Fast Matching of Binary Features

Marius Muja and David G. Lowe
Laboratory for Computational Intelligence
University of British Columbia, Vancouver, Canada
{*mariusm,lowe*}@*cs.ubc.ca*

***Abstract*—There has been growing interest in the use of binary-valued features, such as BRIEF, ORB, and BRISK for efficient local feature matching. These binary features have several advantages over vector-based features as they can be faster to compute, more compact to store, and more efficient to compare. Although it is fast to compute the Hamming distance between pairs of binary features, particularly on modern architectures, it can still be too slow to use linear search in the case of large datasets. For vector-based features, such as SIFT and SURF, the solution has been to use approximate nearest-neighbor search, but these existing algorithms are not suitable for binary features. In this paper we introduce a new algorithm for approximate matching of binary features, based on priority search of multiple hierarchical clustering trees. We compare this to existing alternatives, and show that it performs well for large datasets, both in terms of speed and memory efficiency.**

***Keywords*-nearest neighbors; feature matching; binary features**

I. Introduction

A number of binary visual descriptors have been recently proposed in the literature, including BRIEF [1], ORB [2], and BRISK [3]. These have several advantages over the more established vector-based descriptors such as SIFT [4] and SURF [5], as they are cheaper to compute, more compact to store, and faster to compare with each other.

Binary features are compared using the Hamming distance, which for binary data can be computed by performing a bitwise XOR operation followed by a bit count on the result. This involves only bit manipulation operations which can be performed quickly, especially on modern computers where there is hardware support for counting the number of bits that are set in a word (the POPCNT instruction[1]).

Even though computing the distance between pairs of binary features can be done efficiently, using linear search for matching can be practical only for smaller datasets. For large datasets, linear matching becomes a bottleneck in most applications. The typical solution in such situations is to replace the linear search with an approximate matching algorithm that can offer speedups of several orders of magnitude over linear search, at the cost that a fraction of the nearest neighbors returned are approximate neighbors (but usually close in distance to the exact neighbors).

[1]We are referring to the x86_64 architecture, other architectures have similar instructions for counting the number of bits set.

Many algorithms typically used for approximate matching in case of vector features, which perform a hierarchical decomposition of the search space, are not readily suitable for matching binary features because they assume the features exist in a vector space where each dimension of the features can be continuously averaged. Examples of such algorithms are the kd-tree algorithm, the vocabulary tree and the hierarchical k-means tree.

For matching binary features, the approximate nearest neighbor search algorithms used in the literature are mostly based on various hashing techniques such as locality sensitive hashing [2], semantic hashing [6] or min-hash [7].

In this paper we introduce a new algorithm for matching binary features, based on hierarchical decomposition of the search space. We have implemented this algorithm on top of the publicly available FLANN open source library [8]. We compare the performance of this algorithm to other well know approximate nearest neighbor algorithms implemented by the FLANN library for both binary and vector-based features. We show that when compared with an LSH implementation, which is what is typically used in the literature for matching binary features, our algorithm performs comparatively or better and scales favorably for larger datasets.

II. Related work

Many algorithms for fast matching typically used for vector-based descriptors are not suitable for binary descriptors so the methods proposed for matching them have been either linear search or hashing techniques.

There is also a lot of work published on the topic of learning short binary codes through Hamming embeddings from different feature spaces. Salakhutdinov and Hinton [6] introduce the notion of semantic hashing when they learn a deep graphical model that maps documents to small binary codes. When the mapping is performed such that close features are mapped to close codes (in Hamming space), the nearest neighbor matching can be efficiently performed by searching for codes that differ by a few bits from the query code. A similar approach is used by Torralba et al. [9] who learn compact binary codes from images with the goal of performing real-time image recognition on a large dataset of images using limited memory. Weiss et al. [10] formalize the requirements for good codes and introduce a new technique for efficiently computing binary codes. Other works in which a Hamming embedding is computed from SIFT visual features are those of Jegou at al. [11] and Strecha at al. [12].

