Fast Approximate k-Nearest Neighbors in High Dimensions

Undergraduate Honors Thesis

Prad Nelluru
Department of Computer Science
The University of Texas at Austin

May 8, 2013
ABSTRACT

k-nearest neighbors search is widely-used in many applications, including computer vision, information retrieval, and machine learning. k-means locality sensitive hashing is a data-adaptive method for solving this problem approximately. Previous studies of the algorithm focus only on its accuracy with respect to the number of centroids in the k-means phase. In this work, we show the impact of all of the parameters on both accuracy and speed. The parameters are evaluated on a real-world setup, namely search on 128-dimensional SIFT image descriptors. We reveal that surprisingly little work is required in the k-means phase to build indexes that enable fast and accurate searching. Further, we show that accuracy can be increased dramatically by using both multiple indexes and probes. We also present a parallel implementation of k-means locality sensitive hashing that is 20-25x faster than our baseline version for a desired level of accuracy. These results show that k-means locality-sensitive hashing can be fast and accurate on large, high-dimensional datasets.
To the dance between theory and practice -
ACKNOWLEDGEMENTS

Foremost, I’d like to express my sincere gratitude to my advisors within the Intelligent Systems Group - Rupesh Nasre, and Prof. Kesha Pingali. Rupesh’s encouragement and mentorship have been invaluable in bringing my work to fruition. I’m thankful to Prof. Pingali for kindling my interest in a variety of computer science topics, as well as offering me a chance to conduct research in the ISS group.

I’d also like to thank my second reader, Prof. Calvin Lin, for introducing me to the joys of research in computer science. His classes have invariably convinced me that I, too, could help uncover the beauty in the field.

And, much thanks to Prof. Bill Press for being my third reader. Besides my advisors, I’d like to thank all that have listened to my ideas and helped me solve the challenges I’ve encountered. In particular, all of the members of the ISS group have continually surprised me by their willingness to listen to my ideas and help me solve the problems I’ve encountered.

And finally, I’d like to thank my family for their unwavering support of all my endeavors.
INTRODUCTION

K-nearest neighbors search (KNN) is a well-studied problem in computational geometry. Simply put, the problem is to index a database of vectors such that given a query point, retrieving its $k$ nearest neighbors is fast. KNN is useful as a general-purpose means of comparing data and is used in a variety of fields. For example, image similarity search can be formulated as a nearest neighbors search over image descriptor vectors. As these applications often deal with large, high-dimensional databases, a robust KNN method is needed.

It is well-known that exact KNN is inherently expensive due to the curse of dimensionality \[4\]. So, approximate k-nearest neighbors search (AKNN) is used to dramatically improve search speed, and is often necessary to search the database in reasonable time.

The most popular class of AKNN methods is known as locality-sensitive hashing \[11\]. The key idea is to hash points to buckets such that close points are likely to be in the same bucket. Query points are hashed using the same hash function and the resulting bucket is exhaustively searched to find the $k$ nearest neighbors. Many hash functions have been introduced to perform this mapping, but none are data-adaptive \[12\] \[1\].

K-means locality sensitive hashing (KLSH) is a recent variant that constructs hash functions using the k-means algorithm \[16\]. KLSH uses the k-means algorithm to partition the database into a set of Voronoi cells. The KLSH hash function is simply a mapping between a cell and it’s corresponding bucket in the index. KLSH has been shown to perform better than previous LSH methods because it adapts to the dataset at hand \[16\]. However, using the k-means algorithm to preprocess the database is seen as expensive.

KLSH is a useful algorithm: it’s simple to implement and produces accurate results on real datasets. So, it’s important to learn how to best use this algorithm in practice. To the best of our knowledge, the algorithm’s performance has only been evaluated with respect to one parameter - the number of centroids to be found in the k-means phase.

In this work, we show the effects of all of the parameters of the KLSH algorithm on its performance over a real dataset. We find that the amount of work that needs to be done in the k-means step - the sample size and number of iterations - is a surprisingly small, making
KLSH practical for much larger datasets than previously thought. We also show that using just a few indexes and probes can dramatically improve accuracy. Further, we reveal that both accuracy and speed aren’t greatly affected as the number of nearest neighbors that need to be retrieved increases.

We also detail how we implemented a parallel version of KLSH that achieves 20-25x speedup over our baseline version. In total, this work shows that a combination of good parameters and a fast implementation enables the use of KLSH to solve AKNN for large, high-dimensional datasets.

We start in chapter 2 by placing KLSH in context with other AKNN methods and explaining how both KLSH and the k-means algorithm work. In chapter 3, we detail our implementation of KLSH, discuss the experimental setup, show the effects of various parameters on both the speed and accuracy of KLSH, and show how to improve them. We conclude by discussing future directions for studying KLSH.
BACKGROUND

It has been claimed that k-nearest neighbors search “boasts more applications than virtually any other geometric problem.” The problem is important in machine learning, information retrieval, computer vision, data compression, and more. The wide applicability of nearest neighbors search can be attributed to its usefulness as a general-purpose means of indexing and comparing data. As long as objects can be represented as vectors in a metric space, their similarity can be compared based on the underlying distance metric. A few contexts follow:

1. Statistical classification. Query points can be classified by looking at the classifications of the nearest neighbors in the database.

