A hierarchical SIFT matching implementation in C++

Matteo Pagin[1]

[1] *Department of Information Engineering, University of Padova*

Summary

In recent years the usage of binary-valued features such as BRIEF and ORB has experienced a dramatic upsurge, driven by the following advantages over vector-based solutions: faster computation, quicker comparison and a significantly smaller memory footprint. Nevertheless, these binaries features, especially in certain scenarios, lack the descriptive capability of vector-based features such as SIFT, which in this regard are still capable to outperform solutions like BRIEF and ORB. Therefore, the research efforts have been focusing on combining the strengths of these two approaches. In this regard, a fast SIFT matching algorithm is hereby presented and implemented, featuring the binary quantization of SIFT features in conjunction of a hierarchical-based binary matching technique.

Introduction

The task of detecting salient points and computing their description as features has always been of primary importance in computer vision. This problem has been tackled using primarily two approaches, namely binary descriptors such as BRIEF [1], ORB [2] and vector-based solutions like SIFT [3]. While the former category of descriptors is fast to compute and has a very limited memory footprint, the latter exhibit a top of the class robustness wrt scale and rotation changes, on top of unmatched discriminative capabilities over large datasets [4, 5]. It follows that transforming vector features into binary ones apparently promises to offer the best of both worlds; this work is in fact based on the same premise. In particular, the presented implementation leverages a median quantization strategy, which manages to obtain satisfying accuracy while exhibiting low computational complexity [6].

In turn, matching such descriptors has also been another key problem and in such regard the focus of features matching algorithms has also changed over the years: since the size of the available datasets is growing at a staggering pace, the current challenge is to find significant matches in contexts where an exhaustive, linear search is unfeasible. Even though the majority of algorithms sharing such target are based on an approximate nearest neighbour strategy and/or hashing techniques, one of the most promising algorithms for computer vision applications relies on a hierarchical decomposition of the search space [7].

The purpose of this work is to analyze the performance of the combination of the aforementioned approaches, namely the hierarchical-based matching of median-quantized SIFT features.

Median quantization

A SIFT feature is basically an histogram of gradient directions that is encoded as a 128-dimensional vector. Since its components take values in the range $[0, 255]$ it follows that SIFT

vectors live in a space of 256^{128} possible points. In practice though only a small portion of this huge space is actually used: by examining the marginal distributions of the single components it is possible to notice an exponential-like distribution [6], a phenomenon that indicates that these descriptors do not actually fully exploit their theoretically available expressive power. This is demonstrated even further by the fact that pairwise distances are also relatively small compared to the values that can be expected of in such high dimensional space. It follows that it should be possible to compress the information contained in a SIFT descriptor into a smaller vector, in fact even down to the binary case. In this regard, since the primary goal of this quantization is to preserve the pairwise distances, namely to map similar features vectors into similar binary representations, the optimal strategy is to choose as quantization threshold the median value of a given vector's components. This choice maximizes the entropy of the quantized feature and it has been empirically shown that it also preserves the pairwise distances, leading to matches that are consistent with the original features representation [6].

Fast matching of binary features

Once the SIFT features have been quantized into binary vectors, the presented implementation searches similar descriptors by using the fast matching technique for binary features introduced in [7]. This algorithm essentially decomposes the search space by first building a tree whose non-leaf nodes represent cluster centers and leaf ones contain disjoint subsets of the original features set. Finally this hierarchical structure is used in order to look for matches by performing a number of comparisons that is significantly smaller compared to the linear, exhaustive search case. Clearly this algorithm can be decomposed into two phases: the computation of the hierarchical structures and the subsequent, actual search; the aim is to optimize the complexity of the latter, as the former can be sensibly considered a pre-computation.

