Computing Nearest-Neighbor Fields via Propagation-Assisted KD-Trees

Kaiming He Jian Sun
Microsoft Research Asia

Abstract

Matching patches between two images, also known as computing nearest-neighbor fields, has been proven a useful technique in various computer vision/graphics algorithms. But this is a computationally challenging nearest-neighbor search task, because both the query set and the candidate set are of image size. In this paper, we propose Propagation-Assisted KD-Trees to quickly compute an approximate solution. We develop a novel propagation search method for kd-trees. In this method the tree nodes checked by each query are propagated from the nearby queries. This method not only avoids the time-consuming backtracking in traditional tree methods, but is more accurate. Experiments on public data show that our method is 10-20 times faster than the PatchMatch method [4] at the same accuracy, or reduces its error by 70% at the same running time. Our method is also 2-5 times faster and is more accurate than Coherency Sensitive Hashing [22], a latest state-of-the-art method.

1. Introduction

Non-parametric patch sampling is of central importance in various computer vision/graphics algorithms, including image inpainting/retargeting [31, 28], denoising [8, 10], texture synthesis [13, 30], super-resolution [17], matching self-similarities [26], and rendering [21]. A common step in non-parametric patch sampling is to compute approximate nearest-neighbor fields (ANNF) [4, 22]: given two images A and B, find for every patch in A a similar patch in B. ANNF computation is a special nearest-neighbor search problem, where the query/candidate set consists of all (overlapping) patches in the image A/B. It is challenging to compute ANNF quickly, because the sizes of both sets are very large.

Traditional approximate nearest-neighbor (ANN) algorithms organize the candidates to facilitate searching. A popular and effective way is through trees, such as the kd-tree [15] and other varieties [30, 11, 23, 24]. Tree-based methods organize the candidates adaptively to their distribution in the searching space. A query can find its ANN by checking a small portion of candidates. Tree-based methods are still among the state-of-the-art ANN solutions nowadays [24, 29]. However, the backtracking behavior [15, 2] in the tree methods prevents interactive or real-time computation of ANN *fields*, where the amount of queries is massive.

The queries are treated individually in the traditional ANN methods. But they are strongly dependent in ANNF problems, because they are overlapping patches from the same image. A milestone method called PatchMatch [4] observes that the images are coherent, so the matching result of a query patch can be propagated to the nearby queries and thus be reused. This method is found to be much faster than kd-trees, enabling ANNF computation at interactive rate. Nevertheless, PatchMatch does not consider the candidate distributions and visits many implausible candidates. Also, the results tend to be trapped in local optimum due to the short-distance propagation.

In this paper, we propose Propagation-Assisted KD-Trees for efficient ANNF computation. Our key insight is that we can jointly exploit the distribution of the candidates (all patches in B) and the dependency of the queries (all patches in A). After organizing the candidates in a kd-tree, our method checks for each query its own leaf [1] and an extra leaf propagated from nearby queries. The candidates in the propagated leaf need not be spatially close, so the algorithm can jump out of local optimum. Our algorithm is very fast because it only checks a small number of candidates (*e.g.*, 20 per query) and has no backtracking. In addition to efficiency, our algorithm is highly accurate thanks to the data-adaptive structure of the kd-tree. In the experiments on the public data set [9], we observe 10-20 times speedup versus PatchMatch at the same accuracy, or 70% less error at the same running time. Very recently, a method called Coherency Sensitive Hashing (CSH) [22] improves PatchMatch by introducing a hashing scheme. We find that our method is 2 to 5 times faster than this latest state-of-the-art method and is more accurate.

Interestingly, our experiments also show a traditional kd-tree (where each query is treated independently) combined with a suitable representation has comparable performance with PatchMatch. This is in contrast to the results reported

[1] In our experiments a typical leaf contains 8 candidates.

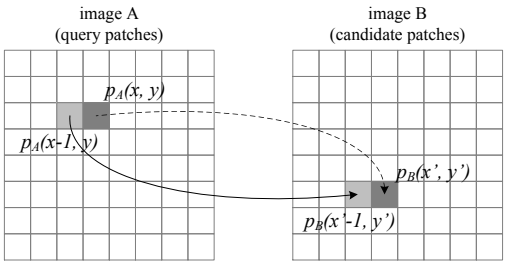

Figure 1. Propagation in PatchMatch. Each square represents the top left pixel of a patch, so the nearby patches are actually *overlapping*. The solid arrow indicates a good match. The dash arrow indicates a candidate to be checked.

in [4], which finds kd-trees are inferior. Our discovery indicates that candidate distributions (exploited by kd-trees) and query dependency (exploited by PatchMatch) can be almost equally contributing to ANNF computation. Thus combining these two aspects as in our method can make significant improvement.