2. Image similarity search. A database of images can be run through a descriptor algorithm to produce a database of fixed-dimensional points. Given a new image, similar images in the database can be found by running the new image through the same descriptor and doing a nearest neighbors search in the database of points.

3. Document retrieval. Documents can be vectorized with a bag-of-words model. Similar documents can be retrieved by finding the nearest neighbors of a query document.

The distance function (and therefore the definition of similarity) used can vary, but we will restrict our discussion to the Euclidean distance metric, which is popular in practice.

A brute force method of solving exact KNN is to compute the distance between a query point and each point in the database, keeping track of the closest $k$ points. With a time complexity of $O(n)$ per query, this linear scan method is impractical for large datasets and high dimensionalities.

The exact KNN can be solved efficiently when the intrinsic dimensionality of the data is low ($d < 10$) using space partitioning trees such as K-D trees. K-D trees partition the database into axis-aligned $d$-dimensional cells in a data-dependent way. Nearest neighbors can be found by traversing the tree to find the cell a query point falls into, and then doing a short-list search among the points in that cell. If the tree is well-constructed, queries take $O(\log(n))$ time.
It is important to consider KNN over high-dimensional data since many real-world applications deal with data with hundreds or thousands of dimensions. For example, a popular image descriptor is the scale-invariant feature transform (SIFT) algorithm, which produces vectors of 128 dimensions under standard parameters. The 28 pixel x 28 pixel handwritten digit images in the widely-used MNIST dataset can be thought of as 28x28 = 784-dimensional vectors.

However, it is well-known that exact KNN is computationally expensive in high dimensions because of the curse of dimensionality. The hypervolume of the space that needs to be searched grows exponentially with the number of dimensions \(d\). Therefore, the space and time requirements for solving exact KNN grow exponentially as a function of \(d\). For K-D trees, this limitation translates into having to look through nearly all the nodes in the database.

### 2.1 Locality-Sensitive Hashing

Since the exact version of the KNN problem is intractable for many real datasets, much effort has been spent in devising approximate algorithms. The approximate k-nearest neighbors problem (AKNN) is important because:

1. Approximate results are permissible for many real-world applications.
2. Some problems couldn’t be solved at all without approximation.
3. AKNN algorithms usually allow the user to adjust the amount of computation required to adjust the quality of the results.

One of the most popular class of methods for AKNN is locality-sensitive hashing (LSH). LSH is a way to reduce the dimensionality of the data such that points that are close in high dimensions are close in low dimensions. Once in low dimensions, data can be discretized and placed into buckets. Queries go through the same process and are compared with points in the relevant buckets. The effectiveness of this method hinges on being able to do a fast dimensionality reduction and being able to quickly find which bucket a given low dimensional point falls into.

For a concrete example of LSH, let’s consider one of first LSH algorithms, introduced by Datar et al. In this method, which we will call classical LSH, high-dimensional points are randomly projected onto the real number line. Then, those points are placed into buckets corresponding to equi-width line segments on that line. A locality-sensitive hash function that maps high dimensional points to buckets is given by
\[ h(v) = \left\lfloor \frac{v \cdot r + b}{w} \right\rfloor \]

where \( v \) is the given high dimensional point, \( r \) is a random Gaussian vector with the same dimensions, \( b \) is a uniformly-generated random bias, and \( w \) is the width of the bucket on the real number line. For example, if \( v \cdot r = 13.02 \), the bias is \( 1.5 \), and the width of each line segment is \( 3 \), \( v \) would fall into the bucket corresponding to \( h(v) = 4 \).

We can see intuitively that this projection is locality-sensitive. Close points in high dimensions have a high chance of mapping to the same buckets; i.e. the difference between their dot-products with \( r \) will be small. Similarly, far points in high dimensions will likely map to different buckets. Note that the chance of similar points colliding into the same buckets is sensitive to the values of \( b \) and \( w \).

To further magnify the chance of close points landing in the same bucket and far points landing in different buckets, we can use \( t \) different hash functions. These hash functions can be readily constructed by picking different values of \( r \) and \( b \). The results of running \( v \) through \( t \) hash functions is a \( t \)-tuple of integer values. Since this \( t \)-dimensional space is sparse, we can then use conventional hashing functions to find map \( t \)-tuples to buckets in a hash table. In practice, we can use \( L \) different such projections of \( t \) hash functions to increase the chance of finding nearest neighbors in the buckets that are searched.

To summarize, LSH works by:

1. Reducing dimensionality such that close points in high dimensions are close in low dimensions. (i.e. random projection onto the real line.)

2. Mapping regions in low dimensional space to buckets, where database points are stored. (i.e. hashing \( t \) projections onto one bucket.)

