Scalable Nearest Neighbor Algorithms for High Dimensional Data

Marius Muja, *Member, IEEE* and David G. Lowe, *Member, IEEE*

Abstract—For many computer vision and machine learning problems, large training sets are key for good performance. However, the most computationally expensive part of many computer vision and machine learning algorithms consists of finding nearest neighbor matches to high dimensional vectors that represent the training data. We propose new algorithms for approximate nearest neighbor matching and evaluate and compare them with previous algorithms. For matching high dimensional features, we find two algorithms to be the most efficient: the randomized k-d forest and a new algorithm proposed in this paper, the priority search k-means tree. We also propose a new algorithm for matching binary features by searching multiple hierarchical clustering trees and show it outperforms methods typically used in the literature. We show that the optimal nearest neighbor algorithm and its parameters depend on the data set characteristics and describe an automated configuration procedure for finding the best algorithm to search a particular data set. In order to scale to very large data sets that would otherwise not fit in the memory of a single machine, we propose a distributed nearest neighbor matching framework that can be used with any of the algorithms described in the paper. All this research has been released as an open source library called fast library for approximate nearest neighbors (FLANN), which has been incorporated into OpenCV and is now one of the most popular libraries for nearest neighbor matching.

Index Terms—Nearest neighbor search, big data, approximate search, algorithm configuration

1 Introduction

The most computationally expensive part of many computer vision algorithms consists of searching for the most similar matches to high-dimensional vectors, also referred to as nearest neighbor matching. Having an efficient algorithm for performing fast nearest neighbor matching in large data sets can bring speed improvements of several orders of magnitude to many applications. Examples of such problems include finding the best matches for local image features in large data sets [1], [2] clustering local features into visual words using the k-means or similar algorithms [3], global image feature matching for scene recognition [4], human pose estimation [5], matching deformable shapes for object recognition [6] or performing normalized cross-correlation (NCC) to compare image patches in large data sets [7]. The nearest neighbor search problem is also of major importance in many other applications, including machine learning, document retrieval, data compression, bio-informatics, and data analysis.

It has been shown that using large training sets is key to obtaining good real-life performance from many computer vision methods [2], [4], [7]. Today the Internet is a vast resource for such training data [8], but for large data sets the performance of the algorithms employed quickly becomes a key issue.

When working with high dimensional features, as with most of those encountered in computer vision applications (image patches, local descriptors, global image descriptors), there is often no known nearest-neighbor search algorithm that is exact and has acceptable performance. To obtain a speed improvement, many practical applications are forced to settle for an approximate search, in which not all the neighbors returned are exact, meaning some are approximate but typically still close to the exact neighbors. In practice it is common for approximate nearest neighbor search algorithms to provide more than 95 percent of the correct neighbors and still be two or more orders of magnitude faster than linear search. In many cases the nearest neighbor search is just a part of a larger application containing other approximations and there is very little loss in performance from using approximate rather than exact neighbors.

In this paper we evaluate the most promising nearest-neighbor search algorithms in the literature, propose new algorithms and improvements to existing ones, present a method for performing automatic algorithm selection and parameter optimization, and discuss the problem of scaling to very large data sets using compute clusters. We have released all this work as an open source library named fast library for approximate nearest neighbors (FLANN).

1.1 Definitions and Notation

In this paper we are concerned with the problem of efficient nearest neighbor search in metric spaces. The nearest neighbor search in a metric space can be defined as follows: given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in a metric space M and a query point $q \in M$, find the element $\mathrm{NN}(q, P) \in P$ that is the

- *M. Muja is with BitLit Media Inc, Vancouver, BC, Canada. E-mail: mariusm@cs.ubc.ca.*
- *D.G. Lowe is with the Computer Science Department, University of British Columbia (UBC), 2366 Main Mall, Vancouver, BC V6T 1Z4, Canada. E-mail: lowe@cs.ubc.ca.*

Manuscript received 26 Aug. 2013; revised 14 Feb. 2014; accepted 1 Apr. 2014. Date of publication 30 Apr. 2014; date of current version 9 Oct. 2014. Recommended for acceptance by T. Tuytelaars.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPAMI.2014.2321376

closest to q with respect to a metric distance $d : M \times M \rightarrow \mathbb{R}$:

$$\mathrm{NN}(q, P) = argmin_{x \in P}\ d(q, x).$$

The *nearest neighbor problem* consists of finding a method to pre-process the set P such that the operation $\mathrm{NN}(q, P)$ can be performed efficiently.

We are often interested in finding not just the first closest neighbor, but several closest neighbors. In this case, the search can be performed in several ways, depending on the number of neighbors returned and their distance to the query point: *K-nearest neighbor (KNN) search* where the goal is to find the closest K points from the query point and *radius nearest neighbor search (RNN)*, where the goal is to find all the points located closer than some distance R from the query point.

We define the *K-nearest neighbor* search more formally in the following manner:

$$\mathrm{KNN}(q, P, K) = A,$$

where A is a set that satisfies the following conditions:

$$|A| = K, A \subseteq P$$

$$\forall x \in A, y \in P - A, d(q, x) \leq d(q, y).$$

The K-nearest neighbor search has the property that it will always return exactly K neighbors (if there are at least K points in P).

The *radius nearest neighbor* search can be defined as follows:

$$\mathrm{RNN}(q, P, R) = \{p \in P, d(q, p)\ <\ R\}.$$

Depending on how the value R is chosen, the radius search can return any number of points between zero and the whole data set. In practice, passing a large value R to radius search and having the search return a large number of points is often very inefficient. *Radius K-nearest neighbor* (RKNN) search, is a combination of K-nearest neighbor search and radius search, where a limit can be placed on the number of points that the radius search should return:

$$\mathrm{RKNN}(q, P, K, R) = A,$$

such that

$$|A| \leq K, A \subseteq P$$

$$\forall x \in A, y \in P - A, d(q, x)\ <\ R \text{ and } d(q, x) \leq d(q, y).$$

2 Background

Nearest-neighbor search is a fundamental part of many computer vision algorithms and of significant importance in many other fields, so it has been widely studied. This section presents a review of previous work in this area.

2.1 Nearest Neighbor Matching Algorithms

We review the most widely used nearest neighbor techniques, classified in three categories: partitioning trees, hashing techniques and neighboring graph techniques.

2.1.1 Partitioning Trees

The kd-tree [9], [10] is one of the best known nearest neighbor algorithms. While very effective in low dimensionality spaces, its performance quickly decreases for high dimensional data.

Arya et al. [11] propose a variation of the k-d tree to be used for approximate search by considering $(1+\varepsilon)$-*approximate* nearest neighbors, points for which $dist(p, q) \leq (1+\varepsilon)dist(p^*, q)$ where p^* is the true nearest neighbor. The authors also propose the use of a priority queue to speed up the search. This method of approximating the nearest neighbor search is also referred to as "error bound" approximate search.

Another way of approximating the nearest neighbor search is by limiting the time spent during the search, or "time bound" approximate search. This method is proposed in [12] where the k-d tree search is stopped early after examining a fixed number of leaf nodes. In practice the time-constrained approximation criterion has been found to give better results than the error-constrained approximate search.

Multiple randomized k-d trees are proposed in [13] as a means to speed up approximate nearest-neighbor search. In [14] we perform a wide range of comparisons showing that the multiple randomized trees are one of the most effective methods for matching high dimensional data.

Variations of the k-d tree using non-axis-aligned partitioning hyperplanes have been proposed: the PCA-tree [15], the RP-tree [16], and the trinary projection tree [17]. We have not found such algorithms to be more efficient than a randomized k-d tree decomposition, as the overhead of evaluating multiple dimensions during search outweighed the benefit of the better space decomposition.

Another class of partitioning trees decompose the space using various clustering algorithms instead of using hyperplanes as in the case of the k-d tree and its variants. Example of such decompositions include the hierarchical k-means tree [18], the GNAT [19], the anchors hierarchy [20], the vp-tree [21], the cover tree [22] and the spill-tree [23]. Nister and Stewenius [24] propose the vocabulary tree, which is searched by accessing a single leaf of a hierarchical k-means tree. Leibe et al. [25] propose a ball-tree data structure constructed using a mixed partitional-agglomerative clustering algorithm. Schindler et al. [26] propose a new way of searching the hierarchical k-means tree. Philbin et al. [2] conducted experiments showing that an approximate flat vocabulary outperforms a vocabulary tree in a recognition task. In this paper we describe a modified k-means tree algorithm that we have found to give the best results for some data sets, while randomized k-d trees are best for others.

Jégou et al. [27] propose the product quantization approach in which they decompose the space into low dimensional subspaces and represent the data sets points by compact codes computed as quantization indices in these subspaces. The compact codes are efficiently compared to the query points using an asymmetric approximate distance. Babenko and Lempitsky [28] propose the inverted multi-index, obtained by replacing the standard quantization in an inverted index with product quantization, obtaining a denser subdivision of the search space. Both these methods are shown to be efficient at searching large data

sets and they should be considered for further evaluation and possible incorporation into FLANN.

2.1.2 Hashing Based Nearest Neighbor Techniques

Perhaps the best known hashing based nearest neighbor technique is locality sensitive hashing (LSH) [29], which uses a large number of hash functions with the property that the hashes of elements that are close to each other are also likely to be close. Variants of LSH such as multi-probe LSH [30] improves the high storage costs by reducing the number of hash tables, and LSH Forest [31] adapts better to the data without requiring hand tuning of parameters.