Performing approximate nearest neighbor search by examining all the points in a Hamming radius works efficiently when the distance between the matching codes is small. When this distance gets larger the number of points in the Hamming radius gets exponentially larger, making the method unpractical. This is the case for visual binary features such as BRIEF or ORB where the minimum distance between matching features can be larger than 20 bits. In cases such as this other nearest neighbor matching techniques must be used. Locality Sensitive Hashing (LSH) [13] is one of the nearest neighbor search techniques that has been shown to be effective for fast matching of binary features[2]. Zitnick [7] uses the min-hash technique for the same purpose.

Brin [14] proposes a nearest neighbor search data structure called GNAT (Geometric Near-Neighbor Access Tree) which does a hierarchical decomposition of the search space to accelerate the search and works for any metric distance. We have used the GNAT tree and the results in that paper to guide us in the design of the algorithm presented below.

III. Matching binary features

In this section we introduce a new data structure and algorithm which we have found to be very effective at matching binary features, including the recently introduced BRIEF [1] and ORB [2]. The algorithm performs a hierarchical decomposition of the search space by successively clustering the input dataset and constructing a tree in which every non-leaf node contains a cluster center and the leaf nodes contain the input points that are to be matched.

A. Building the tree

The tree building process (presented in Algorithm 1) starts with all the points in the dataset and divides them into K clusters, where K is a parameter of the algorithm, called the branching factor. The clusters are formed by first selecting K points at random as the cluster centers followed by assigning to each center the points closer to that center than to any of the other centers. The algorithm is repeated recursively for each of the resulting clusters until the number of points in each cluster is below a certain threshold (the maximum leaf size), in which case that node becomes a leaf node.

The decomposition described above is similar to that of k-medoids clustering (k-medoids is an adaptation of the k-means algorithm in which each cluster center is chosen as one of the input data points instead of being the mean of the cluster elements). However, we are not trying to minimize the squared error (the distance between the cluster centers and their elements) as in the case of k-medoids. Instead we are simply selecting the cluster centers randomly from the input points, which results in a much simpler and more efficient algorithm for building the tree and gives improved independence when using multiple trees as described below. We have experimented with minimizing the squared error and with choosing the cluster centers using the greedy approach used in the GNAT tree [14] and neither brought improvements for the nearest neighbor search performance.

Algorithm 1 Building one hierarchical clustering tree

```
Input: features dataset $D$
Output: hierarchical clustering tree
Parameters: branching factor $K$, maximum leaf size $S_L$
 1: if size of $D < S_L$ then
 2:     create leaf node with the points in $D$
 3: else
 4:     $P \leftarrow$ select $K$ points at random from $D$
 5:     $C \leftarrow$ cluster the points in $D$ around nearest centers $P$
 6:     for each cluster $C_i \in C$ do
 7:         create non-leaf node with center $P_i$
 8:         recursively apply the algorithm to the points in $C_i$
 9:     end for
10: end if
```

We found that building multiple trees and using them in parallel during the search results in significant improvements for the search performance. The approach of using multiple randomized trees is known to work well for randomized kd-trees [15], however the authors of [16] reported that it was not effective for hierarchical k-means trees. The fact that using multiple randomized trees is effective for the algorithm we propose is likely due to the fact that we are choosing the cluster centers randomly and perform no further iterations to obtain a better clustering (as in the case of the hierarchical k-means tree). When using hierarchical space decompositions for nearest neighbor search, the hard cases occur when the closest neighbor to the query point lies just across a boundary from the domain explored, and the search needs to backtrack to reach the correct domain. The benefit of using multiple randomized trees comes from the fact that they are different enough such that, when exploring them in parallel, the closest neighbor probably lies in different domains in different trees, increasing the likelihood of the search reaching a correct domain quickly.