Algorithm 1 Creation of a hierarchical clustering tree

Input: Dataset of features D, branching factor K and max leaf set size S_L
Output: Hierarchical clustering tree

```
if ||D|| < S_L then
    create leaf node set with the points of D
else
    P ← K points selected randomly from D
    C ← list of cluster obtained by partitioning D around the points of P
    for each cluster C_i ∈ C do
        create non-leaf node storing the ith center
        recursively apply the algorithm to the points in C_i
    end for
end if
```

The creation of the tree is performed by recursively clustering the original set into a pre-determined amount of clusters K: first of all K points of the dataset are randomly chosen as centers and stored in non-leaf nodes of the tree, then the set points are partitioned around such centers and are placed as their children. Such process it then repeated in a recursive manner until the cluster size is smaller than a given threshold, as can be seen in Alg. 1. Actually, the algorithm uses multiple copies of such trees; the rationale behind such choice is that since the centers selection is completely random, different trees are likely to have extremely different structures that may lead to quite heterogeneous times needed to reach the target leaf node in the tree. Therefore, by using these structures in parallel the resulting search time is the

minimum of such value across all the trees, effectively increasing the chance of finding a match in a small amount of time.

Algorithm 2 Parallel search of hierarchical clustering trees

Input: Hierarchical clustering trees T_i, query point Q, max amount of points to examine L_{max}
Output: K nearest approximate neighbours of Q

$L \leftarrow 0$ counter of searched points
$PQ, R \leftarrow$ empty priority queues
for each tree T_i **do**
  `TraverseTree`(T_i, PQ, R, Q)
end for
while $PQ \neq \emptyset$ and $L < L_{max}$ **do**
  $N \leftarrow$ top of PQ
  `TraverseTree`(N, PQ, R, Q)
end while
return K points of R that are closer to Q

Algorithm 3 `TraverseTree`(N, PQ, R, Q)

Input: Specific node of a clustering trees N, query point Q, amount of leaves examined L
Output: Updated main queue R and auxiliary refinement queue PQ

if N is a leaf node **then**
  linearly search all points $\in N$ and add them to R
  $L \leftarrow L + \|N\|$
else
  $C \leftarrow$ child nodes of N
  $N' \leftarrow$ closes node of C to Q
  $C_p \leftarrow C \setminus N'$
  add the nodes of C_p to PQ
  `TraverseTree`(N', PQ, R)
end if

The actual process of scanning these hierarchical data structures in order to find significant matches starts by performing a single traverse of the pre-computed trees. During this initial traversal, the node which is the closest to the query is picked and the search continues in a recursive fashion on that very same node until a leaf is reached. Upon such event, the points contained in the leaf are linearly scanned and added to the main queue R; on the other hand the nodes that we neglect along the way are progressively added to an auxiliary priority queue PQ, as explained in Alg. 3. Once the first traverse has been undertaken, a refinement of the search is performed by progressive traversals of the nodes contained in PQ, until either we reach the end of the queue or the amount of points visited surpasses the threshold L_{max}. The latter process allows to improve the accuracy of the results: in fact L_{max} gives as the possibility to fine-tune the trade-off between precision of the search algorithm and its computational cost, therefore rending the whole procedure flexible and possibly suited to different use-cases. Finally, the max amount of leaves and the branching factor do not have a significant impact on the search performance but they have the potential to sensibly influence the tress build time. Once again, this allows to tune the algorithm to the specific application, as some of them may consider the tree creation as a pre-computation, while others may need to do that in real-time, hence exhibit the need of faster hierarchical structures creation.

Implementation details and performance evaluation

The implementation that is hereby presented[1] leverages both aforementioned algorithms, specifically it uses median quantization on SIFT features and then finds matches among them by performing the fast matching introduced in [7]. The programming language that has been chosen for the task is C++, due to the performance-oriented nature of these algorithms. On top of the language standard library, the implementation makes use of an open-source, template-