The performance of hashing methods is highly dependent on the quality of the hashing functions they use and a large body of research has been targeted at improving hashing methods by using data-dependent hashing functions computed using various learning techniques: parameter sensitive hashing [5], spectral hashing [32], randomized LSH hashing from learned metrics [33], kernelized LSH [34], learnt binary embeddings [35], shift-invariant kernel hashing [36], semi-supervised hashing [37], optimized kernel hashing [38] and complementary hashing [39].

The different LSH algorithms provide theoretical guarantees on the search quality and have been successfully used in a number of projects, however our experiments reported in Section 4, show that in practice they are usually outperformed by algorithms using space partitioning structures such as the randomized k-d trees and the priority search k-means tree.

2.1.3 Nearest Neighbor Graph Techniques

Nearest neighbor graph methods build a graph structure in which points are vertices and edges connect each point to its nearest neighbors. The query points are used to explore this graph using various strategies in order to get closer to their nearest neighbors. In [40] the authors select a few well separated elements in the graph as "seeds" and start the graph exploration from those seeds in a best-first fashion. Similarly, the authors of [41] perform a best-first exploration of the k-NN graph, but use a hill-climbing strategy and pick the starting points at random. They present recent experiments that compare favourably to randomized KD-trees, so the proposed algorithm should be considered for future evaluation and possible incorporation into FLANN.

The nearest neighbor graph methods suffer from a quite expensive construction of the k-NN graph structure. Wang et al. [42] improve the construction cost by building an approximate nearest neighbor graph.

2.2 Automatic Configuration of NN Algorithms

There have been hundreds of papers published on nearest neighbor search algorithms, but there has been little systematic comparison to guide the choice among algorithms and set their internal parameters. In practice, and in most of the nearest neighbor literature, setting the algorithm parameters is a manual process carried out by using various heuristics and rarely make use of more systematic approaches.

Bawa et al. [31] show that the performance of the standard LSH algorithm is critically dependent on the length of the hashing key and propose the LSH Forest, a self-tuning algorithm that eliminates this data dependent parameter.

In a previous paper [14] we have proposed an automatic nearest neighbor algorithm configuration method by combining grid search with a finer grained Nelder-Mead downhill simplex optimization process [43].

There has been extensive research on *algorithm configuration* methods [44], [45], however we are not aware of papers that apply such techniques to finding optimum parameters for nearest neighbor algorithms. Bergstra and Bengio [46] show that, except for small parameter spaces, random search can be a more efficient strategy for parameter optimization than grid search.

3 Fast Approximate NN Matching

Exact search is too costly for many applications, so this has generated interest in approximate nearest-neighbor search algorithms which return non-optimal neighbors in some cases, but can be orders of magnitude faster than exact search.

After evaluating many different algorithms for approximate nearest neighbor search on data sets with a wide range of dimensionality [14], [47], we have found that one of two algorithms gave the best performance: the *priority search k-means tree* or the *multiple randomized k-d trees*. These algorithms are described in the remainder of this section.

3.1 The Randomized k-d Tree Algorithm

The randomized k-d tree algorithm [13], is an approximate nearest neighbor search algorithm that builds multiple randomized k-d trees which are searched in parallel. The trees are built in a similar manner to the classic k-d tree [9], [10], with the difference that where the classic kd-tree algorithm splits data on the dimension with the highest variance, for the randomized k-d trees the split dimension is chosen randomly from the top N_D dimensions with the highest variance. We used the fixed value $N_D = 5$ in our implementation, as this performs well across all our data sets and does not benefit significantly from further tuning.

When searching the randomized k-d forest, a single priority queue is maintained across all the randomized trees. The priority queue is ordered by increasing distance to the decision boundary of each branch in the queue, so the search will explore first the closest leaves from all the trees. Once a data point has been examined (compared to the query point) inside a tree, it is marked in order to not be re-examined in another tree. The degree of approximation is determined by the maximum number of leaves to be visited (across all trees), returning the best nearest neighbor candidates found up to that point.

Fig. 1 shows the value of searching in many randomized kd-trees at the same time. It can be seen that the performance improves with the number of randomized trees up to a certain point (about 20 random trees in this case) and that increasing the number of random trees further leads to static or decreasing performance. The memory overhead of using multiple random trees increases linearly with the number of trees, so at some point the speedup may not justify the additional memory used.

Fig. 2 gives an intuition behind why exploring multiple randomized kd-tree improves the search performance. When the query point is close to one of the splitting hyperplanes, its nearest neighbor lies with almost equal

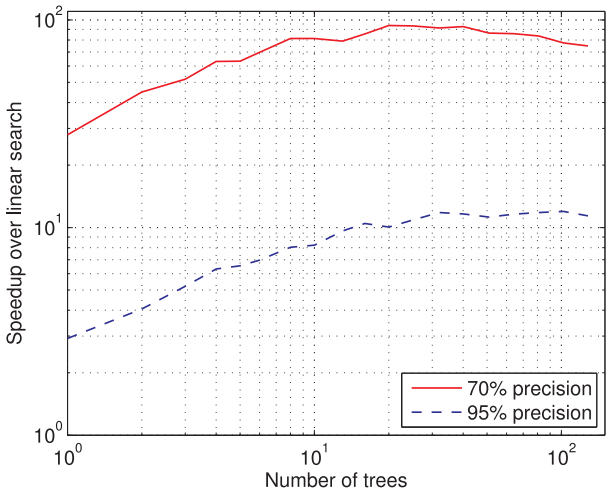


Fig. 1. Speedup obtained by using multiple randomized kd-trees (100K SIFT features data set).

probability on either side of the hyperplane and if it lies on the opposite side of the splitting hyperplane, further exploration of the tree is required before the cell containing it will be visited. Using multiple random decompositions increases the probability that in one of them the query point and its nearest neighbor will be in the same cell.

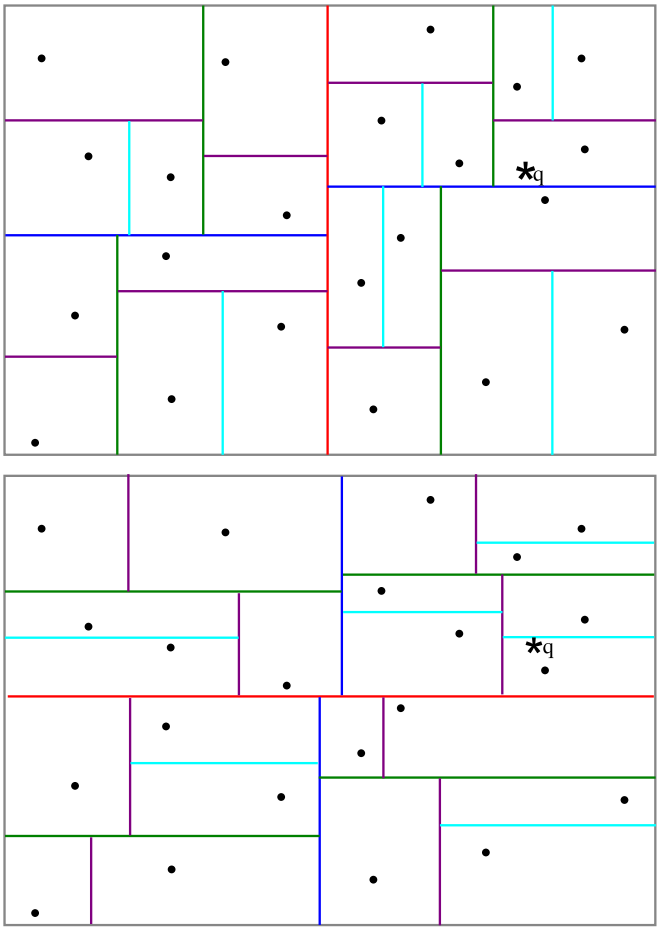


Fig. 2. Example of randomized kd-trees. The nearest neighbor is across a decision boundary from the query point in the first decomposition, however is in the same cell in the second decomposition.

3.2 The Priority Search K-Means Tree Algorithm

We have found the randomized k-d forest to be very effective in many situations, however on other data sets a different algorithm, the *priority search k-means tree*, has been more effective at finding approximate nearest neighbors, especially when a high precision is required. The priority search k-means tree tries to better exploit the natural structure existing in the data, by clustering the data points using the full distance across all dimensions, in contrast to the (randomized) k-d tree algorithm which only partitions the data based on one dimension at a time.

Nearest-neighbor algorithms that use hierarchical partitioning schemes based on clustering the data points have been previously proposed in the literature [18], [19], [24]. These algorithms differ in the way they construct the partitioning tree (whether using k-means, agglomerative or some other form of clustering) and especially in the strategies used for exploring the hierarchical tree. We have developed an improved version that explores the k-means tree using a *best-bin-first* strategy, by analogy to what has been found to significantly improve the performance of the approximate kd-tree searches.

3.2.1 Algorithm Description

The priority search k-means tree is constructed by partitioning the data points at each level into K distinct regions using k-means clustering, and then applying the same method recursively to the points in each region. The recursion is stopped when the number of points in a region is smaller than K (see Algorithm 1).