B. Searching for nearest neighbors using parallel hierarchical clustering trees

The process of searching multiple hierarchical clustering trees in parallel is presented in Algorithm 2. The search starts with a single traverse of each of the trees, during which the algorithm always picks the node closest to the query point and recursively explores it, while adding the unexplored nodes to a priority queue. When reaching the leaf node all the points contained within are linearly searched.

After each of the trees has been explored once, the search is continued by extracting from the priority queue the closest node to the query point and resuming the tree traversal from there.

Algorithm 2 Searching parallel hierarchical clustering trees

```
Input: hierarchical clustering trees $T_i$, query point $Q$
Output: $K$ nearest approximate neighbors of query point
Parameters: max number of points to examine $L_{max}$
1: $L \leftarrow 0$ {$L$ = number of points searched}
2: $PQ \leftarrow$ empty priority queue
3: $R \leftarrow$ empty priority queue
4: for each tree $T_i$ do
5:     call TraverseTree($T_i$,$PQ$,$R$)
6: end for
7: while $PQ$ not empty and $L < L_{max}$ do
8:     $N \leftarrow$ top of $PQ$
9:     call TraverseTree($N$,$PQ$,$R$)
10: end while
11: return $K$ top points from $R$
procedure TraverseTree($N$,$PQ$,$R$)
1: if node $N$ is a leaf node then
2:     search all the points in $N$ and add them to $R$
3:     $L \leftarrow L + |N|$
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 search ends when the number of points examined exceeds a maximum limit given as a parameter to the search algorithm. This limit specifies the degree of approximation desired from the algorithm. The higher the limit the more exact neighbors are found, but the search is more expensive. In practice it is often useful to express the degree of approximation as search precision, the percentage of exact neighbors found in the total neighbors returned. The relation between the search precision and the maximum point limit parameter can be experimentally determined for each dataset using a cross-validation approach.

IV. Evaluation

We use the Winder/Brown patch dataset [17] to evaluate the performance of the algorithm described in section III. These patches were obtained from consumer images of known landmarks. We first investigate how the different parameters influence the performance of the algorithm and then compare it to other approximate nearest neighbor algorithms described in [16] and implemented by the FLANN library [8]. We also implemented the hierarchical clustering algorithm on top of the open source FLANN library in order to take advantage of the parameter auto-tuning framework provided by the library. To show the scalability of the algorithm we use the 80 million tiny images dataset of [18].

