When L increases, the standard deviations of both Bi-level LSH and standard LSH decrease. The reason is that the probability that two points $\mathbf{u}$, $\mathbf{v}$ are projected into the same bucket by one of the L random projections is

$$1-\prod_{i=1}^{L}(1-\mathbb{P}[h_i(\mathbf{u}) = h_i(\mathbf{v})]) = 1-\prod_{i=1}^{L}\frac{\min(|\mathbf{a}_i \cdot \mathbf{u} - \mathbf{a}_i \cdot \mathbf{v}|, W)}{W}.$$

Therefore, the final probability is related to the geometric average of the probabilities when using different projections and will converge to one quickly. However, we can see that the standard deviations of Bi-level LSH when using $L = 10$ are almost as small as the standard deviations of standard LSH when using $L = 20$ or 30, which means that we save $2/3$ runtime cost and memory for LSH hash tables when using our Bi-level LSH to obtain similar quality with standard LSH.

Figure 6 shows the comparison between the standard LSH algorithm and our Bi-level LSH algorithm while using E_8 lattice as the space quantizer. We can see that the results are similar to the case when the algorithm uses $\mathbb{Z}^M$ lattice, though E_8 lattice offers better performance at times. Bi-level LSH also outperforms standard LSH in this case.

b) Multiprobed LSH vs. Multiprobed Bi-level LSH: Figure 7 and Figure 8 show the comparison between Bi-level LSH and standard LSH when both methods are enhanced with the multiprobe technique [8], using $\mathbb{Z}^M$ and E_8 lattice, respectively. We use 240 probes for each methods. For $\mathbb{Z}^M$ lattice, we use the heap-based method in [8] to compute the optimal search order for each query. For E_8 lattice, we simply use the 240 neighbors of the lattice node that one query belongs to. As in previous cases, we observe that the Bi-level LSH results in better quality as compared to standard LSH. Figure 11 and Figure 12 compare the selectivity-recall curve among different methods when using $\mathbb{Z}^M$ lattice or E_8 lattice, respectively. We can see that for $\mathbb{Z}^M$ lattice, the use of multiprobe results in improved quality, as compared not using multiprobe. On the other hand, using multiprobe with the E_8 lattice slightly reduces the quality. The reason is that multiprobe is mainly used to obtain better quality with fewer LSH hash tables (i.e. L) by probing nearby buckets besides the bucket that the query lies in. However, the probability that these additional buckets contain the actual k-nearest neighbors is usually smaller than that of the bucket containing the query item. As a result, multiprobe scheme requires higher selectivity in order to obtain small improvements in quality parameters. E_8 lattice has higher density than $\mathbb{Z}^M$ lattice and therefore, additional E_8 buckets that need to be probed will contain more potential elements for short-list search but only a few them would actually correspond to the k-nearest neighbors. As a result, multiprobe technique results in larger performance degradation with E_8 lattice as compared to $\mathbb{Z}^M$ lattice. However, we observe that the multiprobe technique can reduce the variance caused by random projections. The reason is that multiprobe technique is equivalent to using larger L, which reduces the deviation caused by random projections.

c) Hierarchical LSH vs. Hierarchical Bi-level LSH: Figure 9 and Figure 10 show the comparison between Bi-level LSH and standard LSH when both methods are enhanced with the hierarchical structure introduced in Section IV-B, using $\mathbb{Z}^M$ and E_8 lattice, respectively. Given a set of queries, we first compute the potential elements for short-list search using Bi-level LSH or standard LSH. Next, we compute the median of the short-list sizes of all queries and use the result as the threshold about whether a bucket contains enough k-nearest neighbor candidates. For those queries with short-list size smaller than the median threshold, we search the LSH table hierarchy to find suitable bucket whose size is larger than the threshold. The hierarchical strategy is used to reduce the deviation caused by different queries. And Figure 9 and Figure 10 demonstrate that hierarchical strategy can reduce the deviations that are introduced by random projections. Moreover, as shown in Figure 11 and Figure 12, the hierarchical strategy does not result in quality degradation as multiprobe techniques. In all these cases Bi-level LSH results in improved performance.

d) Comparision among All Methods: We compare the selectivity-recall ratio among all the methods in Figure 11 and Figure 12. For $\mathbb{Z}^M$ lattice, multiprobed Bi-level LSH provides the best recall ratio. Moreover, Bi-level LSH, hierarchical Bi-level LSH and multiprobed standard LSH result in similar recall ratios. In contrast, standard LSH and hierarchical standard LSH have the lowest value of recall ratios. For E_8 lattice, Bi-level LSH, hierarchical Bi-level LSH and multiprobed Bi-level LSH provide the highest recall ratios. Standard LSH and hierarchical standard LSH have similar quality. The multiprobed standard LSH has the worst quality. We also compare the deviation caused by different queries in Figure 11 and Figure 12. We can see that the hierarchical Bi-level LSH provides results with the smallest deviation among all the methods. Hierarchical standard LSH also provides results with smaller deviation than standard LSH and multiprobed standard LSH.