Algorithm 1 Building the priority search k-means tree

Input: features dataset D, branching factor K, maximum iterations I_{max}, centre selection algorithm to use C_{alg}
Output: k-means tree

if $|D| < K$ **then**
  create leaf node with the points in D
else
  $P \leftarrow$ *select K points from D using the* C_{alg} *algorithm*
  $converged \leftarrow$ **false**
  $iterations \leftarrow 0$
  while not *converged* **and** $iterations < I_{max}$ **do**
    $C \leftarrow$ *cluster the points in D around nearest centres P*
    $P_{new} \leftarrow$ *means of clusters in C*
    if P=P_{new} **then**
      $converged \leftarrow$ **true**
    end if
    $P \leftarrow P_{new}$
    $iterations \leftarrow iterations + 1$
  end while
  for *each cluster* $C_i \in C$ **do**
    create non-leaf node with center P_i
    recursively apply the algorithm to the points in C_i
  end for
end if

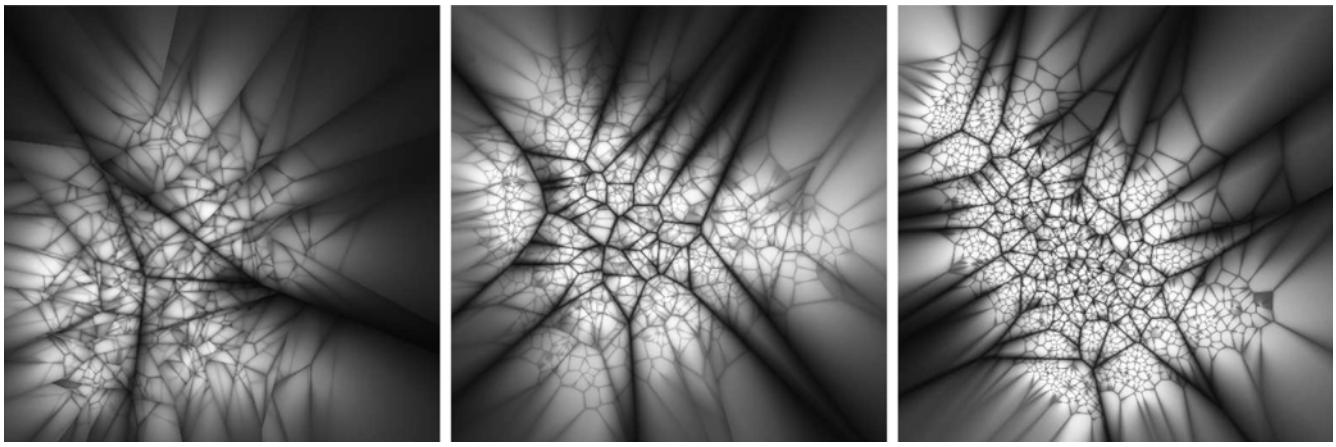


Fig. 3. Projections of priority search k-means trees constructed using different branching factors: 4, 32, 128. The projections are constructed using the same technique as in [26], gray values indicating the ratio between the distances to the nearest and the second-nearest cluster centre at each tree level, so that the darkest values (ratio ≈ 1) fall near the boundaries between k-means regions.

The tree is searched by initially traversing the tree from the root to the closest leaf, following at each inner node the branch with the closest cluster centre to the query point, and adding all unexplored branches along the path to a priority queue (see Algorithm 2). The priority queue is sorted in increasing distance from the query point to the boundary of the branch being added to the queue. After the initial tree traversal, the algorithm resumes traversing the tree, always starting with the top branch in the queue.

Algorithm 2 Searching the priority search k-means tree

Input: k-means tree T, query point Q, maximum number of points to examine L, number of neighbors K
Output: K nearest approximate neighbors of query point

procedure SEARCHKMEANSTREE(T,Q,L,K)
1: $count \leftarrow 0$
2: $PQ \leftarrow$ *empty priority queue*
3: $R \leftarrow$ *empty priority queue*
4: *call* TRAVERSEKMEANSTREE(T,PQ,R,$count$,Q)
5: **while** PQ not empty **and** $count < L$ **do**
6: $N \leftarrow$ *top of* PQ
7: *call* TRAVERSEKMEANSTREE(N,PQ,R,$count$,Q)
8: **end while**
9: **return** K *top points from* R

procedure TRAVERSEKMEANSTREE(N,PQ,R,$count$,Q)
1: **if** *node* N *is a leaf node* **then**
2: *search all the points in* N *and add them to* R
3: $count \leftarrow count + |N|$
4: **else**
5: $C \leftarrow$ *child nodes of* N
6: $C_q \leftarrow$ *closest node of* C *to query* Q
7: $C_p \leftarrow C \backslash C_q$
8: *add all nodes in* C_p *to* PQ
9: *call* TRAVERSETREE(C_q,PQ,R)
10: **end if**

The number of clusters K to use when partitioning the data at each node is a parameter of the algorithm, called the *branching factor* and choosing K is important for obtaining good search performance. In Section 3.4 we propose an algorithm for finding the optimum algorithm parameters, including the optimum branching factor. Fig. 3 contains a visualisation of several hierarchical k-means decompositions with different branching factors.

Another parameter of the priority search k-means tree is I_{max}, the *maximum number of iterations* to perform in the k-means clustering loop. Performing fewer iterations can substantially reduce the tree build time and results in a slightly less than optimal clustering (if we consider the sum of squared errors from the points to the cluster centres as the measure of optimality). However, we have observed that even when using a small number of iterations, the nearest neighbor search performance is similar to that of the tree constructed by running the clustering until convergence, as illustrated by Fig. 4. It can be seen that using as few as seven iterations we get more than 90 percent of the nearest-neighbor performance of the tree constructed using full convergence, but requiring less than 10 percent of the build time.

The algorithm to use when picking the initial centres in the k-means clustering can be controlled by the C_{alg} parameter. In our experiments (and in the FLANN library) we have

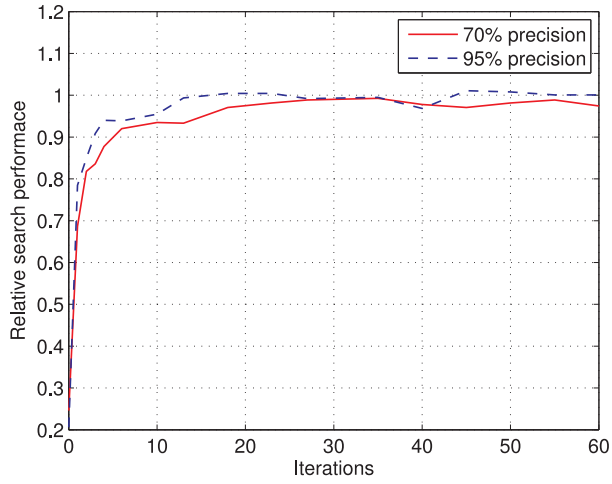


Fig. 4. The influence that the number of k-means iterations has on the search speed of the k-means tree. Figure shows the relative search time compared to the case of using full convergence.

used the following algorithms: random selection, Gonzales' algorithm (selecting the centres to be spaced apart from each other) and KMeans++ algorithm [48]. We have found that the initial cluster selection made only a small difference in terms of the overall search efficiency in most cases and that the random initial cluster selection is usually a good choice for the priority search k-means tree.

3.2.2 Analysis

When analysing the complexity of the priority search k-means tree, we consider the tree construction time, search time and the memory requirements for storing the tree.

Construction time complexity. During the construction of the k-means tree, a k-means clustering operation has to be performed for each inner node. Considering a node v with n_v associated data points, and assuming a maximum number of iterations I in the k-means clustering loop, the complexity of the clustering operation is $O(n_v dKI)$, where d represents the data dimensionality. Taking into account all the inner nodes on a level, we have $\sum n_v = n$, so the complexity of constructing a level in the tree is $O(ndKI)$. Assuming a balanced tree, the height of the tree will be $(\log n/\log K)$, resulting in a total tree construction cost of $O(ndKI(\log n/\log K))$.

Search time complexity. In case of the time constrained approximate nearest neighbor search, the algorithm stops after examining L data points. Considering a complete priority search k-means tree with branching factor K, the number of top down tree traversals required is L/K (each leaf node contains K points in a complete k-means tree). During each top-down traversal, the algorithm needs to check $O(\log n/\log K)$ inner nodes and one leaf node.

For each internal node, the algorithm has to find the branch closest to the query point, so it needs to compute the distances to all the cluster centres of the child nodes, an $O(Kd)$ operation. The unexplored branches are added to a priority queue, which can be accomplished in $O(K)$ amortized cost when using binomial heaps. For the leaf node the distance between the query and all the points in the leaf needs to be computed which takes $O(Kd)$ time. In summary the overall search cost is $O(Ld(\log n/\log K))$.

3.3 The Hierarchical Clustering Tree

Matching binary features is of increasing interest in the computer vision community with many binary visual descriptors being recently proposed: BRIEF [49], ORB [50], BRISK [51]. Many algorithms suitable for matching vector based features, such as the randomized kd-tree and priority search k-means tree, are either not efficient or not suitable for matching binary features (for example, the priority search k-means tree requires the points to be in a vector space where their dimensions can be independently averaged).