A. Algorithm parameters

The behavior of the algorithm presented in section III is influenced by three parameters: the number of parallel trees built, the branching factor and the maximum size of the leaf nodes for each tree. In this section we evaluate the impact of each of these parameters on the performance of the algorithm by analyzing the speedup over linear search with respect to the search precision for different values of the parameters. Speedup over linear search is used both because it is an intuitive measure of the algorithm's performance and because it's relatively independent of the hardware on which the algorithm is run.

For the experiments in this section we constructed a dataset of approximately 310,000 BRIEF features extracted from all the patch datasets of [17] combined (the Trevi, Halfdome and Notredame datasets).

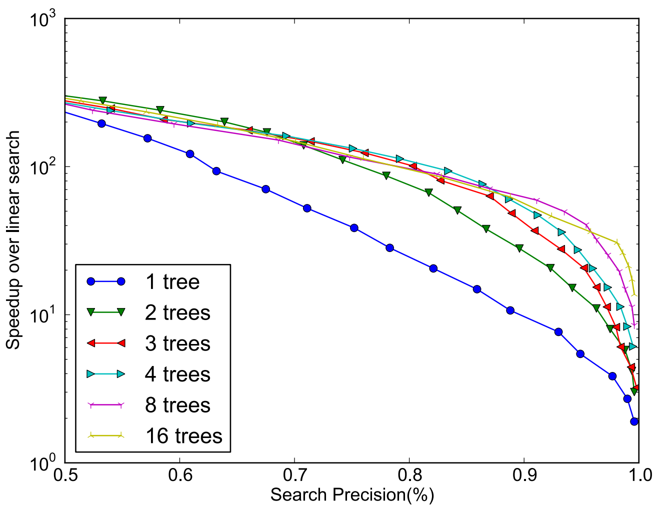


Figure 2. Speedup over linear search for different number of parallel randomized trees used by the index

Figure 2 shows how the algorithm's performance depends on the number of parallel trees used. It can be seen that there's a significant benefit in using more than a single tree and that the performance generally increases as more trees are used. The optimum number of trees depends on the desired precision, for example, for search precisions below 70% there is no benefit in using more than two parallel trees, however for precisions above 85% a higher number (between 4 and 8 trees) gives better results. Obviously more trees implies more memory and a longer tree build time, so in practice the optimum number of trees depends on multiple factors such as the desired search precision, the available memory and constraints on the tree build time.

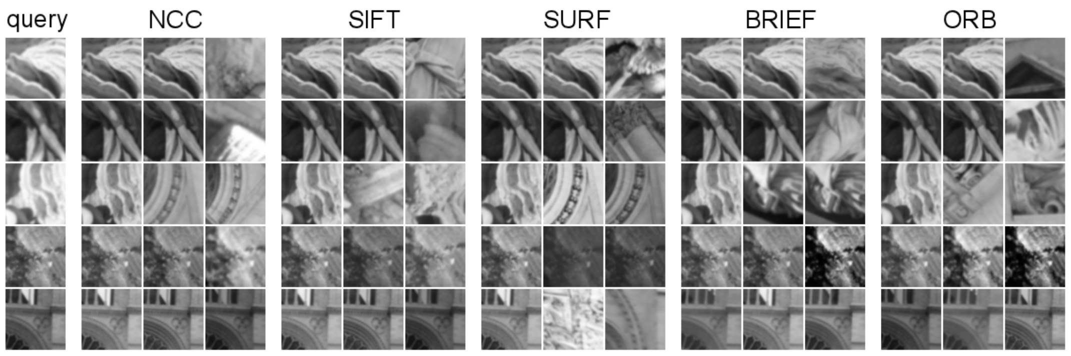


Figure 1. Random sample of query patches and the first three nearest neighbors returned when using different feature types

The branching factor also has an impact on the search performance, as figure 3 shows. Higher branching factors perform better for high precisions (above 80%), but there is little gain for branching factors above 16 or 32. Also very large branching factors perform worse for lower precisions and have a higher tree build time.

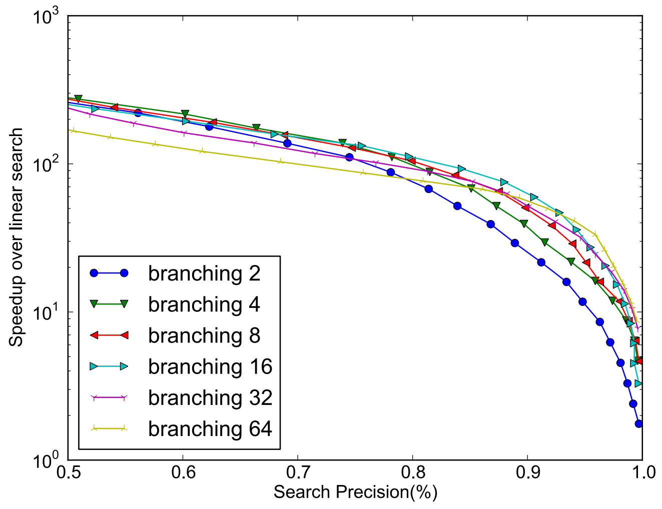


Figure 3. Speedup over linear search for different branching factors