Notice that projecting onto the real number line corresponds to projecting onto one dimension. Intuitively, one can see that there’s information loss in this transformation. (It’s easy for high dimensional points to fall into the same buckets.) So, one might ask if there are ways to project onto higher dimensional spaces to reduce this information loss. Indeed, later work has resulted in LSH via projections onto 8 dimensions [12] and 24 dimensions [1].

The LSH methods mentioned so far are all based on structured quantizers, methods that partition the space into a regular cells of equal size. Classical LSH partitions the real number line into equi-width line segments. An equi-width line segment can be defined as the region surrounding a particular integer point. So, all the equi-width line segments on the line can be described by the set of integer points that induce those line segments, a 1-d lattice. (More formally, a lattice
is a discrete subgroup of $\mathbb{R}^n$.) We can see that a simple 1-d lattice can be constructed that induces a maximal "cover" of the space (i.e. the line segments induced by the lattice covers the entire real number line.)

Similarly, maximal lattice coverings have been found for dimensions 1, 4, 8, and 24 \cite{14}. (These maximally covering lattices induce $d$-dimensional hyperspheres. A line segment is a 1-dimensional hypersphere.) It has been found that structured LSH methods based on higher dimensional lattices like the E8 lattice ($d = 8$) and the Leech Lattice ($d = 24$) perform better than the classical LSH approach \cite{16}. (We do not know of a maximally covering lattice for any $d > 24$.) Again, the intuitive reason for this behavior is because less information is being lost in the dimensionality reduction.

Structured LSH approaches have yielded theoretically near-optimal hashing algorithms for AKNN \cite{1}. However, it has been found that these approaches don’t perform well in practice on real datasets \cite{16}. Since structured quantizers partition data into regular regions, some regions can be very populated while others lie empty. (Real data are not uniformly-distributed, the ideal case for structured quantizers.) This load imbalance between cells translates to wildly varying numbers of points in the corresponding buckets in the index structure. Load imbalance is problematic because some queries will have to search through buckets with too many points (increasing query time unnecessarily) and others will search through buckets with too few points (reducing accuracy.) To alleviate these problems, an unstructured quantizer, a method that partitions the dataset in a data-dependent manner, is desired.

## 2.2 K-Means Locality Sensitive Hashing

### k-means locality sensitive hashing (KLSH)

(KLSH) is a data-adaptive LSH algorithm for approximate k-nearest neighbors search. The key idea of KLSH is to partition the database of points into Voronoi cells using the k-means clustering algorithm. It has been shown to work well in practice compared to previous LSH approaches based on structured quantizers because of how well it adapts to diverse real datasets \cite{16}.

At a high level, the KLSH algorithm works in three phases as follows:

1. k-Means clustering. The k-means algorithm is run on a learning dataset to discover $k$ clusters.

2. Database indexing. Each database point is placed in a bucket corresponding to one of the $k$ clusters it falls into.

3. Query answering. Each query finds the cluster it falls into and searches through the corresponding bucket to find the nearest neighbors.
KLSH is a member of the family of locality-sensitive hashing functions. Since each point falls into one of $k$ clusters, KLSH’s indexing method can be seen as a dimensionality reduction onto one dimension. Different sets of clusters can be seen as different hash functions. Query answering happens similarly to other LSH algorithms.

Since the k-means algorithm is central to KLSH, we’ll study the algorithm further.

2.3 THE K-MEANS ALGORITHM

The k-means algorithm is a popular method for automatically partitioning vector-based data into a given number of clusters $k$. It does this by trying to find $k$ centroids (mean points) that minimize squared distance to the points closest to them.

The problem itself is NP-Hard, even for $k = 2$, but simple heuristic algorithms have been developed that quickly converge to local minima. The most popular method, called Lloyd’s algorithm, works as follows:

1. Assignment step. Assign each data point to its nearest centroid.
2. Update step. Move each centroid to the mean of its assigned points.

Given a set of initial centroids, iterations of these two steps move them toward a local minimum until they finally converge. Initial centroids can be picked in a variety of ways, the simplest of which is random sampling from the dataset.

As we can see, k-means is a relatively simple algorithm. It has been shown to work well in practice for a wide variety of applications. For this reason, it has been studied extensively for more than fifty years. In fact, it has been identified as one of the top-ten algorithms in data mining.

The algorithm is useful in the AKNN context because it allows us to construct data-adaptive partitions in the original space. Structured LSH methods use regular, unit-size hyperspheres to partition the data. KLSH uses the Voronoi cells induced by the centroids to partition the data. Because these Voronoi cells can vary in shape and size and are more likely to occur in dense regions of the space, they are much better at adaptively partitioning the dataset than regular hyperspheres.