[1] Available at: `https://github.com/pagmatt/bin-features-matching`

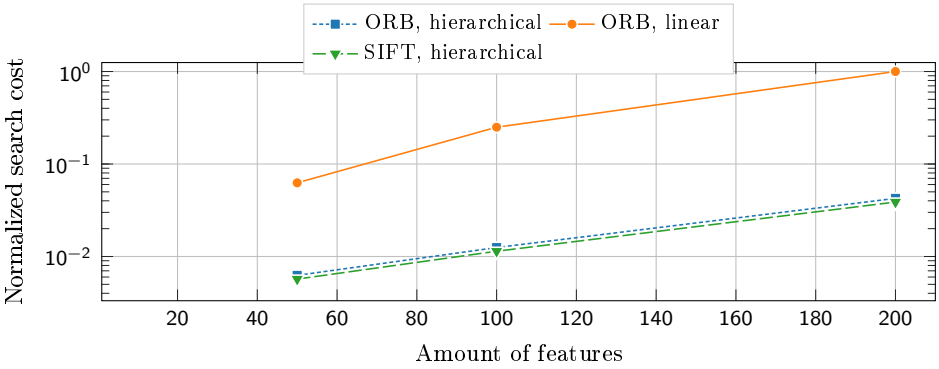

Figure 1: Normalized computational cost of the search subroutines as the amount of computed features ranges $\in \{50, 100, 200\}$. The Y axis uses a logarithmic scale.

based tree library[2] and of OpenCV[3] in order to compute the ORB and SIFT features from the dataset images.

In order to carry out the performance evaluation of the presented strategy a linear, exhaustive search among the whole dataset approach has been selected as baseline. Regarding the performance indicators, computational cost and search accuracy have been elected as the two most crucial factors. Furthermore, the complexity of the algorithm has been measured in number of CPU cycles (and their relative load), in order to provide as much abstraction from the underlying hardware as possible. This goal has been met by executing the program via the debugger Valgrind, enabling its profiling tool Callgrind[4] and finally analyzing the output traces in Kcachegrind. An important remark is that during such profiling the tree creation process has been neglected, focusing on the computational cost of the search subroutines only. As can be appreciated in Fig. 1, the implemented strategy manages to significantly outperform the linear search approach: in fact the hierarchical binary search of quantized SIFT features is also marginally faster than the ORB case: with the quantization process each SIFT descriptor is encoded in as little as a 128 bits vector, while an ORB feature requires from a min of 128 to a maximum of 512 bits. Therefore, the pair-wise Hamming distances computation of quantized SIFT features is actually faster than the ORB case, since it involves a smaller amount of bits.

Finally, the resulting search accuracy can be observed in Fig. 2, where the top matches for both ORB and SIFT are exhibited. These samples are obtained through a skimming process of the initial matches set which is based on the Nearest Neighbour Distance Ratio (NNDR) criteria introduced in [3].

Figure 2: Top 6 matches returned by ORB (on the left) and SIFT (on the right). In both cases, the leftmost images are the search queries while the rightmost ones are the corresponding matches.

[2] `https://github.com/kpeeters/tree.hh`
[3] `https://github.com/opencv/opencv`
[4] `https://linux.die.net/man/1/callgrind`

As expected, SIFT features are capable of offering stronger descriptive capabilities and to outperform in such regard the ORB descriptors, providing in the majority of the cases matches that are indeed relevant ones. Furthermore, perhaps surprisingly, such superiority is not lost after the quantization process, suggesting that in such regard the choice of a median threshold in indeed a successful one.

Conclusions

The presented implementation combines a binary quantization technique with an approximate matching algorithm, managing to provide accurate results with a low computational complexity. Therefore, it is well-suited to modern computer vision applications that need to interact with the massive datasets of the current Big Data era.

References

[1] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *European conference on computer vision*. Springer, 2010, pp. 778–792.

[2] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*. IEEE, 2011, pp. 2564–2571.

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

[4] K. Grauman and T. Darrell, "Efficient image matching with distributions of local invariant features," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 2. IEEE, 2005, pp. 627–634.

[5] E. Karami, S. Prasad, and M. Shehata, "Image matching using sift, surf, brief and orb: performance comparison for distorted images," *arXiv preprint arXiv:1710.02726*, 2017.

[6] K. A. Peker, "Binary sift: Fast image retrieval using binary quantized sift features," in *2011 9th International Workshop on Content-Based Multimedia Indexing (CBMI)*. IEEE, 2011, pp. 217–222.

[7] M. Muja and D. G. Lowe, "Fast matching of binary features," in *2012 Ninth conference on computer and robot vision*. IEEE, 2012, pp. 404–410.