2. Related Work

We review the works most related to our method: Tree-based methods, and PatchMatch and its improvements.

Tree-based methods. A classical method for ANN search is the kd-tree [15]. It is a binary tree where each node denotes a subset of the candidate data with a partitioning boundary of the data space. The partitioning is adaptive to the distribution of the data. Given a query, the search methods [15, 2] descend the tree to a leaf and backtrack other leaves that are close to the query. The searching accuracy depends on the amount of backtracking [2, 27].

To better organize the candidates and reduce backtracking, various types of trees have been proposed, including the k-means tree [16, 24], bbd-tree [3], TSVQ [30], vp-tree [23], rp-tree [11], multiple randomized kd-trees [27, 24], and so on. Though all these methods can improve accuracy, they spend extra effort in building the trees, *e.g.*, the building time of a k-means tree [24] is 10-20 times more than a kd-tree. This is not favored in ANNF computation, because we often have to build the tree *on-line*.

Recent studies [24, 29] show that tree-based methods, including the simple kd-tree, are still among the state-of-the-art ANN methods nowadays. But it is worth noticing that in these evaluations the queries are treated independently.

PatchMatch. The PatchMatch method [4] utilizes the dependency among the queries and performs searching collaboratively. It is observed that the images are coherent: the patches in a neighborhood in image A are likely to match the patches in a neighborhood in image B. Typically, if a pair of patches are similar, they are likely to remain similar when shifted one or some pixels simultaneously (see Fig.1). Thus the matching result of a query patch can be propagated to the next query, providing a good initial guess which is updated by some randomly sampled candidates. The random search in turn provides better sources for propagation. This process is iterated. PatchMatch has been shown 20 to 100 times faster than kd-trees plus Principal Component Analysis (PCA) [4]. But because PatchMatch does not organize the data beforehand, most randomly sampled candidates are unlikely to be good matches. And its local propagation often leads to over-smoothed results.

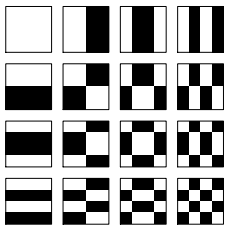

Figure 2. The first 16 Walsh-Hadamard Transform (WHT) bases (white = 1, black = -1). Each basis is a $4n \times 4n$-pixel kernel.

Most recently, a method called Coherency Sensitive Hashing (CSH) [22] improves PatchMatch by combining Locality Sensitive Hashing (LSH) [12]. All the patches (queries and candidates) are hashed into bins, and similar patches have a good chance to fall into the same bin. Random candidates are sampled from the bins that most potentially contain good matches. This hashing method is combined with propagation. This method is also iterative, switching the hashing function in each pass. It shows 3-4 times of speedup versus PatchMatch. But this binning scheme is in general unbalanced and not aware of the data distribution, and the bin sizes need to be adjusted carefully.

Random sampling is required in PatchMatch/CSH to reduce candidates checked. Our method has no randomness, and reduces candidates checked via data organization.

3. Algorithm

We observe that the distribution of the candidates and the dependency of the queries can be exploited jointly. We use a kd-tree with a suitable representation to organize the candidates. Then we propose a novel propagation-assisted search method for fast and accurate querying.

3.1. Patch Representation

A p-by-p patch in a color image can be represented by a vector in a $3p^2$-dimensional space. The similarity between two patches is described by the L_2 distance in this space. Because the kd-tree is less effective for high dimensional data, it is recommended to reduce dimensionality via PCA [27, 4]. But projecting each patch on each PCA basis requires slow $O(3p^2)$ time pre-computation.