C. Performance Variation due to Parameters

We now evaluate the performance of Bi-level LSH as a function of different parameters. In all comparisons, we fix $L = 20$ and change the other parameters, including number of partitions in the first level, LSH hash function dimension M and whether to use K-means or RP-tree for the first level partition.

Figure 13(a) shows the results when Bi-level LSH uses different number of partitions (1, 8, 16, 32, 64) in the first level. We can observe that when the number of sub-groups increases, the quality increases given the same value of selectivity. However, this increase slows down after 32 partitions.

Figure 13(b) compares Bi-level LSH with standard LSH with for different values of M. Bi-level LSH uses the hash code $\tilde{H}(\mathbf{v}) = \big(\text{RP-tree}(\mathbf{v}), H(\mathbf{v})\big)$. The first level adds some additional code before the standard LSH code. This comparison shows that the improvement of Bi-level LSH is due to using better code but not longer code. As shown in the result, Bi-level LSH provides better quality than standard LSH using different M.

Figure 13(c) compares the Bi-level LSH when using K-means and RP-tree in the first level. We can see that the quality and deviation when using RP-tree is better than those when using K-means.

D. GPU Acceleration

Here we compare the efficiency of three approaches. The first is our naive GPU implementation, which uses parallel hash table based on cuckoo hashing to accelerate hash table

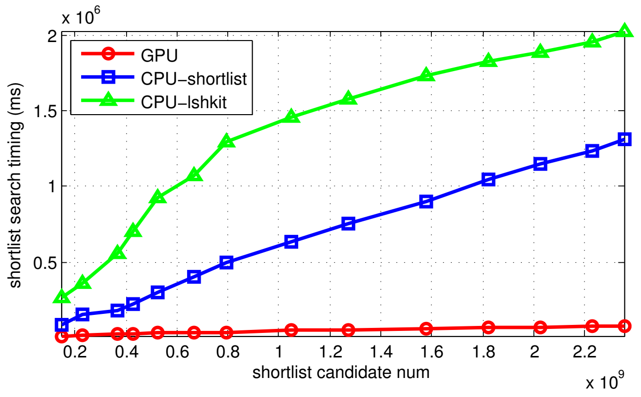


Fig. 4. Performance comparison on short-list search: training set size 100,000, testing set size 100,000, set $K = 500$, $L = 10$, $M = 8$, change W to generate different number of short-list candidates. We compare different methods: pure CPU (CPU-lshkit), GPU hash table and CPU short-list search (CPU-shortlist) and pure GPU (GPU).

access and uses parallel heap-sorting for short-list search. The second approach replaces the parallel short-list search by serial short-list search on GPU but still uses parallel hash table. The final method is based on LSHKIT (http://lshkit.sourceforge.net), where hash table and short-list search are implemented on a single core CPU. In our experiments, we change the bucket size (W) to generate different number of short-list candidates. For different numbers of short-list candidates, the timing result for the three approaches is shown in Figure 4. The second method is about 2x faster than the third one, where the acceleration is obtained by GPU parallel hash table. The first method is about 15-20x faster than the second one, where the main acceleration is obtained based on GPU short-list search. Overall, our GPU-based per-thread per-query Bi-Level LSH algorithm can provide 40x acceleration over the CPU implementation. Another 2-5x acceleration can be obtained by replacing the parallel heap-sorting by the work-queue based method. Please refer to [16] for more details on GPU performance.