The weaknesses of k-means clustering in the AKNN context are two-fold. Firstly, the algorithm requires the user to input $k$, the number of clusters to find (which translates to the number of buckets in the index). It’s hard to know how many clusters exist in a dataset apriori, so practitioners usually resort to trying different values of $k$ and evaluating the results. There has been work in automatically
computing the number of clusters in a dataset that may alleviate this problem \cite{19}. A second problem is its high sensitivity to the choice of initial centroids. Different initializations will lead to different local minima. For the AKNN application, a bad convergence translates to severe load imbalance among the index buckets. Better initialization methods \cite{3} \cite{2} that take into account the distribution of the data are available to alleviate this problem.

In some contexts, the cluster model of k-means algorithm can be a weakness, but not in AKNN. The algorithm tends toward spherical, identically-sized clusters. For many applications, this approach might not be powerful enough. (It may be valuable to recognize clusters of different shapes and sizes.) For AKNN purposes, this model is not a limitation, but in fact a strength. Spherical clusters work well because then, centroids more accurately summarize their clusters. Identically-sized clusters help improve load-balance among index buckets.
With the conceptual background of KLSH in place, the details of the implementation of KLSH are described in this chapter. Since the algorithm can naturally be split into three phases, we’ll consider their implementations in sequence.

3.1 ALGORITHMS

3.1.1 The k-Means Clustering Phase

The clustering phase of the algorithm discovers a group of centroids that partition the database into Voronoi cells. At a high level, it creates a sample of the database and runs the k-means algorithm on the sample to create \( k \) centroids. These centroids are then used to index the database; each point in the database is placed in a bucket corresponding to its closest centroid. So, each bucket holds the points within a Voronoi cell induced by the corresponding centroid.

The database is sampled to produce the set of training points. Sampling is necessary for large databases because running the k-means algorithm on the full database can be expensive. Running k-means on the entire database will use all the information available about the distribution of the database to find the centroids. However, using the full database is unlikely to be much better than a well-sized sample because:

1. the results of k-means depend heavily on the initial conditions
2. the sample distribution is likely to mirror that of the database

Once a sample is selected, the rest of the phase consists of running Lloyd’s algorithm on the sample dataset. The algorithm for the clustering phase is in algorithm 1. The implementation uses the fact that the only reason to keep track of which points belong to which centroids is to create the new centroids. Therefore, it doesn’t add points to lists corresponding to each of the centroids, but instead just adds the points to running sum vectors. To compute the new centroids for the next iteration, the sum vectors are divided by the number of points that have been added to them. If no points fall into the re-
gion of a centroid, a random new centroid is chosen instead. (It’s not useful to have a centroid in a sparse part of the space.)

**Algorithm 1** The k-Means Clustering Phase

**Input:** database $D$, sample size $s$, centroid count $k$, iterations $itr$

**Output:** list of centroids $C$

1. insert $s$ points randomly from database into array $S$
2. initialize array $C$ with $k$ points randomly picked from $S$
3. initialize array $N$ with $k$ zeroes
4. for $i = 0 \rightarrow itr$ do
5.   for point $p$ in $S$ do
6.     find id $l$ of closest centroid to $p$
7.     add $p$ to $C[l]$
8.     increment $N[l]$
9.   end for
10. for all $c = 0 \rightarrow k$ do
11.   if $N[c] = 0$ then
12.     set $C[c]$ to random point in database
13.   else
14.     set $C[c]$ to $C[c]/N[c]$
15.     set $N[c]$ to 0
16.   end if
17. end for
18. return $C$

The sample set is constructed by generating random indexes into the database. The set of centroids is initialized randomly in a similar manner as well. Most of the computation in this phase is in finding the closest centroid to a point, which requires a distance computation to each centroid.

3.1.2 The Indexing Phase

The second phase is the indexing phase, which uses the centroids computed in the clustering phase to place database points into buckets. Each database point is compared with all the centroids, and is placed into the bucket corresponding to the closest one. The algorithm for the indexing phase is in algorithm 2.

The implementation of this phase is quite straightforward. Each index is a list of lists, implemented as a C++ vector that contains vectors. Nearly all of the computation in this phase happens in finding which centroid group each point belongs to.
Algorithm 2 The Indexing Phase

Input: database $D$, centroid group $C$
Output: index (list of lists) $I$

1: initialize array $I$ with $|C|$ empty buckets (lists)
2: for point $p$ in $D$ do
3:   find id $l$ of closest centroid to $p$
4:   append $p$ to bucket $I[l]$
5: end for
6: return $I$

3.1.3 The Query Answering Phase

The final phase is the query answering phase, where the index is used to find the $k$ nearest neighbors for each query. The algorithm for this phase is in algorithm 3.