Binary descriptors are typically compared using the Hamming distance, which for binary data can be computed as a bitwise XOR operation followed by a bit count on the result (very efficient on computers with hardware support for counting the number of bits set in a word[1]).

This section briefly presents a new data structure and algorithm, called the *hierarchical clustering tree*, which we found to be very effective at matching binary features. For a more detailed description of this algorithm the reader is encouraged to consult [47] and [52].

The hierarchical clustering tree performs a decomposition of the search space by recursively clustering the input data set using random data points as the cluster centers of the non-leaf nodes (see Algorithm 3).

Algorithm 3 Building one hierarchical clustering tree

Input: features dataset D
Output: hierarchical clustering tree
Parameters: branching factor K, maximum leaf size S_L

```
if size of D < S_L then
    create leaf node with the points in D
else
    P ← select K points at random from D
    C ← cluster the points in D around nearest centers P
    for each cluster C_i ∈ C do
        create non-leaf node with center P_i
        recursively apply the algorithm to the points in C_i
    end for
end if
```

In contrast to the priority search k-means tree presented above, for which using more than one tree did not bring significant improvements, we have found that building multiple hierarchical clustering trees and searching them in parallel using a common priority queue (the same approach that has been found to work well for randomized kd-trees [13]) resulted in significant improvements in the search performance.

3.4 Automatic Selection of the Optimal Algorithm

Our experiments have revealed that the optimal algorithm for approximate nearest neighbor search is highly dependent on several factors such as the data dimensionality, size and structure of the data set (whether there is any correlation between the features in the data set) and the desired search precision. Additionally, each algorithm has a set of parameters that have significant influence on the search performance (e.g., number of randomized trees, branching factor, number of k-means iterations).

As we already mention in Section 2.2, the optimum parameters for a nearest neighbor algorithm are typically chosen manually, using various heuristics. In this section we propose a method for automatic selection of the best nearest neighbor algorithm to use for a particular data set and for choosing its optimum parameters.

By considering the nearest neighbor algorithm itself as a parameter of a generic nearest neighbor search routine A, the problem is reduced to determining the parameters $\theta \in \Theta$ that give the best solution, where Θ is also known as the *parameter configuration space*. This can be formulated as an optimization problem in the parameter configuration space:

$$\min_{\theta \in \Theta} c(\theta)$$

with $c : \Theta \rightarrow \mathbb{R}$ being a cost function indicating how well the search algorithm A, configured with the parameters θ, performs on the given input data.

1. The POPCNT instruction for modern x86_64 architectures.

We define the cost as a combination of the search time, tree build time, and tree memory overhead. Depending on the application, each of these three factors can have a different importance: in some cases we don't care much about the tree build time (if we will build the tree only once and use it for a large number of queries), while in other cases both the tree build time and search time must be small (if the tree is built on-line and searched a small number of times). There are also situations when we wish to limit the memory overhead if we work in memory constrained environments. We define the cost function as follows:

$$c(\theta) = \frac{s(\theta) + w_b b(\theta)}{\min_{\theta \in \Theta}(s(\theta) + w_b b(\theta))} + w_m m(\theta), \qquad (1)$$

where $s(\theta)$, $b(\theta)$ and $m(\theta)$ represent the search time, tree build time and memory overhead for the tree(s) constructed and queried with parameters θ. The memory overhead is measured as the ratio of the memory used by the tree(s) and the memory used by the data: $m(\theta) = m_t(\theta)/m_d$.

The weights w_b and w_m are used to control the relative importance of the build time and memory overhead in the overall cost. The build-time weight (w_b) controls the importance of the tree build time relative to the search time. Search time is defined as the time to search for the same number of points as there are in the tree. The time overhead is computed relative to the optimum time cost $\min_{\theta \in \Theta}(s(\theta) + w_b b(\theta))$, which is defined as the optimal search and build time if memory usage were not a factor.

We perform the above optimization in two steps: a global exploration of the parameter space using grid search, followed by a local optimization starting with the best solution found in the first step. The grid search is a feasible and effective approach in the first step because the number of parameters is relatively low. In the second step we use the Nelder-Mead downhill simplex method [43] to further locally explore the parameter space and fine-tune the best solution obtained in the first step. Although this does not guarantee a global minimum, our experiments have shown that the parameter values obtained are close to optimum in practice.

We use random sub-sampling cross-validation to generate the data and the query points when we run the optimization. In FLANN the optimization can be run on the full data set for the most accurate results or using just a fraction of the data set to have a faster auto-tuning process. The parameter selection needs to only be performed once for each type of data set, and the optimum parameter values can be saved and applied to all future data sets of the same type.

4 Experiments

For the experiments presented in this section we used a selection of data sets with a wide range of sizes and data dimensionality. Among the data sets used are the Winder/Brown patch data set [53], data sets of randomly sampled data of different dimensionality, data sets of SIFT features of different sizes obtained by sampling from the CD cover data set of [24] as well as a data set of SIFT features extracted from the overlapping images forming panoramas.

We measure the accuracy of an approximate nearest neighbor algorithm using the *search precision* (or just *precision*), defined as the fraction of the neighbors returned by the approximate algorithm which are exact nearest neighbors. We measure the search performance of an algorithm as the time required to perform a linear search divided by the time required to perform the approximate search and we refer to it as the *search speedup* or just *speedup*.

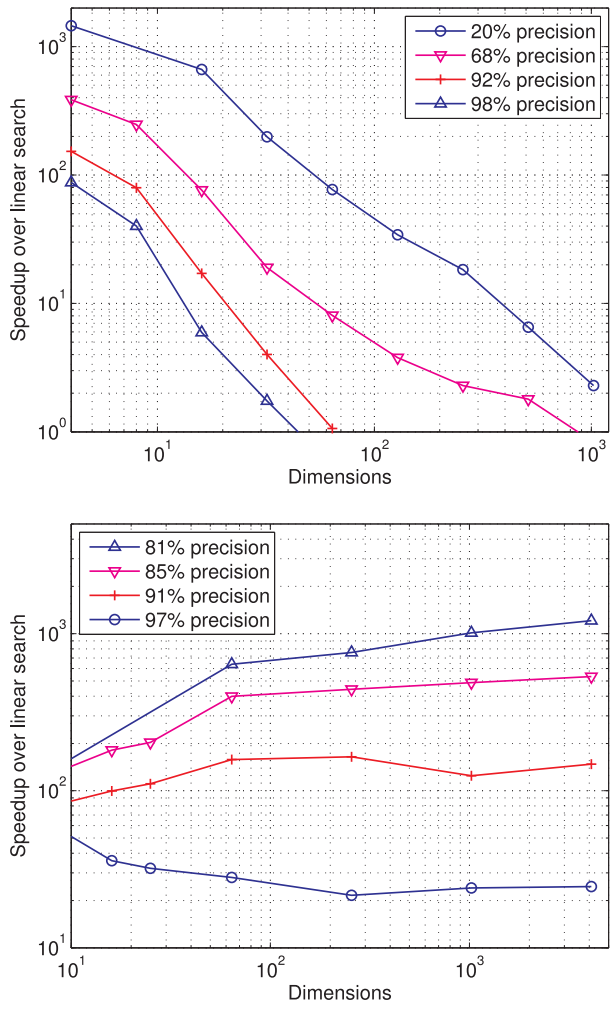


Fig. 5. Search efficiency for data of varying dimensionality. We experimented on both random vectors and image patches, with data sets of size 100K. The random vectors (top figure) represent the hardest case in which dimensions have no correlations, while most real-world problems behave more like the image patches (bottom figure).

4.1 Fast Approximate Nearest Neighbor Search

We present several experiments we have conducted in order to analyse the performance of the two algorithms described in Section 3.

4.1.1 Data Dimensionality

Data dimensionality is one of the factors that has a great impact on the nearest neighbor matching performance. The top of Fig. 5 shows how the search performance degrades as the dimensionality increases in the case of random vectors. The data sets in this case each contain 10^5 vectors whose values are randomly sampled from the same uniform distribution. These random data sets are one of the most difficult problems for nearest neighbor search, as no value gives any predictive information about any other value.

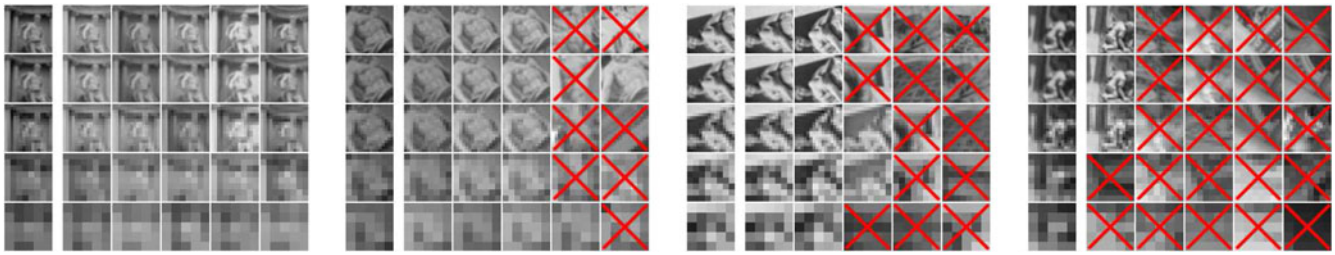


Fig. 6. Example of nearest neighbor queries with different patch sizes. The Trevi Fountain patch data set was queried using different patch sizes. The rows are arranged in decreasing order by patch size. The query patch is on the left of each panel, while the following five patches are the nearest neighbors from a set of 100,000 patches. Incorrect matches with respect to ground truth are shown with an X.