Instead we use the Walsh-Hadamard Transform (WHT) [20] as the bases (Fig. 2). It is a Fourier-like orthogonal

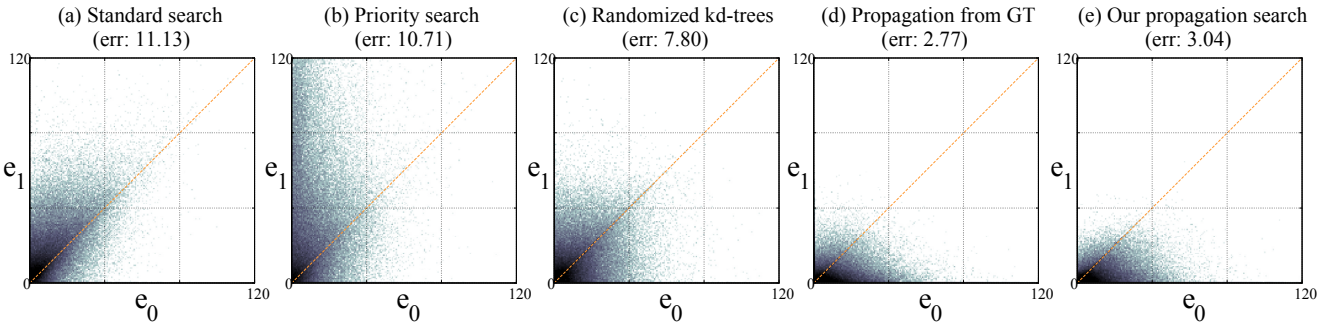

Figure 3. The joint distribution $P(e_0, e_1)$ in different search strategies: (a) standard search; (b) priority search; (c) randomized kd-trees; (d) propagation from GT; (e) our propagation method. A darker color means higher probability. The Leaf #0 is the same in all these figures, so the marginal distribution $P(e_0)$ is identical. We only change Leaf #1. In the brackets is the average error of the best candidate in the union set of Leaf #0 and #1. The error in Leaf #0 alone is 13.84. The error is reduced only when $e_1 < e_0$, *i.e.*, below the diagonal line. The distributions are computed using the images in [25]. Each leaf contains $m = 8$ candidates in this test. Here the error is computed through the 24-d representation.

transform. In natural images the first few WHT bases contribute a large portion of L_2 distance [20], just like the PCA bases. Speed-wise, projecting on each WHT basis requires only 2 operations per patch [6]. We use the first 16 WHT bases for the Y channel[2] and 4 for each chrominance channel (Cb/Cr) throughout this paper. Thus we represent each patch by a 24-d vector. In CSH [22] the WHT is used for constructing hashing functions.

3.2. Building a KD-Tree

After computing the WHT, we build a traditional kd-tree [15] in the 24-d representation space. Given any candidate set, we choose the dimension with the maximum spread[3] and split the space by the median value of the candidate data in this dimension. The median split is to ensure balance. The candidate set is divided recursively until the each terminal node (leaf) contains at most m candidates. We test m=8 to 64 in this paper.

3.3. Analysis of Search Strategies

In the search step, a query greedily descends the tree from the root to a leaf by checking the partitioning boundary in each node. We denote this leaf as Leaf #0. The query lies in the hyper-cube determined by this leaf, so any search strategy should check all candidates in Leaf #0. But there is a chance that better candidates lie in other leaves, *e.g.*, when the query is very close to the boundary. One may find a better result by checking other leaves. A common question in tree-based methods is: *how to determine the next leaves to be checked?*

The *standard search* method [15] backtracks in depth-first order and examines those leaves close to the query. Denote the first leaf visited in backtracking as Leaf #1. We define e_1 as the error of the best match in this leaf:

$$e_1(p_A) = \min_{p_B \in \text{Leaf } \#1} (\|p_A - p_B\|_2) - \|p_A - \hat{p}_B\|_2, \quad (1)$$

where p_A is a query patch, p_B is a candidate patch, and $\hat{p}_B$ is the true nearest neighbor in the whole image. We can define e_0 as the error of the best match in Leaf #0 in a similar way. To see how well Leaf #1 improves the result, we test on the images in [25] and plot the joint probability distribution $P(e_0, e_1)$ in Fig. 3 (a). Notice that Leaf #1 can improve the result only when $e_1 < e_0$, *i.e.*, below the line $e_1 = e_0$. But the distribution is mostly above this line. In this example the average error in Leaf #0 alone ($\text{mean}(e_0)$) is 13.84, and combining Leaf #1 reduces the error to 11.13 ($\text{mean}(\min(e_0, e_1))$). The improvement is minor.

The *priority search* method [2] backtracks the sub-trees in the order of their distance to the query, so better leaves are expected to be visited earlier. In this case, the distribution $P(e_0, e_1)$ is shown in Fig. 3 (b). We see that most points are still located above the line $e_1 = e_0$, and the improvement is not much (error = 10.71).