Algorithm 3 The Query Answering Phase

Input: index $I$, queries $Q$, nearest neighbors count $k$
Output: answers (list of lists) $A$

1: initialize list of lists $A$ of to hold results for each query
2: for query point $q$ in $Q$ do
3:   initialize heap $h$ to track nearest neighbors
4:   find id $l$ of closest centroid in $I$ to $q$
5:   for point $p$ in $I[l]$ do
6:     if $p$ is closer than the farthest point $f$ in $h$ then
7:       remove $f$ from $h$
8:       add $p$ to $h$
9:     end if
10:   end for
11:   add points in $h$ to $A[q]$
12: end for
13: return $A$

Each query finds which Voronoi cell it falls into and finds the nearest neighbors within the cell. This translates to each query finding its nearest centroid and searching through the points in the corresponding bucket. The closest centroid is found as in the previous phases. The nearest neighbors are found by searching through the bucket while keeping track of the closest points in a max-heap ordered by distance to the query point. A heap is an efficient structure to use for this case because finding the worst point can be done in $O(1)$ time and inserting and deleting new near neighbors points can be done in $O(\log(n))$ time. The heap structure is reused between iterations to reduce the cost of dynamic memory operations. The list of answers is preallocated for the same reason.
Finding the nearest centroids to a given query point is expensive as in previous phases. The expense of searching for the closest points in the index bucket can be high, depending on the number of points in the bucket.

### 3.2 Experimental Methodology

#### 3.2.1 Evaluation Metrics

AKNN methods can be evaluated in two ways: running time and recall. Running time is the wall-clock time taken by the program to index the database and answer the queries. This measure encompasses all the computation required by the program to solve the AKNN problem. The Boost CPU timer library [9] is used to time the phases; its resolution is sufficient for the times we encounter. File input/output operations are not included.

Average recall is a measure of the quality of the AKNN algorithm. It’s defined as the average percentage of the $k$ nearest neighbors returned for a query that occur in the correct $k$ nearest neighbors. The correct nearest neighbors can be computed using the brute force linear scan method or by referring to a precomputed ground truth file. Note: recall doesn’t take into account the ordering of the returned near neighbors.

Intuitively, it’s apparent that running time and recall are at odds with each other. We can simply run the linear scan algorithm to guarantee 100% recall at the expense of time. On the other hand, we can do a minimum of work to achieve a low recall. In light of this trade-off, a practical question to ask is: how fast can we get the algorithm to run at a certain recall rate? Depending on the application, various recall levels can be acceptable.

#### 3.2.2 Experimental Setup

The database used in this evaluation is the INRIA Holidays collection of 128-dimensional image descriptors. Each of the 1 million vectors in the database is the result of running a different image through SIFT, a popular image descriptor algorithm, with the standard settings [13]. The images are photos of vacation scenes; each series of photos within the dataset contains the same location with slightly adjusted photo capture settings. For this dataset, the nearest neighbors for a vector correspond to images taken at the same location.

Two sets of queries are used: SIFT10K (10,000 queries) and SIFT100K (100,000) queries. These query sets were extracted from the INRIA Holidays dataset as well. The purpose of using two query sets is to be able to learn how the various parameters can affect the total running time with different query loads.
Each experiment was run on a single node of the Stampede supercomputer \cite{20} at the Texas Advanced Computing Center (TACC). Each node has two 2.7 GHz 8-core Xeon E5 (Sandy-Bridge) processors with 32 GB of RAM.

The base version of the program has been compiled with GCC 4.7.1 with the following flags: -O3 -march=native -funroll-loops -fprefetch-loop-arrays. (The -ffast-math and -openmp flags are added later.)

### 3.3 Analysis of KLSH

In this section, we’ll start with an analysis of the performance of the base KLSH implementation with respect to its parameters. These results will be help contextualize the various modifications to the algorithm described in the next section.

![Centroid Count vs Time and Average Recall](image)

**Figure 1:** The effect of centroid count on time and recall. Query set: SIFT10K. Parameters: sample size = 20,000, iterations = 4, nearest neighbors = 10

The number of centroids $C$ used in KLSH affects all three phases: the cost of running k-means to find the $C$ centroids, the cost of finding the closest centroid in the indexing and answering phases, and the number of points that have to be searched through in an index bucket.
3.3 Analysis of KLSH

In essence, $C$ reflects the tradeoff between being more selective about which points to look through and the loss of accuracy that comes with not looking through all the points.

Figure 2: Close up view of the effect of centroid count on time and recall, without $C = 1$. Query set: SIFT10K. Parameters: sample size = 20,000, iterations = 4, nearest neighbors = 10

As figure 1 shows, there’s a strong relationship between $C$ and the performance of the algorithm. If $C = 1$, the algorithm degenerates into linear search. Since it searches through the entire database to find the nearest neighbors, its recall is perfect. But, as we increase $C$, the total time taken by the algorithm drastically decreases. More centroids mean more buckets and fewer points to sift through; but, as result, average recall drops since it’s likely that near neighbors are actually in other buckets. With 300 centroids, only half of the nearest 10 neighbors are in the bucket corresponding to the closest centroid. As $C$ increases, the indexing time also increases, to the point that it offsets the gains made in the answering phase. (Figure 2) Also, it’s apparent that the k-means phase is much less expensive than the later phases. Indexing time is linearly dependent on $C$.