The last parameter we examine is the maximum leaf size. From figure 4 we can see that a maximum leaf size of 150 performs better than a small leaf size (16, which is equal to the branching factor) or a large leaf size (500). This can be explained by the fact that computing the distance between binary features is an efficient operation, and for small leaf sizes the overhead of traversing the tree to examine more leaves is greater than the cost of comparing the query feature to all the features in a larger leaf. If the leaves are very large however, the cost of linearly examining all the features in the leaves ends up being greater than the cost of traversing the tree.

The above experiments show that the best parameters to choose depend on several factors and are often a trade-off between memory usage/tree build time and search performance. A similar cross-validation approach to the one proposed in [16] can be used for choosing the optimum parameters for a particular application.

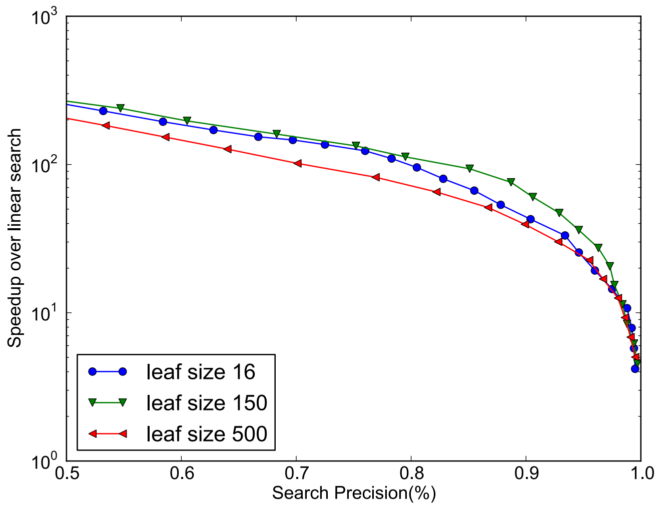


Figure 4. Speedup over linear search for different number leaf node sizes (branching factor = 16)

B. Comparison to other approximate nearest neighbor algorithms

We compare the nearest neighbor search performance of the hierarchical clustering algorithm introduced in section III to that of the nearest neighbor search algorithms implemented by the publicly available FLANN library. For comparison we use a combination of different features types, both vector features such as SIFT, SURF and NCC (normalized cross correlation) and binary features such as BRIEF and ORB. In the case of normalized cross correlation we downscaled the patches to 16x16 pixels. Figure 1 shows a random sample of 5 query patches and the first three neighbor patches returned when using each of the descriptors mentioned above.

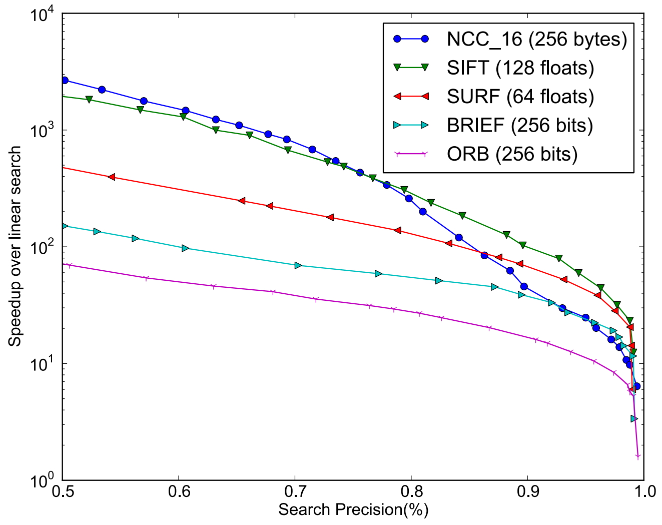


Figure 5. Speedup over linear for different popular features types (both binary and vector)

Figure 5 shows the nearest neighbor search speedup for different feature types. Each point on the graph is taken as the best performing algorithm for that particular feature type (randomized kd-trees or hierarchical k-means for SIFT, SURF, NCC and the hierarchical clustering algorithm introduced in section III for BRIEF and ORB). In each case the optimum choice of parameters that maximizes the speedup for a given precision is used.

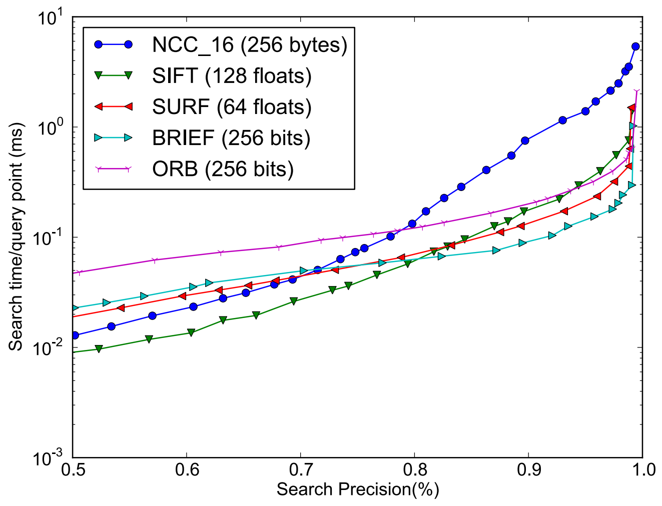


Figure 6. Absolute search time for different popular features types (both binary and vector)