As can be seen in the top part of Fig. 5, the nearest-neighbor searches have a low efficiency for higher dimensional data (for 68 percent precision the approximate search speed is no better than linear search when the number of dimensions is greater than 800).

The performance is markedly different for many real-world data sets. The bottom part of Fig. 5 shows the speedup as a function of dimensionality for the Winder/Brown image patches[2] resampled to achieve varying dimensionality. In this case however, the speedup does not decrease with dimensionality, it's actually increasing for some precisions. This can be explained by the fact that there exists a strong correlation between the dimensions, so that even for 64×64 patches (4,096 dimensions), the similarity between only a few dimensions provides strong evidence for overall patch similarity.

Fig. 6 shows four examples of queries on the Trevi data set of patches for different patch sizes.

4.1.2 Search Precision

We use several data sets of different sizes for the experiments in Fig. 7. We construct 100K and 1 million SIFT feature data sets by randomly sampling a data set of over 5 million SIFT features extracted from a collection of CD cover images [24].[3] We also use the 31 million SIFT feature data set from the same source.

The desired search precision determines the degree of speedup that can be obtained with any approximate algorithm. Looking at Fig. 7 (the sift1M data set) we see that if we are willing to accept a precision as low as 60 percent, meaning that 40 percent of the neighbors returned are not the exact nearest neighbors, but just approximations, we can achieve a speedup of three orders of magnitude over linear search (using the multiple randomized kd-trees). However, if we require a precision greater than 90 percent the speedup is smaller, less than 2 orders of magnitude (using the priority search k-means tree).

We compare the two algorithms we found to be the best at finding fast approximate nearest neighbors (the multiple randomized kd-trees and the priority search k-means tree) with existing approaches, the ANN [11] and LSH algorithms [29][4] on the first data set of 100,000 SIFT features. Since the LSH implementation (the E^{2}LSH package) solves the R-near neighbor problem (finds the neighbors within a radius R of the query point, not the nearest neighbors), to find the nearest neighbors we have used the approach suggested in the E^{2}LSH's user manual: we compute the R-near neighbors for increasing values of R. The parameters for the LSH algorithm were chosen using the parameter estimation tool included in the E^{2}LSH package. For each case we have computed the precision achieved as the percentage of the query points for which the nearest neighbors were correctly found. Fig. 8 shows that the priority search k-means algorithm outperforms both the ANN and LSH algorithms by about an order of magnitude. The results for ANN are consistent with the experiment in Fig. 1, as ANN uses only a single kd-tree and does not benefit from the speedup due to using multiple randomized trees.

Fig. 9 compares the performance of nearest neighbor matching when the data set contains true matches for each feature in the test set to the case when it contains false matches. A true match is a match in which the query and the nearest neighbor point represent the same entity, for example, in case of SIFT features, they represent image patches of the same object. In this experiment we used two 100K SIFT features data sets, one that has ground truth determined from global image matching and one that is randomly sampled from a 5 million SIFT features data set and it contains only false matches for each feature in the test set. Our experiments showed that the randomized kd-trees have a significantly better performance for true matches, when the query features are

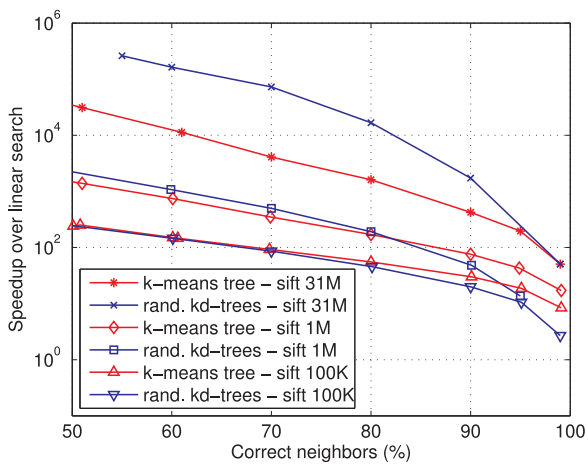


Fig. 7. Search speedup for different data set sizes.

2. http://phototour.cs.washington.edu/patches/default.htm.
3. http://www.vis.uky.edu/stewe/ukbench/data/.
4. We have used the publicly available implementations of ANN (http://www.cs.umd.edu/~mount/ANN/) and LSH (http://www.mit.edu/~andoni/LSH/).

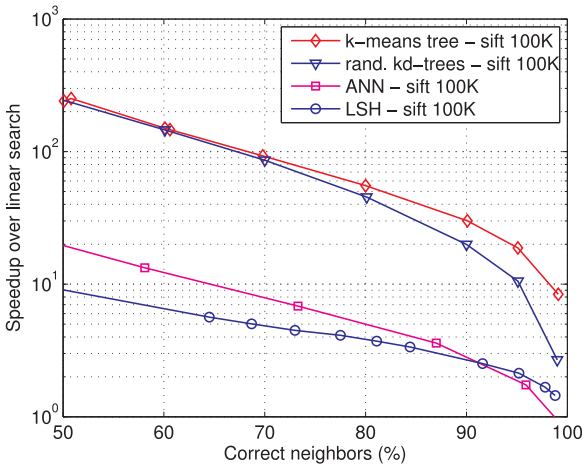


Fig. 8. Comparison of the search efficiency for several nearest neighbor algorithms.

likely to be significantly closer than other neighbors. Similar results were reported in [54].

Fig. 10 shows the difference in performance between the randomized kd-trees and the priority search k-means tree for one of the Winder/Brown patches data set. In this case, the randomized kd-trees algorithm clearly outperforms the priority search k-means algorithm everywhere except for precisions close to 100 percent. It appears that the kd-tree works much better in cases when the intrinsic dimensionality of the data is much lower than the actual dimensionality, presumably because it can better exploit the correlations among dimensions. However, Fig. 7 shows that the k-means tree can perform better for other data sets (especially for high precisions). This shows the importance of performing algorithm selection on each data set.

4.1.3 *Automatic Selection of Optimal Algorithm*

In Table 1, we show the results from running the parameter selection procedure described in Section 3.4 on a data set containing 100K randomly sampled SIFT features. We used two different search precisions (60 and 90 percent) and several combinations of the tradeoff factors w_b and w_m. For the build time weight, w_b, we used three different possible

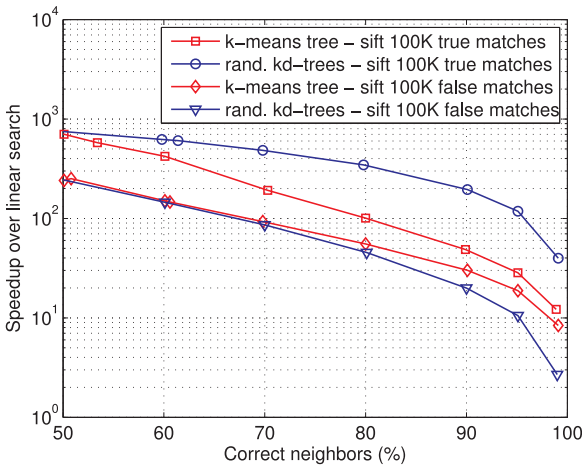


Fig. 9. Search speedup when the query points don't have "true" matches in the data set versus the case when they have.

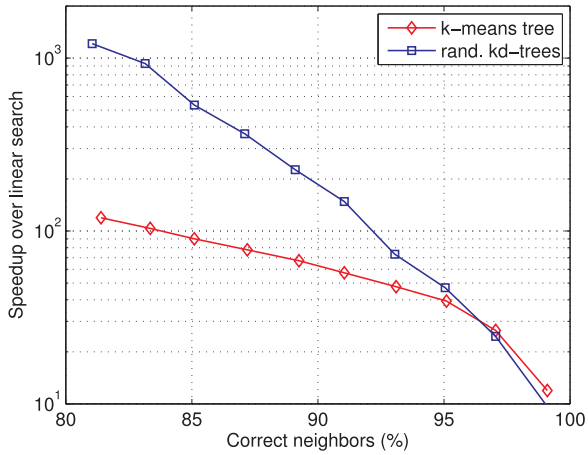


Fig. 10. Search speedup for the Trevi Fountain patches data set.

values: 0 representing the case where we don't care about the tree build time, 1 for the case where the tree build time and search time have the same importance and 0.01 representing the case where we care mainly about the search time but we also want to avoid a large build time. Similarly, the memory weight was chosen to be 0 for the case where the memory usage is not a concern, ∞ representing the case where the memory use is the dominant concern and 1 as a middle ground between the two cases.

4.2 Binary Features

This section evaluates the performance of the hierarchical clustering tree described in Section 3.3.

We use the Winder/Brown patches data set [53] to compare the nearest neighbor search performance of the hierarchical clustering tree to that of other well known nearest neighbor search algorithms. For the comparison we use a combination of both vector features such as SIFT, SURF, image patches and binary features such as BRIEF and ORB. The image patches have been downscaled to 16×16 pixels and are matched using normalized cross correlation. Fig. 11 shows the nearest neighbor search times for the different feature types. Each point on the graph is computed using the best performing algorithm for that particular feature type (randomized kd-trees or priority search k-means tree for SIFT, SURF, image patches and the hierarchical clustering algorithm for BRIEF and ORB). In each case the optimum choice of parameters that maximizes the speedup for a given precision is used.