The number of iterations $I$ reflects the effort spent in k-means phase to move the centroids to the center of nearby dense regions. With
Figure 3: The effect of iteration count on time and recall. Query set: SIFT10K. Parameters: sample size = 20,000, centroids = 100, nearest neighbors = 10
zero iterations, the k-means phase degenerates into just picking random points as centroids. These are unlikely to serve as good centroids because they’re unlikely to be at the center of dense regions. (But, they’re likely to be somewhere near a dense region since initial centroids are picked randomly from the underlying database’s distribution.)

Figure 3 shows that running a few iterations is helpful, but there isn’t much benefit by running more. This behavior can be explained by the observing that the centroids move the most at the beginning of k-means. Since subsequent iterations move the centroids by a small distance, the marginal benefit of the centroid being even closer to a cluster is low. Because the cost of more iterations increases linearly but the benefits do not, just a few iterations is enough. This result is surprising as it’s mentioned in the literature that $I = 20$ works well in practice, whereas the results on SIFT10K indicate that as few as 4 iterations is optimal [16].

Figure 4: The effect of sample size on time and recall. Query set: SIFT10K. Parameters: iterations = 4, centroids = 100, nearest neighbors = 10

The sample size $S$ determines the amount of information we’d like to take into account when determining the centroids. A larger sam-
ple size increases the chance that the computed centroids effectively represent the underlying database distribution, but also increases the time spent in generating a random set of points and running k-means on them.

Since the algorithm randomly selects the sample points and the starting centroids in the k-means phase, it is not deterministic. However, this isn’t a problem in practice with a sufficient sample size - the variability in the total time and average recall is minimal. (However, outliers can still occur for a variety of system-level reasons.)

Sample size results show that for small query sets like SIFT10K, there isn’t much benefit from a large sample size. The expense of a larger sample size is not worth the gains in answering time from a better index. There’s virtually no difference in the recall levels due to sample size. It’s surprising that such a small number of samples can generate effective centroids. (For example, a sample size of 20,000 is 2% of the database.)

Figure 5: The effect of sample size on time and recall on a big query set. Query set: SIFT100K. Parameters: iterations = 4, centroids = 100, nearest neighbors = 10

For bigger query sets like SIFT100K, the behavior with regard to C is the same, as expected. But, unexpectedly, the effect of I and S (min-
imal gains after a certain value) is the same as well. The hypothesis that bigger query sets can afford to spend more time in the precomputation stage for higher gains in the answering stage doesn’t hold up. The case for sample size on a the large SIFT100K query set is in figure 5.

![K (number of nearest neighbors) vs Time and Average Recall](image)

Figure 6: The effect of K on time and recall. Query set: SIFT100K. Parameters: centroids = 100, iterations = 4, sample size = 20,000

And finally, how does the number of nearest neighbors K affect time and recall? It’s expected that as the K increases, the probability of near neighbors falling into different buckets increases, reducing recall. At the same time, the total time taken shouldn’t increase much. The cost of searching through the entire bucket is the same regardless of K. The only difference is the cost of adding and removing from the heap data structure that tracks the nearest neighbors; since these operations are $O(\log(n))$ and happen fairly infrequently, they don’t add a great amount of computation to the query answering process. And the results (figure 6) support this argument. It’s surprising how little both total time and recall are impacted as K increases. Since the size of the buckets is quite large (1,000,000 database vectors stored.
in 100 buckets = 10,000 points per bucket, on average) there’s a high chance that the

These performance results give us several insights:

1. Drawing a large sample and performing more than a few iterations in the k-means phase is not beneficial for both small and large query sets.

2. The k-means phase takes a small amount of time compared to the indexing and answering phases. And, even for small query sets, the answering phase dominates the total time.

3. Using parameters that make the algorithm run fast ($C = 100, I = 4, S = 20,000$) results in a very low average recall ($50 - 60\%$).

3.4 Modifications to KLSH

Based on the previous analysis, it’s apparent that the algorithm needs to be improved: recall is far too low for the faster parameter sets, and the total time required to achieve high recall is far too high. This section describes various modifications that can be done to KLSH to improve its accuracy and speed.

3.4.1 Overall Optimizations

First, we discuss three optimizations that overall greatly improved the speed of the implementation. Initial profiling revealed that the Euclidean distance computations took up 97% of the program’s time. Nearly all of the work done by the program is in finding the closest centroid or looking through the index buckets. (In a typical run of KLSH with SIFT10K, the distance function is called more than 1.2 billion times!) Since distance computations are only used to compare distances to points, it suffices to just compute distance squared, thus avoiding the costly square root function.

A second optimization for distance computation is vectorization. Vectorization allows multiple floating-point computations to happen in one clock cycle. Vectorized code can be handwritten using compiler intrinsics, but it can be tedious and error-prone. Modern compilers are able to automatically vectorize certain code patterns. The distance-squared function is not automatically vectorized because of floating-point semantics. The function computes differences between two arrays, squares those differences, and adds them to an ongoing total. Such a computation is seen as a floating-point reduction, which isn’t automatically parallelized. Since the results of floating-point operations can vary depending on the order of the operations, the compiler will not automatically vectorize floating-point reductions, taking the cautious route. KLSH doesn’t require a precise distance
function because the differences in distances between points are far greater than could be perturbed by floating point operation error. So, it's safe to enable the compiler’s fast math flag (-ffast-math in GCC), which enables it to optimize the distance-squared function through vectorization. This optimization resulted in an across-the-board 40% reduction in total time.