VII. Conclusion and Future Work

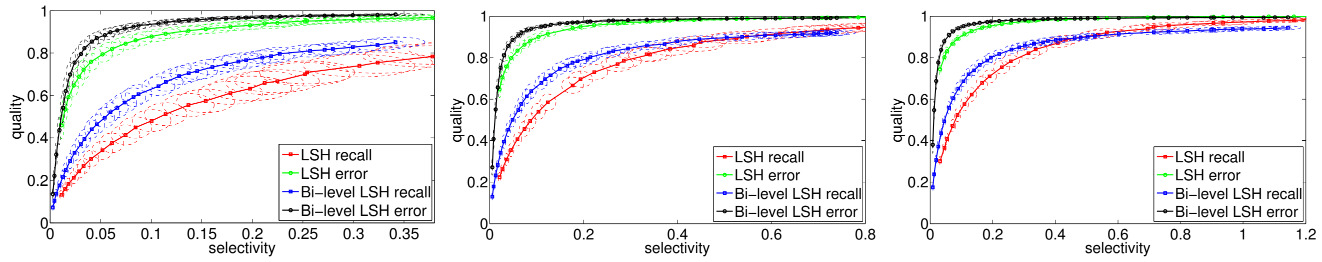
We have presented a new Bi-level LSH based algorithm for efficient k-nearest neighbor search. We use RP-tree to reduce the variation caused by randomness in LSH framework, to make hashing adaptive to datasets and improve the compactness of LSH coding. We also construct a hierarchical LSH table to make the algorithm adapt to different queries. The hierarchy is also enhanced by E_8 lattice to handle high dimensional datasets. In practice, our algorithm can compute k-nearest neighbor results with improved quality as compared to prior LSH methods, when given the same selecitivity budget and can generate results with less variance in quality and runtime cost.

There are many avenues for future work. We hope to test our algorithm on more real-world datasets, including images, textures, videos, etc. We also need to design efficient out-of-core algorithms to handle very large datasets (e.g. $>$ 100GB).

Acknowledgement: This research is supported in part by ARO Contract W911NF-10-1-0506, NSF awards 0917040, 0904990 and 1000579.