In Fig. 12 we compare the hierarchical clustering tree with a multi-probe locality sensitive hashing implementation [30]. For the comparison we used data sets of BRIEF and ORB features extracted from the recognition benchmark images data set of [24], containing close to 5 million features. It can be seen that the hierarchical clustering index outperforms the LSH implementation for this data set. The LSH implementation also requires significantly more memory compared to the hierarchical clustering trees for when high precision is required, as it needs to allocate a large number of hash tables to achieve the high search precision. In the experiment of Fig. 12, the multi-probe LSH required six times more memory than the hierarchical search for search precisions above 90 percent.

TABLE 1
The Algorithms Chosen by Our Automatic Algorithm and Parameter Selection Procedure (sift100K Data Set)

$Pr.(\%)$	w_b	w_m	Algorithm Configuration	Dist. Error	Search Speedup	Memory Used	Build Time
60%	0	0	k-means, 16, 15	0.096	181.10	0.51	0.58
	0	1	k-means, 32, 10	0.058	180.9	0.37	0.56
	0.01	0	k-means, 16, 5	0.077	163.25	0.50	0.26
	0.01	1	kd-tree, 4	0.041	109.50	0.26	0.12
	1	0	kd-tree,1	0.044	56.87	0.07	0.03
	*	∞	kd-tree,1	0.044	56.87	0.07	0.03
90%	0	0	k-means, 128, 10	0.008	31.67	0.18	1.82
	0	1	k-means, 128, 15	0.007	30.53	0.18	2.32
	0.01	0	k-means, 32, 5	0.011	29.47	0.36	0.35
	1	0	k-means, 16, 1	0.016	21.59	0.48	0.10
	1	1	kd-tree,1	0.005	5.05	0.07	0.03
	*	∞	kd-tree,1	0.005	5.05	0.07	0.03

The "Algorithm Configuration" column shows the algorithm chosen and its optimum parameters (number of random trees in case of the kd-tree; branching factor and number of iterations for the k-means tree), the "Dist Error" column shows the mean distance error compared to the exact nearest neighbors, the "Search Speedup" shows the search speedup compared to linear search, the "Memory Used" shows the memory used by the tree(s) as a fraction of the memory used by the data set and the "Build Time" column shows the tree build time as a fraction of the linear search time for the test set.

5 Scaling Nearest Neighbor Search

Many papers have shown that using simple non-parametric methods in conjunction with large scale data sets can lead to very good recognition performance [4], [7], [55], [56]. Scaling to such large data sets is a difficult task, one of the main challenges being the impossibility of loading the data into the main memory of a single machine. For example, the size of the raw tiny images data set of [7] is about 240 GB, which is greater than what can be found on most computers at present. Fitting the data in memory is even more problematic for data sets of the size of those used in [4], [8], [55].

When dealing with such large amounts of data, possible solutions include performing some dimensionality reduction on the data, keeping the data on the disk and loading only parts of it in the main memory or distributing the data on several computers and using a distributed nearest neighbor search algorithm.

Dimensionality reduction has been used in the literature with good results ([7], [27], [28], [32], [57]), however even with dimensionality reduction it can be challenging to fit the data in the memory of a single machine for very large data sets. Storing the data on the disk involves significant performance penalties due to the performance gap between memory and disk access times. In FLANN we used the approach of performing distributed nearest neighbor search across multiple machines.

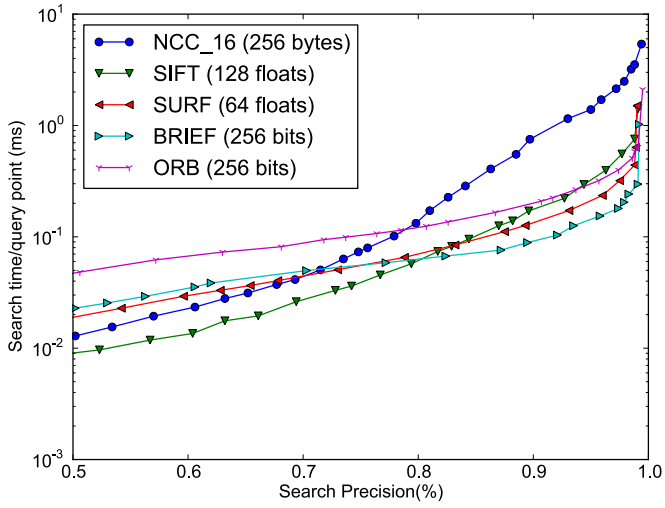


Fig. 11. Absolute search time for different popular feature types (both binary and vector).

5.1 Searching on a Compute Cluster

In order to scale to very large data sets, we use the approach of distributing the data to multiple machines in a compute cluster and perform the nearest neighbor search using all the machines in parallel. The data is distributed equally between the machines, such that for a cluster of N machines each of them will only have to index and search $1/N$ of the whole data set (although the ratios can be changed to have more data on some machines than others). The final result of the nearest neighbor search is obtained by merging the partial results from all the machines in the cluster once they have completed the search.

In order to distribute the nearest neighbor matching on a compute cluster we implemented a Map-Reduce like algorithm using the message passing interface (MPI) specification.

Algorithm 4 describes the procedure for building a distributed nearest neighbor matching index. Each process in the cluster executes in parallel and reads from a distributed filesystem a fraction of the data set. All processes build the nearest neighbor search index in parallel using their respective data set fractions.

Algorithm 4 Constructing a distributed index on a compute cluster

Input: dataset D, index parameters P

procedure BUILDINDEX(D,P)

1: $i \leftarrow$ MPI_rank()
2: $D_i \leftarrow$ read portion of the dataset D corresponding to rank i using the distributed filesystem
3: build index in parallel on each process with dataset D_i and parameters P
4: MPI_barrier() // synchronize all processes

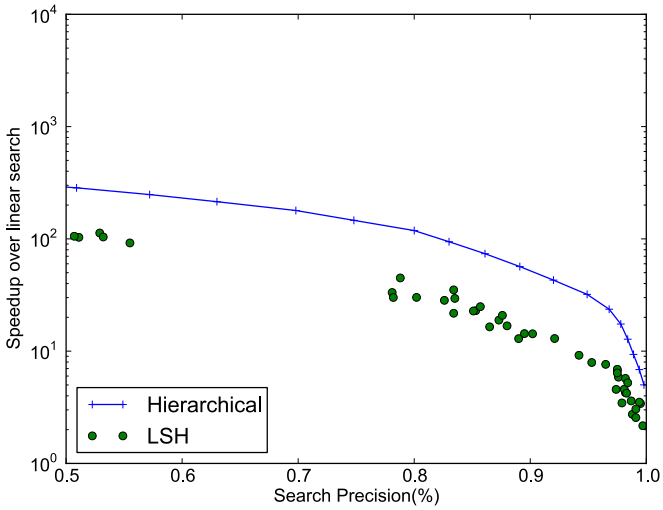


Fig. 12. Comparison between the hierarchical clustering index and LSH for the Nister/Stewenius recognition benchmark images data set of about 5 million features.

In order to search the distributed index the query is sent from a client to one of the computers in the MPI cluster, which we call the master server (see Fig. 13). By convention the master server is the process with rank 0 in the MPI cluster, however any process in the MPI cluster can play the role of master server.

The master server broadcasts the query to all of the processes in the cluster and then each process can run the nearest neighbor matching in parallel on its own fraction of the data. When the search is complete an MPI reduce operation is used to merge the results back to the master process and the final result is returned to the client.

The master server is not a bottleneck when merging the results. The MPI reduce operation is also distributed, as the partial results are merged two by two in a hierarchical fashion from the servers in the cluster to the master server. Additionally, the merge operation is very efficient, since the distances between the query and the neighbors don't have to be re-computed as they are returned by the nearest neighbor search operations on each server.

Algorithm 5 Searching a distributed index on a compute cluster

Input: query Q, search parameters P

procedure SEARCHINDEX(Q,P)

1: MPI_broadcast(Q,P) // broadcast query and parameters to all processes
2: $NN_i \leftarrow$ run nearest neighbor search with query Q and parameters P on each process i
3: NN $\leftarrow$ MPI_reduce(NN_i) // merge results using a MPI reduce operation
4: **return** NN

When distributing a large data set for the purpose of nearest neighbor search we chose to partition the data into multiple disjoint subsets and construct independent indexes for each of those subsets. During search the query is broadcast to all the indexes and each of them performs the nearest neighbor search within its associated data. In a different approach, Aly et al. [58] introduce a distributed k-d tree

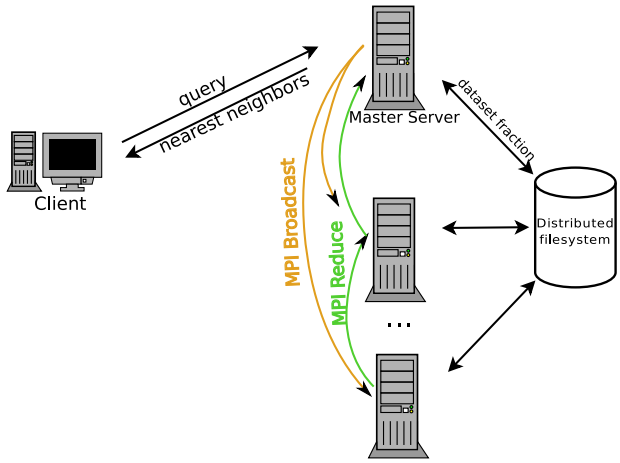


Fig. 13. Scaling nearest neighbor search on a compute cluster using message passing interface standard.

implementation where they place a root k-d tree on top of all the other trees (leaf trees) with the role of selecting a subset of trees to be searched and only send the query to those trees. They show the distributed k-d tree has higher throughput compared to using independent trees, due to the fact that only a portion of the trees need to be searched by each query.