Another change that had a significant impact on the speed of the program is flattening the memory layout for oft-accessed data structures. Previously, the point databases stored pointers to point objects, which contained a pointer to the float array in memory that held the point’s data. Point data is accessed often - twice per distance computation. And, chasing pointers can be very expensive because of having to go through the memory management unit. Storing all of the points contiguously in a long float array sped up the program by 20-30%.

3.4.2 Improving Recall

Average recall is improved by two modifications: multiple indexes and multiprobing. Recall suffers when the nearest neighbors of a query point are not present in the bucket it searches through in the answering phase. This situation can occur only when the nearest neighbors are in different Voronoi cells. There are two ways of overcoming this problem.

The first is to partition the space several times, creating several sets of Voronoi cells, so that it’s likely that the nearest neighbors will be present at least one of the cells a query point falls into. This multi-indexing method can be implemented by computing multiple indexes with different sets of centroids. In a locality-sensitive hashing context, this method is analogous to using multiple hash functions. Using multiple sets of centroids alleviates some of the problems with how k-means converges to local minima.

The second is to look through multiple nearby Voronoi cells. If a query point falls near the boundary of its a cell, searching through just its own cell would yield only a fraction of the nearest neighbors. So, why not look in nearby regions? We can do just that by searching through the buckets corresponding to several nearby centroids. This technique is called multi-probing.

Both of these methods increase the time required to answer the query. Using multiple indexes requires multiple k-means and indexing steps. In the answering phase, the nearest centroid has to be found for each index and the corresponding buckets have to be searched. One important optimization for multi-indexing is keeping track of points we have already seen in a bitset; it’s not useful to recompute the distances to points in the database if we have already considered them while looking through another index. This opti-
3.4 Modifications to Klsh

Modification improves overall performance by 5-10% for many of the best parameter sets. Multiprobing requires keeping track of multiple near centroids and looking through their buckets; its impact is only in the answering phase. Of course, these two methods are not exclusive.

![Multiple Indexes vs Time and Average Recall](image)

Figure 7: The effect of multiple indexes on time and recall. Query set: SIFT10K. Parameters: iterations = 4, centroids = 100, nearest neighbors = 10, sample size = 20,000, probes = 1

The results of varying the number of indexes and probes are shown in figure 7 and figure 8, respectively. We see that using several indexes adds time to all the phases. The time spent in the answering phase is not linear in the number of indexes because we don’t recompute distances to points seen in other tables. (Each new index adds only a small number of new points.) In both cases, we can see that using multi-indexing and multi-probing increases recall significantly. In fact, we can get arbitrarily close to perfect recall with more effort.

We have looked at multi-indexing and multi-probing separately, but since we can use them together it’s useful to note that in practice, using just a few indexes (2-3) and a few probes (2-3) suffices to get very good average recall (> 95%). Different indexes offer different “views” of the database and probing multiple times takes advantage of more of the information that’s already present in the indexes.
Figure 8: The effect of multiple probes on time and recall. Query set: SIFT10K. Parameters: iterations = 4, centroids = 100, nearest neighbors = 10, sample size = 20,000, indexes = 1
Parallelization

Parallelizing the KLSH implementation yields significant performance benefits. The program is parallelized using OpenMP, which provides a simple interface for automatically parallelizing code for shared memory machines. The framework was useful for this application because most of the program’s work happens in for-loops where each iteration is independent; work isn’t dynamically generated.

In the k-means phase (algorithm 1), both sampling points and the k-means iterations can be done in parallel. The work of picking \( S \) samples is divided equally among \( P \) threads. Each iteration of k-means, the work of finding the id of the closest centroid for each point is divided in the same way. The code that adds points to the sum vectors of their closest neighbors needs to be in a critical section as multiple threads might be trying to add to the same sum vector.

In the indexing phase (algorithm 2), finding which bucket a database point should be placed into can be done in parallel. However, adding to the buckets themselves must be done one at a time, so a critical section is required.

And finally, in the query answering phase (algorithm 3), each query is given to a different thread. Since queries access the indexes and the database in a read-only manner, there are no dependencies between queries. Once the answers have been found, they are written to a preallocated answers array.

The results of parallelizing the implementation are shown in figure 9 and 10. As we expect, recall doesn’t change as we use more threads, since the computations are the same regardless of the number of threads. There is a clear improvement in the speed of the program as the number of threads increases. With 16 threads, it takes an average of 7.78 seconds (down from 76 seconds for the serial implementation) to run k-means on a sample of 20,000, index the 1,000,000 database vectors, and retrieve the nearest neighbors for 10,000 queries. This represents an overall speedup of 9.7x on a good parameter set. (With perfect speedup, the program would run in 4.7 seconds with 16 threads.)