In [27] it is observed that the leaves visited via backtracking are mutually dependent, and the improvement of visiting more leaves is diminishing. The method in [27] instead builds multiple *randomized kd-trees*, so the searches are largely independent among trees. Here we generate Leaf #1 by descending a second kd-tree with random rotation [27]. We find the joint distribution $P(e_0, e_1)$ is well symmetric (Fig. 3 (c)). Almost a half of the results are improved by Leaf #1. The error is reduced to 7.80. This is perhaps the best outcome we can expect for any *individual* query after checking two leaves.

But the queries in ANNF tasks can be handled *cooperatively*. If a query has found a good match, it can provide information for the nearby queries. Suppose a query patch

[2]Computing 16 WHT bases requires p to be a multiple of 4. But we can relax this constraint by interpolation and allow any patch size.

[3]The *spread* is defined as the difference between the largest and smallest values in the dimension [3].

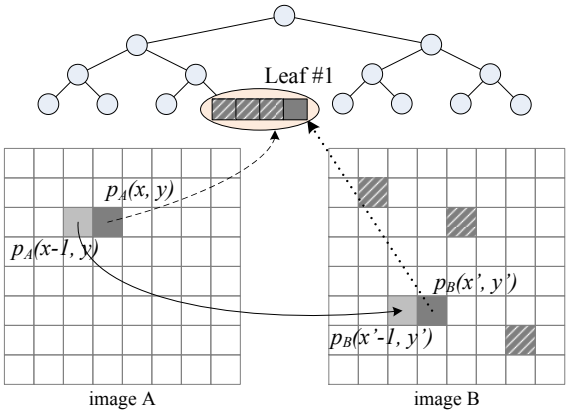

Figure 4. Propagation-assisted kd-tree. The symbols are analogous to those in Fig. 1. The dotted arrow indicates the propagated leaf. Given the matching result $p_A(x-1, y) \rightarrow p_B(x'-1, y')$, the candidates for $p_A(x, y)$ are all the patches in the leaf that contains the patch $p_B(x', y')$ (like the shaded ones).

$p_A(x-1, y)$ has found a similar patch $p_B(x'-1, y')$ (Fig.4). We use it to improve the result of $p_A(x, y)$. We take the leaf that contains $p_B(x', y')$ as Leaf #1. This is an extension of the propagation step in PatchMatch: we propagate a group of candidates instead of a single one. These candidates appear similar but need not be spatially nearby (see the shaded patches in Fig.4). We name this strategy as *propagation-assisted kd-tree search*.

In a very special case that $p_B(x'-1, y')$ is the *ground truth* best match of $p_A(x-1, y)$, we plot the error distribution of $p_A(x, y)$ in Fig. 3 (d). We observe that over half of them are improved by Leaf #1 (below the line $e_1 = e_0$). The error is significantly reduced to 2.77. This test suggests that finding a very good result without backtracking is possible, when the queries can propagate their information.

Though the propagation search using the ground truth best match of $p_A(x-1, y)$ to help the search of $p_A(x, y)$ is not practical, we can expect a similar performance if the match of $p_A(x-1, y)$ is good enough. Actually we can find a good match of $p_A(x-1, y)$ by another propagation search (propagated from $p_A(x-2, y)$), and so on. We describe the details in the next subsection. The distribution $P(e_0, e_1)$ in Fig. 3 (e) is generated by the Leaf #1 obtained in our algorithm. It is not much different from Fig. 3 (d), and the error (=3.04) is only slightly larger.

3.4. Propagation-Assisted KD-tree Search

Next we describe our search algorithm driven by the above analysis. Our algorithm scans the image A in raster order (from left to right, top to bottom). For a query patch $p_A(x, y)$ being scanned, we do the following 3 steps:

Step 1: Descend the tree to the Leaf #0.

Step 2: Propagate a leaf from left, *i.e.*, along x-axis (as in Fig. 4). Specifically, denoting $p_B(x'-1, y')$ as the result found by $p_A(x-1, y)$. We pick out the patch $p_B(x', y')$ and retrieve the leaf containing it. This leaf can be retrieved without any traversing[4]. Similarly, we also propagate a leaf from top, *i.e.*, along y-axis.

Step 3: Find the nearest-neighbor of $p_A(x, y)$ in all the leaves obtained in Step 1 & 2.