The partitioning of the data set into independent subsets, as described above and implemented in FLANN, has the advantage that it doesn't depend on the type of index used (randomized kd-trees, priority search k-means tree, hierarchical clustering, LSH) and can be applied to any current or future nearest neighbor algorithm in FLANN. In the distributed k-d tree implementation of [58] the search does not backtrack in the root node, so it is possible that subsets of the data containing near points are not searched at all if the root k-d tree doesn't select the corresponding leaf k-d trees at the beginning.

5.2 Evaluation of Distributed Search

In this section we present several experiments that demonstrate the effectiveness of the distributed nearest neighbor matching framework in FLANN. For these experiments we have used the 80 million patch data set of [7].

In an MPI distributed system it's possible to run multiple parallel processes on the same machine, the recommended approach is to run as many processes as CPU cores on the machine. Fig. 14 presents the results of an experiment in which we run multiple MPI processes on a single machine with eight CPU cores. It can be seen that the overall performance improves when increasing the number of processes from 1 to 4, however there is a decrease in performance when moving from four to eight parallel processes. This can be explained by the fact that increasing the parallelism on the same machine also increases the number of requests to the main memory (since all processes share the same main memory), and at some point the bottleneck moves from the CPU to the memory. Increasing the parallelism past this point results in decreased performance. Fig. 14 also shows the direct search performance obtained by using FLANN directly without the MPI layer. As expected, the direct search performance is identical to the performance obtained when using the MPI layer with a

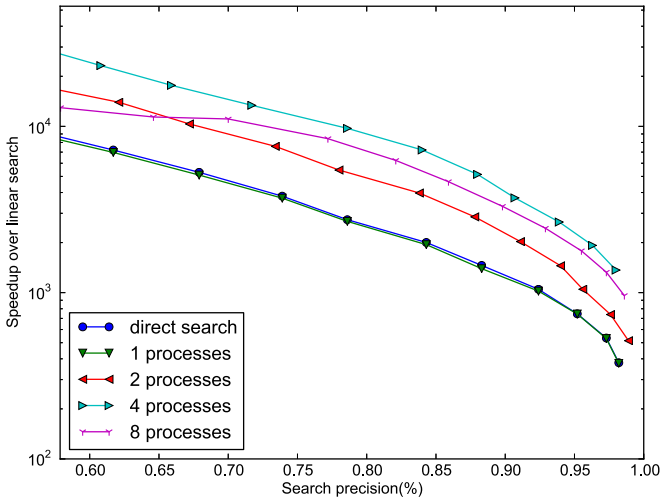


Fig. 14. Distributing nearest neighbor search on a single multi-core machine. When the degree of parallelism increases beyond a certain point the memory access becomes a bottleneck. The "direct search" case corresponds to using the FLANN library directly, without the MPI layer.

single process, showing no significant overhead from the MPI runtime. For this experiment and the one in Fig. 15 we used a subset of only 8 million tiny images to be able to run the experiment on a single machine.

Fig. 15 shows the performance obtained by using eight parallel processes on one, two or three machines. Even though the same number of parallel processes are used, it can be seen that the performance increases when those processes are distributed on more machines. This can also be explained by the memory access overhead, since when more machines are used, fewer processes are running on each machine, requiring fewer memory accesses.

Fig. 16 shows the search speedup for the data set of 80 million tiny images of [7]. The algorithm used is the

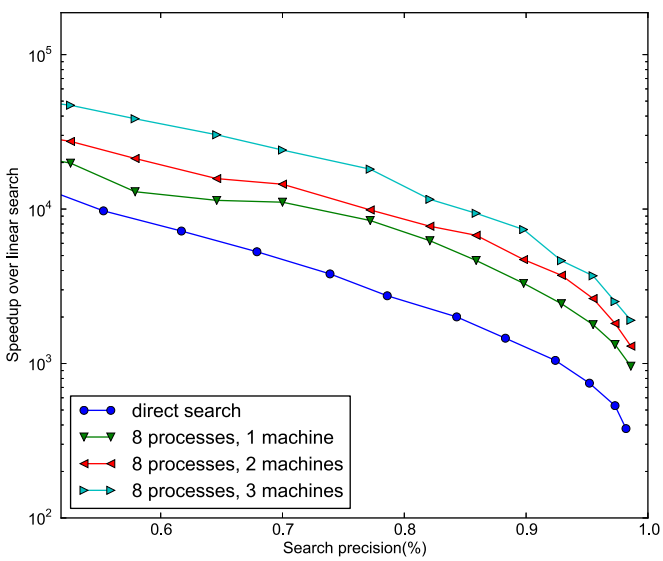


Fig. 15. The advantage of distributing the search to multiple machines. Even when using the same number of parallel processes, distributing the computation to multiple machines still leads to an improvement in performance due to less memory access overhead. "Direct search" corresponds to using FLANN without the MPI layer and is provided as a comparison baseline.

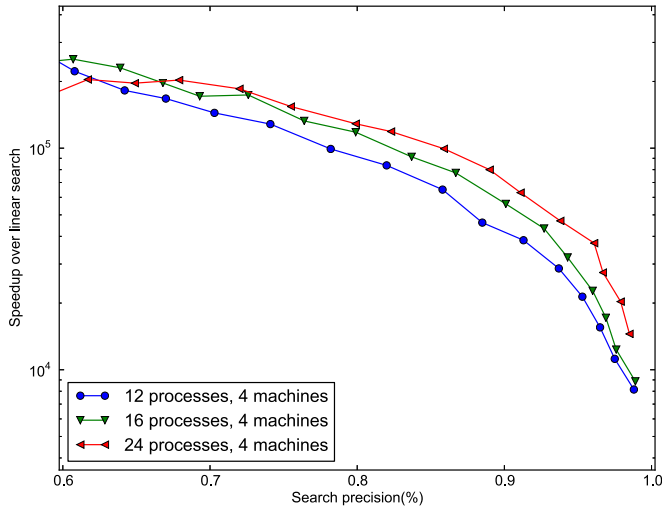


Fig. 16. Matching 80 million tiny images directly using a compute cluster.

radomized k-d tree forest as it was determined by the autotuning procedure to be the most efficient in this case. It can be seen that the search performance scales well with the data set size and it benefits from using multiple parallel processes.

All the previous experiments have shown that distributing the nearest neighbor search to multiple machines results in an overall increase in performance in addition to the advantage of being able to use more memory. Ideally, when distributing the search to N machines the speedup would be N times higher, however in practice for approximate nearest neighbor search the speedup is smaller due to the fact that the search on each of the machines has sub-linear complexity in the size of the input data set.

6 The FLANN Library

The work presented in this paper has been made publicly available as an open source library named Fast Library for Approximate Nearest Neighbors[5] [59].

FLANN is used in a large number of both research and industry projects (e.g., [60], [61], [62], [63], [64]) and is widely used in the computer vision community, in part due to its inclusion in OpenCV [65], the popular open source computer vision library. FLANN also is used by other well known open source projects, such as the point cloud library (PCL) and the robot operating system (ROS) [63]. FLANN has been packaged by most of the mainstream Linux distributions such as Debian, Ubuntu, Fedora, Arch, Gentoo and their derivatives.

7 Conclusions

This paper addresses the problem of fast nearest neighbor search in high dimensional spaces, a core problem in many computer vision and machine learning algorithms and which is often the most computationally expensive part of these algorithms. We present and compare the algorithms we have found to work best at fast approximate search in high dimensional spaces: the randomized k-d trees and a newly introduced algorithm, the priority search k-means tree. We introduce a new algorithm for fast approximate

5. http://www.cs.ubc.ca/research/flann.

matching of binary features. We address the issues arising when scaling to very large size data sets by proposing an algorithm for distributed nearest neighbor matching on compute clusters.

REFERENCES

[1] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, 2004.

[2] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Object retrieval with large vocabularies and fast spatial matching," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2007, pp. 1–8.

[3] J. Sivic and A. Zisserman, "Video Google: A text retrieval approach to object matching in videos," in *Proc. IEEE 9th Int. Conf. Comput. Vis.*, 2003, pp. 1470–1477.

[4] J. Hays and A. A. Efros, "Scene completion using millions of photographs," *ACM Trans. Graph.*, vol. 26, p. 4, 2007.

[5] G. Shakhnarovich, P. Viola, and T. Darrell, "Fast pose estimation with parameter-sensitive hashing," in *Proc. IEEE 9th Int. Conf. Comput. Vis.*, 2003, pp. 750–757.