There might be several factors that are reducing overall speedup. The answering phase takes a greatly varying amount of time per query depending on the number of items in the buckets and which elements are in the caches. A problem that results is very uneven times taken by threads if they were given identical numbers of queries. To help alleviate this problem, a dynamic work scheduler is used. Indeed, dynamic schedulers surprisingly seem to beat static and guided schedulers in most of the program’s parallelized for loops. So, the cost of contention on a worklist might be an issue, especially because the worklist is being accessed by two cores at once. (OpenMP’s worklist is unlikely to be processor-aware.) I’ve tried to ensure that no
Figure 9: The effect of multithreading on time and recall. Query set: SIFT10k. Parameters: iterations = 4, centroids = 240, nearest neighbors = 10, indexes = 3, probes = 3, sample size = 20,000
Figure 10: Speedup over the serial implementation. Query set: SIFT10K. Parameters: iterations = 4, centroids = 240, nearest neighbors = 10, indexes = 3, probes = 3, sample size = 20,000
dynamic memory operations are happening in the parallelized for loops, but if there are, contention for the memory allocator can be a concern.

3.4.4 Other optimizations

We also tried implementing several other modifications, which didn’t improve the KLSH implementation significantly.

First, we tried other methods of initializing k-means. Randomly picking centroids at the beginning of the k-means step is quick but can lead to terrible convergences. We implemented k-means++, a smart initialization technique that picks centroids that are spread apart in the database [2]. K-means++ is quite expensive and didn’t show any improvement in the later stages of KLSH. This behavior shows that a random initialization of k-means is probably good enough and provides enough information to build good indexes. Even if the centroids picked aren’t the absolute best, there’s leeway since we build multiple indexes and probe multiple times.

A second optimization was to reduce the number of distance computations by precomputing pairwise distances between all the centroids in the indexing phase. These distances allowed us to omit some distance computations by using the triangle inequality. Although this optimization reduced the number of distance computations computed during the indexing phase by 25%, there wasn’t much of a speed increase since the indexing phase is small and any improvement was still being overshadowed by parallelization overheads.

A final optimization was to use an approximate k-means algorithm, namely minibatch k-means [18]. For each k-means iteration, this algorithm uses the information from a different small sample from the database. This approach required many more iterations to achieve the same centroid quality as Lloyd’s algorithm. It also required the tuning of yet another parameter, the sample size per iteration. So, in practice it’s not worth using this algorithm, especially since the k-means step is a very small portion of the total runtime.

3.4.5 Memory Behavior

Though we’ve been concentrating our efforts on improving average recall and total time, it’s important to note the memory usage of this algorithm. Almost all of the memory is taken up by the database of vectors. With one million 128-dimensional vectors, the database we’re using takes up almost 500 MB in RAM. The small SIFT10K query set uses up just 5 MB while the larger SIFT100K query set uses about 50 MB. Further, the program creates several indexes, each of which contain all of the point ids (1,000,000 integers in this case.) These indices take less than 5 MB each. Overall, under the SIFT10K workload, the
program uses no more than 600 MB of RAM, almost all of it the point
database. This compares quite favorably to other locality-sensitive
hashing methods, which can require so many indexes that they take
up as much space as the database \cite{16}.

3.5 Future Work

In the previous sections, we have seen an analysis of the KLSH al-
gorithm and several modifications that can be made to improve the
performance of the algorithm. There are several challenges that fu-
ture work can tackle.

Improving the scaling of the the KLSH implementation, especially
its query answering phase, would increase the program’s speed sig-
nificantly. More work on exactly what is causing sublinear scaling
behavior is needed.

The main bottleneck in KLSH is computing distances. With high
query loads, almost all the time is spent searching through buckets
in the answering phase. Reducing the number of distance computa-
tions that need to be done would go a long way toward improving
the speed of KLSH. One possible avenue is to construct a hierarchical
index using k-means at several levels. The computation in the pre-
processing steps will be higher, but having to search smaller buckets
during the short-list phase could improve speed. This approach has
been studied, but it’s unclear whether it helps in general \cite{16}. 
CONCLUSION

In the previous chapters, we started with the background for understanding k-means locality sensitive hashing (KLSH) and proceeded to analyze both the algorithm and its implementation.

The main contribution of this work is an in-depth analysis of the KLSH algorithm from a practical perspective. The following are the key results:

1. A few surprising discoveries about the effect of various parameters on the performance of KLSH: during the k-means phase, a surprisingly small sample with just a few iterations performs quite well. In addition, running time (and even recall, after a certain point) doesn’t vary much with $k$, the number of nearest neighbors sought per query.

2. Although the accuracy of the basic version of the algorithm is low, it can be extended to be as accurate as needed with multi-indexing and multi-probing.

3. Various optimizations such as vectorization and parallelization can bring vast improvements in running time.

This work shows that KLSH is practical for large, high-dimensional datasets. Further research into improving the speed of KLSH is needed to extend its usefulness further.