Examining figure 5 it can be seen that the hierarchical clustering tree gives significant speedups over linear search ranging between one and two orders of magnitude for search precisions in the range 50-99%. This is less impressive than the search speedups obtained for the vector features, but this is because the linear search is much more efficient for binary features and all the speedups are computed relative to the linear search time for that particular feature. To better compare the efficiency of the nearest neighbor search between different features types, we show in figure 6 the same results, but this time using the absolute search time instead of search speedup. It can be seen that the search time for binary features (BRIEF, ORB) is similar to that of vector features (SIFT, SURF) for high search precisions.

In figure 7 we compare the hierarchical clustering tree with a multi-probe locality sensitive hashing implementation [19], also available in FLANN. For the comparison we used datasets of BRIEF and ORB features extracted from the Winder/Brown datasets and from the recognition benchmark images dataset of [20]. The first dataset contains approximately 100,000 features, while the second contains close to 5 million features. It can be seen that for the first dataset the hierarchical clustering tree performs better for high search precisions ($> 80\%$), while the LSH implementation performs better for lower search precisions ($< 80\%$), while for the second dataset the hierarchical tree is faster than LSH for all precisions. This also shows that the algorithm proposed scales well with respect to the dataset size.

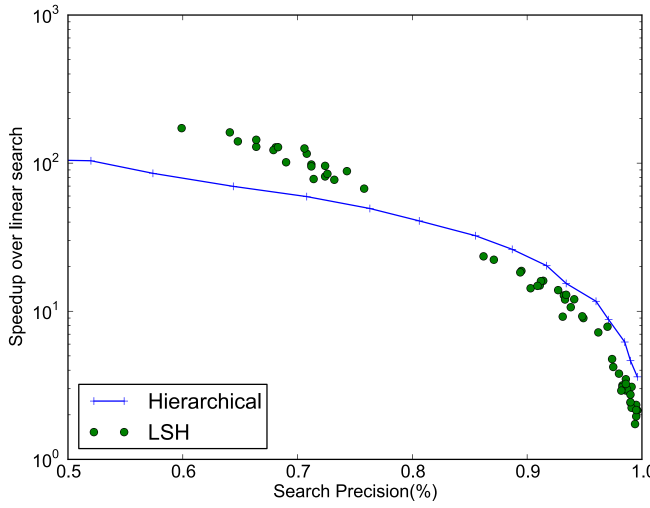
Another thing to note is that the LSH implementation requires significantly more memory compared to the hierarchical clustering trees for when high precision is required, as it needs to allocate a large number of hash tables to achieve the high search precision. In the experiments from figure 7 LSH required 6 times more memory than the hierarchical search in case of the larger dataset. Having both LSH and the hierarchical clustering trees implementations in FLANN makes it possible to use the automatic algorithm configuration from FLANN, which takes into account both memory constraints as well as computation time to select the optimal algorithm for a specific dataset.

C. *Algorithm scalability for large datasets*