[6] A. C. Berg, T. L. Berg, and J. Malik, "Shape matching and object recognition using low distortion correspondences," in *Proc. IEEE CS Conf. Comput. Vis. Pattern Recog.*, 2005, vol. 1, pp. 26–33.

[7] A. Torralba, R. Fergus, and W.T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, Nov. 2008.

[8] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2009, pp. 248–255.

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[10] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, 1977.

[11] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *J. ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[12] J. S. Beis and D. G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 1997, pp. 1000–1006.

[13] C. Silpa-Anan and R. Hartley, "Optimised KD-trees for fast image descriptor matching," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2008, pp. 1–8.

[14] M. Muja and D.G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *Proc. Int. Conf. Computer Vis. Theory Appl.*, 2009, pp. 331–340.

[15] R. F. Sproull, "Refinements to nearest-neighbor searching in k-dimensional trees," *Algorithmica*, vol. 6, no. 1, pp. 579–589, 1991.

[16] S. Dasgupta and Y. Freund, "Random projection trees and low dimensional manifolds," in *Proc. 40th Annu. ACM Symp. Theory Comput.*, 2008, pp. 537–546.

[17] Y. Jia, J. Wang, G. Zeng, H. Zha, and X. S. Hua, "Optimizing kd-trees for scalable visual descriptor indexing," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2010, pp. 3392–3399.

[18] K. Fukunaga and P. M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Trans. Comput.*, vol. C-24, no. 7, pp. 750–753, Jul. 1975.

[19] S. Brin, "Near neighbor search in large metric spaces," in *Proc. 21th Int. Conf. Very Large Data Bases*, 1995, pp. 574–584.

[20] A. W. Moore, "The anchors hierarchy: Using the triangle inequality to survive high dimensional data," in *Proc. 16th Conf. Uncertainity Artif. Intell.*, 2000, pp. 397–405.

[21] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proc. ACM-SIAM Symp. Discrete Algorithms*, 1993, pp. 311–321.

[22] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proc. 23rd Int. Conf. Mach. Learning*, 2006, pp. 97–104.

[23] T. Liu, A. Moore, A. Gray, K. Yang," An investigation of practical approximate nearest neighbor algorithms," presented at the Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 2004.

[24] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2006, pp. 2161–2168.

[25] B. Leibe, K. Mikolajczyk, and B. Schiele, "Efficient clustering and matching for object class recognition," in *Proc. British Mach. Vis. Conf.*, 2006, pp. 789–798.

[26] G. Schindler, M. Brown, and R. Szeliski, "City-Scale location recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2007, pp. 1–7.

[27] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 1, pp. 1–15, Jan. 2010.

[28] A. Babenko and V. Lempitsky, "The inverted multi-index," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2012, pp. 3069–3076.

[29] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[30] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: Efficient indexing for high-dimensional similarity search," in *Proc. Int. Conf. Very Large Data Bases*, 2007, pp. 950–961.

[31] M. Bawa, T. Condie, and P. Ganesan, "LSH forest: Self-tuning indexes for similarity search," in *Proc. 14th Int. Conf. World Wide Web*, 2005, pp. 651–660.

[32] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Proc. Adv. Neural Inf. Process. Syst.*, 2008, p. 6.

[33] P. Jain, B. Kulis, and K. Grauman, "Fast image search for learned metrics," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2008, pp. 1–8.

[34] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *Proc. IEEE 12th Int. Conf. Comput. Vis.*, 2009, pp. 2130–2137.

[35] B. Kulis and T. Darrell, "Learning to hash with binary reconstructive embeddings," in *Proc. 23rd Adv. Neural Inf. Process. Syst.*, 2009, vol. 22, pp. 1042–1050.

[36] M. Raginsky and S. Lazebnik, "Locality-sensitive binary codes from shift-invariant kernels," in *Proc. Adv. Neural Inf. Process. Syst.*, 2009, vol. 22, pp. 1509–1517.

[37] J. Wang, S. Kumar, and S. F. Chang, "Semi-supervised hashing for scalable image retrieval," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2010, pp. 3424–3431.

[38] J. He, W. Liu, and S. F. Chang, "Scalable similarity search with optimized kernel hashing," in *Proc. Int. Conf. Knowledge Discovery Data Mining*, 2010, pp. 1129–1138.

[39] H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu, "Complementary hashing for approximate nearest neighbor search," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2011, pp. 1631–1638.

[40] T. B. Sebastian and B. B. Kimia, "Metric-based shape retrieval in large databases," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2002, vol. 3, pp. 291–296.

[41] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, 2011, pp. 1312–1317.

[42] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li, "Scalable k-NN graph construction for visual descriptors," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2012, pp. 1106–1113.

[43] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Comput. J.*, vol. 7, no. 4, pp. 308–313, 1965.

[44] F. Hutter, "Automated configuration of algorithms for solving hard computational problems," Ph.D. dissertation, *Comput. Sci. Dept.*, Univ. British Columbia, Vancouver, BC, Canada, 2009.

[45] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "ParamILS: An automatic algorithm configuration framework," *J. Artif. Intell. Res.*, vol. 36, pp. 267–306, 2009.

[46] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.

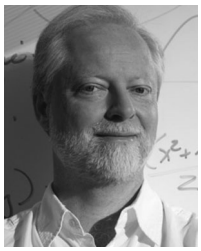
[47] M. Muja, "Scalable nearest neighbour methods for high dimensional data," Ph.D. dissertation, *Comput. Sci. Dept.*, Univ. British Columbia, Vancouver, BC, Canada, 2013.

[48] D. Arthur and S. Vassilvitskii, "K-Means++: The advantages of careful seeding," in *Proc. Symp. Discrete Algorithms*, 2007, pp. 1027–1035.

[49] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary robust independent elementary features," in *Proc. 11th Eur. Conf. Comput. Vis.*, 2010, pp. 778–792.

[50] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *Proc. IEEE Int. Conf. Comput. Vis.*, Barcelona, Spain, 2011, pp. 2564–2571.
[51] S. Leutenegger, M. Chli, and R. Siegwart, "BRISK: Binary robust invariant scalable keypoints," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2011, pp. 2548–2555.
[52] M. Muja and D. G. Lowe, "Fast matching of binary features," in *Proc. 9th Conf. Comput. Robot Vis.*, 2012, pp. 404–410.
[53] S. Winder and M. Brown, "Learning local image descriptors," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2007, pp. 1–8.
[54] K. Mikolajczyk and J. Matas, "Improving descriptors for fast tree matching by optimal linear projection," in *Proc. IEEE 11th Int. Conf. Comput. Vis.*, 2007, pp. 1–8.
[55] J. Hays and A. A. Efros, "IM2GPS: Estimating geographic information from a single image," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2008, pp. 1–8.
[56] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *IEEE Intell. Syst.*, vol. 24, no. 2, pp. 8–12, Mar./Apr. 2009.
[57] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2008, pp. 1–8.
[58] M. Aly, M. Munich, and P. Perona, "Distributed Kd-trees for retrieval from very large image collections," presented at the *British Mach. Vis. Conf.*, DuDundee, U.K., 2011.
[59] M. Muja and D. G. Lowe, "FLANN: Fast library for approximate nearest neighbors," [Online]. Available: http://www.cs.ubc.ca/research/flann
[60] M. Cummins and P. Newman, "Highly scalable appearance-only SLAM-FAB-MAP 2.0," presented at the Robotics: Science and Systems Conf., vol. 5, Seattle, Washington, USA, 2009.
[61] M. Havlena, A. Torii, M. Jancosek, and T. Pajdla, "Automatic reconstruction of Mars artifacts," in *Proc. Eur. Planet. Sci. Congress*, 2009, p. 280.
[62] M. Havlena, A. Torii, J. Knopp, and T. Pajdla, "Randomized structure from motion based on atomic 3D models from camera triplets," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2009, pp. 2874–2881.
[63] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A.Y. Ng, "ROS: An open-source robot operating system," in *Proc. ICRA Open-Source Softw. Workshop*, 2009.
[64] P. Turcot and D.G. Lowe, "Better matching with fewer features: The selection of useful features in large database recognition problems," in *Proc. Comput. Vis. Workshops*, 2009, pp. 2109–2116.
[65] G. Bradski and A. Kaehler, *Learning OpenCV: Comput. Vision with the OpenCV Library*. Sebastopol, CA, USA: O'Reilly Media, 2008.

Marius Muja received the BSc degree in computer science from "Politehnica" University of Timisoara in 2005, and the PhD degree in computer science from University of British Columbia in 2013. He is currently the CTO and cofounder of BitLit Media Inc., a company that develops recognition algorithms for allowing users to prove the ownership of paper books in order to obtain access to the electronic versions. His interests include scalable nearest neighbor matching techniques, large-scale image retrieval, and object recognition. He is a member of the IEEE.



David G. Lowe received the BSc degree in computer science from the University of British Columbia in 1978, and the PhD degree in computer science from Stanford University in 1984. He is currently a professor of computer science at the University of British Columbia and a fellow of the Canadian Institute for Advanced Research. He is also the chairman and the co-founder of Cloudburst Research Inc., a company that develops computer vision apps for mobile devices. His research interests include object recognition, local image features, and computational models for human vision. He is a member of the IEEE.