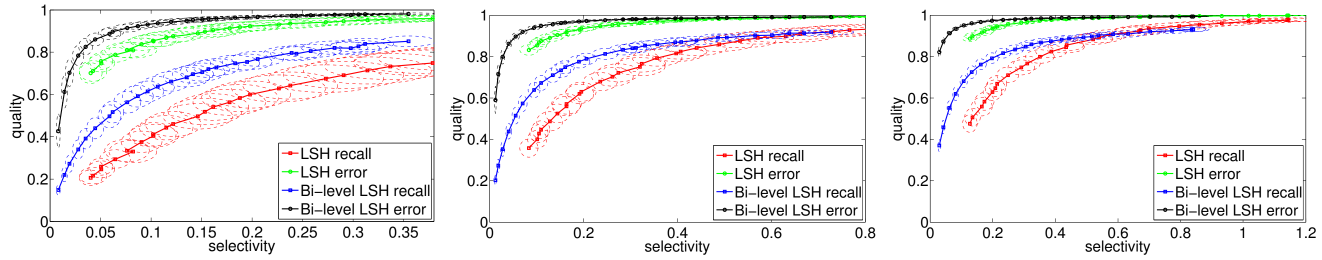
References

[1] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh, "A road network embedding technique for k-nearest neighbor search in moving object databases," in *SIGSPATIAL GIS*, 2002, pp. 94–100.
[2] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *International Conference on Machine Learning*, 2006, pp. 97–104.
[3] N. Katayama and S. Satoh, "The SR-tree: an index structure for high-dimensional nearest neighbor queries," in *International Conference on Management of Data*, 1997, pp. 369–380.
[4] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *International Conference on Very Large Data Bases*, 1998, pp. 194–205.
[5] J. M. Kleinberg, "Two algorithms for nearest-neighbor search in high dimensions," in *Symposium on Theory of Computing*, 1997, pp. 599–608.
[6] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Symposium on Computational Geometry*, 2004, pp. 253–262.
[7] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *International Conference on Very Large Data Bases*, 1999, pp. 518–529.
[8] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: efficient indexing for high-dimensional similarity search," in *International Conference on Very Large Data Bases*, 2007, pp. 950–961.
[9] M. Bawa, T. Condie, and P. Ganesan, "LSH forest: self-tuning indexes for similarity search," in *International Conference on World Wide Web*, 2005, pp. 651–660.
[10] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li, "Modeling lsh for performance tuning," in *Conference on Information and Knowledge Management*, 2008, pp. 669–678.
[11] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Symposium on Foundations of Computer Science*, 2006, pp. 459–468.
[12] H. Jégou, L. Amsaleg, C. Schmid, and P. Gros, "Query adaptative locality sensitive hashing," in *International Conference on Acoustics, Speech and Signal Processing*, 2008, pp. 825–828.
[13] M. Raginsky and S. Lazebnik, "Locality-sensitive binary codes from shift-invariant kernels," in *Advances in Neural Information Processing Systems*, 2009.
[14] J. He, W. Liu, and S.-F. Chang, "Scalable similarity search with optimized kernel hashing," in *International Conference on Knowledge Discovery and Data Mining*, 2010, pp. 1129–1138.
[15] A. Dhesi and P. Kar, "Random projection trees revisted," in *Advances in Neural Information Processing Systems*, 2010.
[16] J. Pan and D. Manocha, "Fast GPU-based locality sensitive hashing for k-nearest neighbor computation," in *International Conference on Advances in Geographic Information Systems*, 2011.
[17] D. E. Knuth, *The art of computer programming, volume 3: sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1998.
[18] A. Joly and O. Buisson, "A posteriori multi-probe locality sensitive hashing," in *International Conference on Multimedia*, 2008, pp. 209–218.
[19] Y. Freund, S. Dasgupta, M. Kabra, and N. Verma, "Learning the structure of manifolds using random projections," in *Advances in Neural Information Processing Systems*, 2007.
[20] S. Dasgupta and Y. Freund, "Random projection trees and low dimensional manifolds," in *Symposium on Theory of Computing*, 2008, pp. 537–546.
[21] D. Yan, L. Huang, and M. I. Jordan, "Fast approximate spectral clustering," in *International Conference on Knowledge Discovery and Data Mining*, 2009, pp. 907–916.
[22] O. Egecioglu and B. Kalantari, "Approximating the diameter of a set of points in the Euclidean space," *Information Processing Letters*, vol. 32, pp. 205–211, 1989.
[23] S. Liao, M. A. Lopez, and S. T. Leutenegger, "High dimensional similarity search with space filling curves," in *International Conference on Data Engineering*, 2001, pp. 615–622.



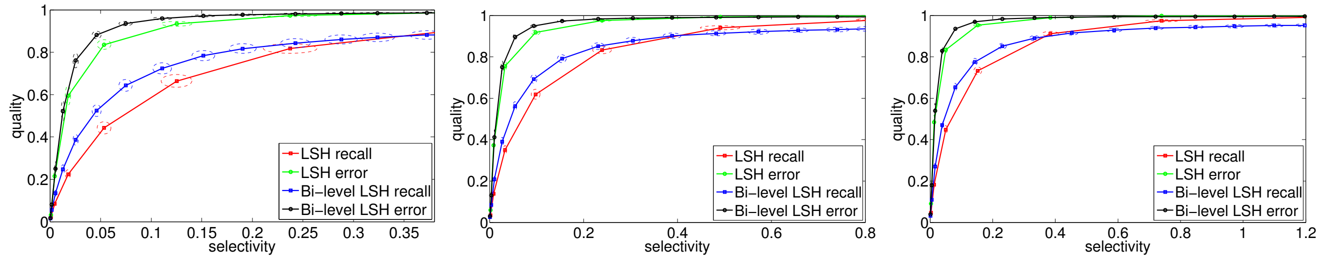
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Fig. 5. Quality comparison between our Bi-level LSH and standard LSH when using $\mathbb{Z}^M$ lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



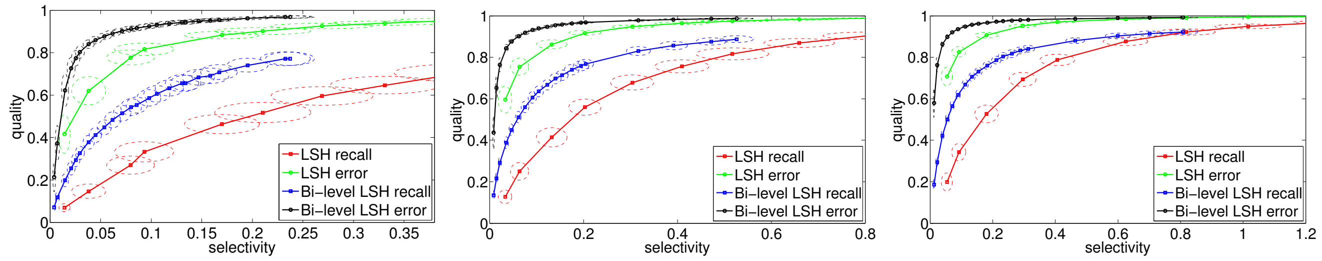
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Fig. 6. Quality comparison between our Bi-level LSH and standard LSH when using E_8 lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