We use the 80 million tiny images dataset of [18] to demonstrate the scalability of the algorithm presented in this paper. Figure 8 shows the search speedup for a dataset of 80 million BRIEF features extracted from the images in the dataset. We use the MPI[2]-based search framework from FLANN to run the experiment on a computing cluster and show the resulting speedups with respect to the search precision for different number of parallel processes. It can be seen that the search performance scales well with the dataset size and it benefits considerably from using multiple parallel processes.

In addition to the improved search performance, using multiple parallel processes on a compute cluster has the

[2]MPI (Message Passing Interface) is a standardized library specification designed for implementing parallel systems



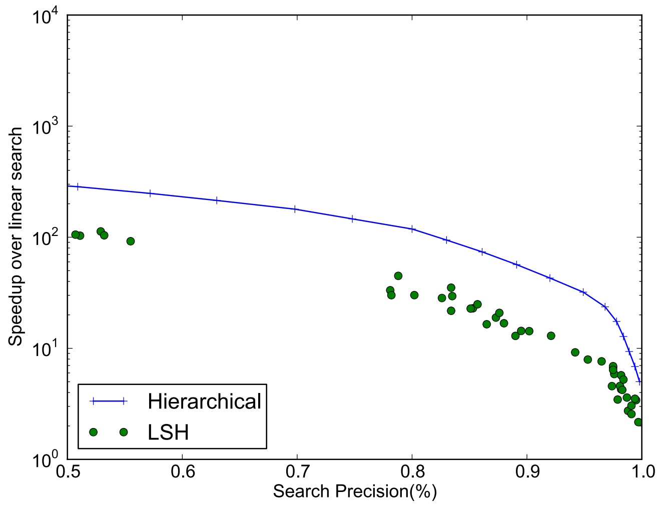


Figure 7. Comparison between the hierarchical clustering index and LSH for the Winder/Brown dataset of about 100,000 features(left) and the Nister/Stewenius recognition benchmark images dataset of about 5 million features(right)

additional benefit that the size of the dataset is not limited by the memory available on a single machine. Even though binary features are generally more compact in than vector-based features, for very large datasets the memory available on a single machine can quickly become a limiting factor. The distributed search framework from FLANN uses a map-reduce algorithm to run the search on multiple machines by splitting the input dataset among the different machines in the compute cluster, running the search on each machine on a fraction of the dataset and merging the search results at the end.

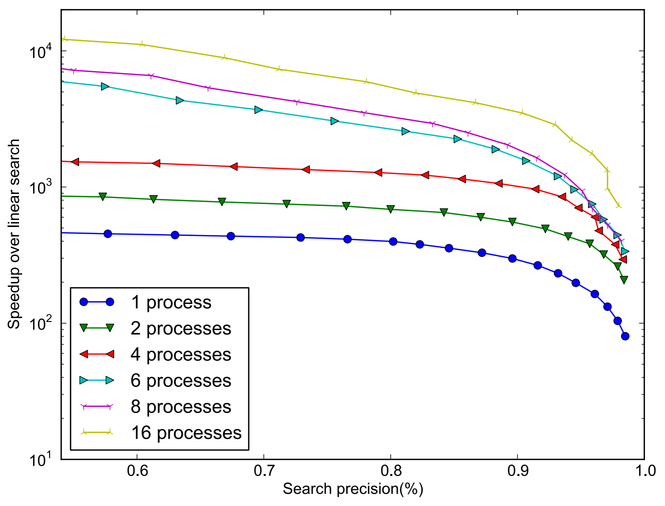


Figure 8. Scaling binary feature matching using a compute cluster (80 million BRIEF features)

V. Conclusions

In this paper we introduced a new algorithm for fast approximate matching of binary features using parallel searching of randomized hierarchical trees. We implemented this algorithm on top of the publicly available FLANN open source library.

We have shown that the proposed algorithm, although simple to implement and efficient to run, is very effective at finding nearest neighbors of binary features. We have also shown that the performance of the algorithm is on par or better with that of LSH, the algorithm most often used at present for binary feature matching, and that it scales well for large datasets.

References

[1] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, “BRIEF: Binary Robust Independent Elementary Features,” in *European Conference on Computer Vision*, Sep. 2010.

[2] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: an efficient alternative to SIFT or SURF,” in *International Conference on Computer Vision*, Barcelona, 2011.