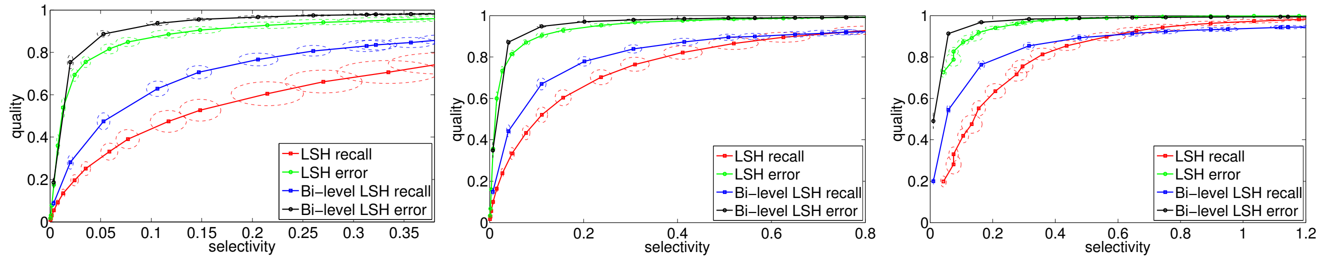
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Fig. 7. Quality comparison between our multiprobed Bi-level LSH and standard multiprobed LSH when using $\mathbb{Z}^M$ lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



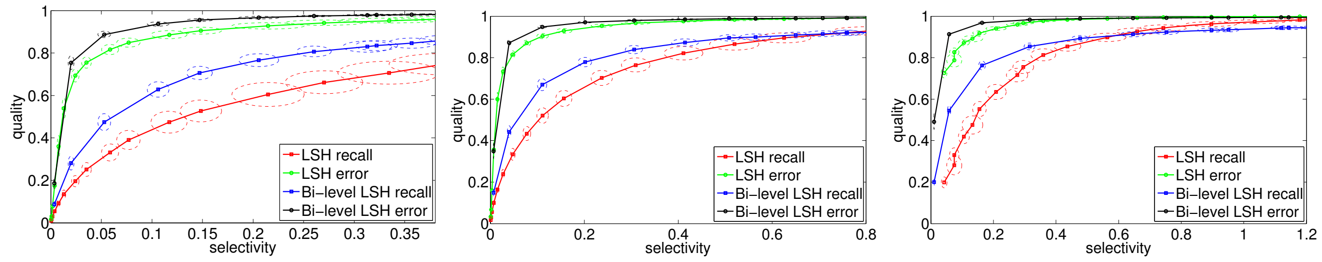
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Fig. 8. Quality comparison between our multiprobed Bi-level LSH and standard multiprobed LSH when using E_8 lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



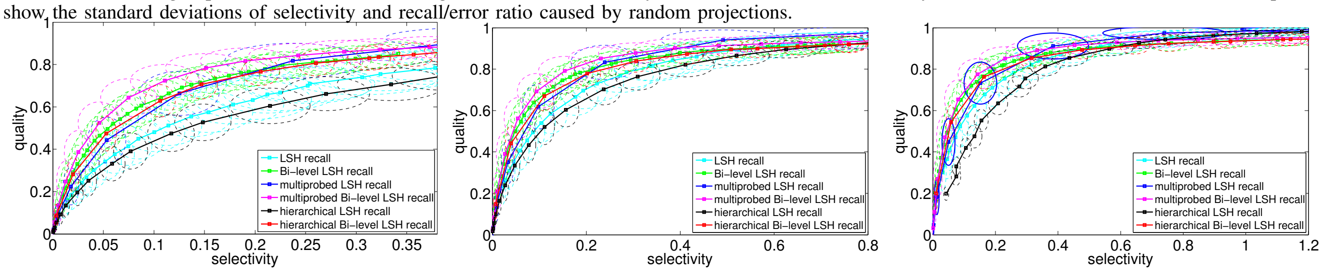
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Fig. 9. Quality comparison between our hierarchical Bi-level LSH and standard hierarchical LSH when using $\mathbb{Z}^M$ lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



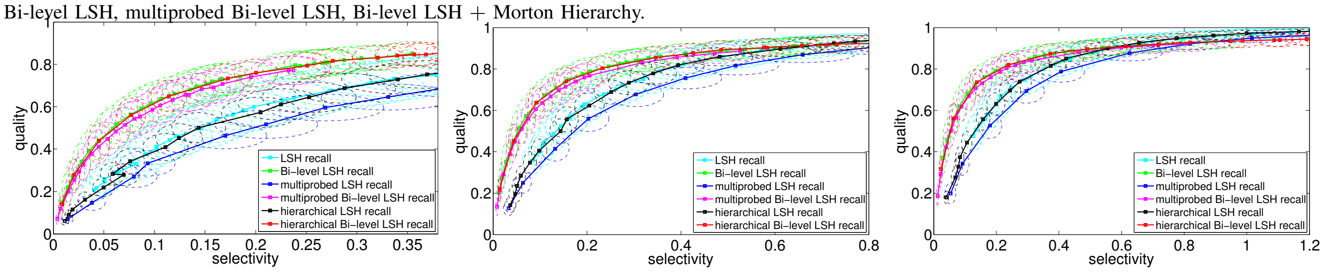
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Fig. 10. Quality comparison between our hierarchical Bi-level LSH and standard hierarchical LSH when using E_8 lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. Each figure shows the selectivity–recall curves and the selectivity–error curves for both methods. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



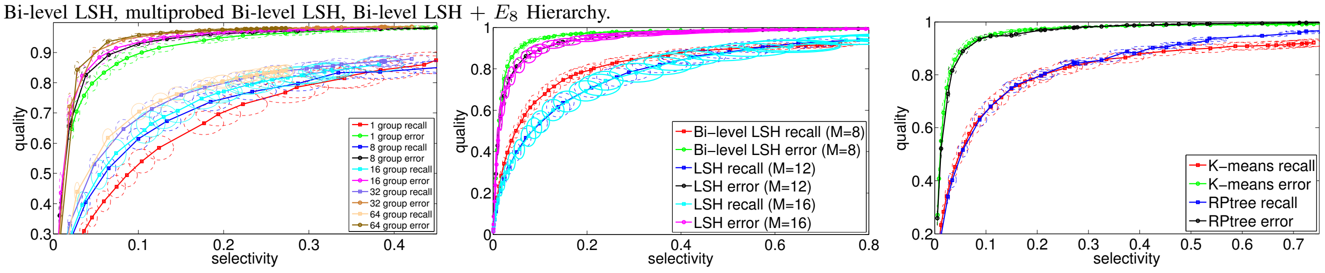
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Fig. 11. Variance caused by queries when using $\mathbb{Z}^M$ lattice. We compare six methods: standard LSH, multiprobed LSH, standard LSH + Morton hierarchy, Bi-level LSH, multiprobed Bi-level LSH, Bi-level LSH + Morton Hierarchy.



(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Fig. 12. Variance caused by queries when using E_8 lattice. We compare six methods: standard LSH, multiprobed LSH, standard LSH + E_8 hierarchy, Bi-level LSH, multiprobed Bi-level LSH, Bi-level LSH + E_8 Hierarchy.



(a) (b) (c)

Fig. 13. all $L = 20$ (a) different group number (b) different M (c) RP-tree and K-means

[24] K. Kato and T. Hosino, “Multi-GPU algorithm for k-nearest neighbor problem,” *Concurrency and Computation: Practice and Experience*, vol. 23, 2011.

[25] S. Brown and J. Snoeyink, “GPU nearest neighbors using a minimal kd-tree,” in *Workshop on Massive Data Algorithmics*, 2010.

[26] J. Pan, C. Lauterbach, and D. Manocha, “Efficient nearest-neighbor computation for GPU-based motion planning,” in *International Conference on Intelligent Robots and Systems*, 2010, pp. 2243–2248.

[27] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, “Real-time parallel hashing on the GPU,” *ACM Transations on Graphics*, vol. 28, pp. 154:1–9, 2009.

[28] J. Pan and D. Manocha, “Gpu-based parallel collision detection for fast motion planning,” *International Journal of Robotics Research*, to appear.

[29] H. Sundell and P. Tsigas, “Fast and lock-free concurrent priority queues for multi-thread systems,” *Journal of Parallel and Distributed Computing*, vol. 65, pp. 609–627, 2005.