[3] S. Leutenegger, M. Chli, and R. Siegwart, “BRISK: Binary Robust Invariant Scalable Keypoints,” in *IEEE International Conference on Computer Vision (ICCV)*, 2011.

[4] D. D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[5] H. Bay, T. Tuytelaars, and L. Van Gool, “Surf: Speeded up robust features,” *European Conference on Computer Vision*, pp. 404–417, 2006.

[6] R. Salakhutdinov and G. Hinton, “Semantic hashing,” *International Journal of Approximate Reasoning*, vol. 50, no. 7, pp. 969–978, 2009.

[7] C. Zitnick, “Binary coherent edge descriptors,” *European Conference on Computer Vision*, pp. 1–14, 2010.

[8] M. Muja and D. G. Lowe, “FLANN - Fast Library for Approximate Nearest Neighbors,” http://people.cs.ubc.ca/∼mariusm/index.php/FLANN/FLANN.

[9] A. Torralba, R. Fergus, and Y. Weiss, “Small codes and large databases for recognition,” *CVPR*, 2008.

[10] Y. Weiss, A. Torralba, and R. Fergus, “Spectral hashing,” in *Advances in neural information*. NIPS, 2008.

[11] H. Jégou, M. Douze, and C. Schmid, “Hamming embedding and weak geometric consistency for large scale image search,” *European Conference on Computer Vision (ECCV)*, pp. 304–317, 2008.

[12] C. Strecha, A. Bronstein, M. Bronstein, and P. Fua, “LDA-Hash: Improved matching with smaller descriptors,” *PAMI*, no. 99, pp. 1–1, 2010.

[13] A. Gionis, P. Indyk, and R. Motwani, “Similarity search in high dimensions via hashing,” in *Proceedings of the International Conference on Very Large Data Bases*, 1999, pp. 518–529.

[14] S. Brin, “Near Neighbor Search in Large Metric Spaces,” in *VLDB*, 1995, pp. 574–584.

[15] C. Silpa-Anan and R. Hartley, “Optimised KD-trees for fast image descriptor matching,” in *CVPR*, 2008.

[16] M. Muja and D. G. Lowe, “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration,” in *International Conference on Computer Vision Theory and Application VISSAPP’09)*, 2009, pp. 331–340.

[17] S. Winder and M. Brown, “Learning Local Image Descriptors,” in *Computer Vision and Pattern Recognition, 2007. CVPR’07. IEEE Conference on*, vol. pages. Ieee, 2007, pp. 1–8.

[18] A. Torralba, R. Fergus, and W. Freeman, “80 million tiny images: A large data set for nonparametric object and scene recognition,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 11, p. 19581970, 2008.

[19] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, “Multi-probe LSH: efficient indexing for high-dimensional similarity search,” in *VLDB ’07 Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 950–961.

[20] D. Nister and H. Stewenius, “Scalable Recognition with a Vocabulary Tree,” in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2 (CVPR’06)*, vol. 2. Ieee, pp. 2161–2168.