Our algorithm is non-iterative and finishes in one scan. It need not randomly sample candidates. The above algorithm alone performs quite well, but we can further improve its speed and accuracy by three more operations, namely, *enrichment*, *pruning*, and *re-ranking*.

Enrichment The kd-tree is a good method for searching k nearest-neighbors ($k > 1$). We adopt this property to further improve quality. We maintain the best k candidates for each query. Although this does not impact the outcome of the current query, the extra $k-1$ candidates can propagate (at most) $k-1$ leaves to the following query, enriching its candidates pool. The propagation is just similar to that in Step 2. In all the experiments we set $k = 2$. Combining with the "pruning" operation introduced below, the enrichment operation reduces the error by 15% with almost no extra time cost.

Pruning For each query we obtain at most $2k$ propagated leaves (k from left and k from top) besides the query's own Leaf #0. To reduce time cost, we select only one leaf from these $2k$ leaves. This is achieved by first checking the patch which "guides" us to a leaf (*e.g.*, $p_B(x', y')$ in Fig.4). We have $2k$ such guide patches and compute their similarities to the query, approximately measuring how good the leaves may be. We only keep the single leaf with the best guide patch, and ignore the others. The plot in Fig. 3 (e) is given by the Leaf #1 obtained in this way.

With this pruning technique our algorithm checks at most $2k$+$2m$ candidates per query: $2k$ "guide" patches, and $2m$ patches from Leaf #0 and #1 (each leaf has m patches).

Candidate checking and re-ranking Given the candidates in Leaf #0 and #1, the simplest way to find an ANN of a query is to compute its L_2 distance to each candidate using the 24-d representation. Although this strategy may lose accuracy due to dimensionality reduction, in experiments we find that its time-error tradeoff is satisfactory.

We can obtain a better solution by computing the L_2 distance in the original $3p^2$-dimensional space, but this is much slower. A compromising way is to find K nearest-neighbors

[4]We maintain a leaf pointer for each candidate p_B when building a tree.

using the 24-d representation, and re-rank these K candidates in the original space. Re-ranking not only improves the result of the current query, but also impacts the propagation quality. We set $K = 2$ in this paper[5]. We compare both strategies (without and with re-ranking) in experiments.

3.5. Complexity

Given an N-pixel image, the tree building time is $O(dN \log N)$ where $d = 24$ is the dimensionality. The reason for the factor d is because all dimensions of all data in each node are scanned to find the maximum spread. To reduce time cost, we only sample $1/d$ data in spread computation (in other operations we still use all data). This simplification rarely changes the first several branches, and has negligible influence on quality in experiments. The tree building time is thus reduced to $O(N \log N)$.

Suppose both images are of N pixels. The search time is $O(N \log N)$+$O(Nmd)$, where the first term is for descending the tree and the second term is for candidate checking. In practice the linear part $O(Nmd)$ is dominant, *e.g.*, when N=1Mp ($\log N$=20), m=8, and d=24. Table 1 shows the typical running time of each stage.

image size	WHT	tree building	search	total
N	*O(Nd)*	*O(N*log*N)*	*O(NlogN)* +*O(Nmd)*	
0.5Mp	0.08s	0.2s	0.32s	0.6s
2Mp	0.3s	0.9s	1.5s	2.7s

Table 1. Typical running time of each stage (m=8, no re-ranking).

The memory complexity of our method is about $O(Nd)$, mainly for storing the WHT coefficients. In practice, it takes around 100MB memory when N=1Mp.

4. Experiments

We compare our algorithm with PatchMatch (PM) and CSH. Our algorithm is implemented in C++. The PM and CSH codes are from the authors' websites [25, 9], both essentially in C++. All algorithms are run on a PC with an Intel Core i7 3.0GHz CPU and 8GB RAM. Here we compare single core implementations[6]. We experiment on the public data set *VidPairs* [9]. This is a challenging data set where each pair of images have very large displacement.

Time-accuracy tradeoffs In Fig. 5 we show the time-accuracy curves using 8-by-8 patches and 2Mp images. The accuracy is described by the average L_2 distance between each query patch and its nearest neighbor found by an algorithm, using the original RGB colors of the p-by-p pixels. The running time is averaged on all 133 image pairs in the data set. We test our method using $m \in \{8, 16, 32, 64\}$. We also test our method without and with re-ranking. All pre-computation time (WHT and tree-building) is included in all reports.

In Fig. 5 it is clear that our algorithm significantly outperforms PM and CSH. At the same accuracy our fastest setting (no re-ranking and m=8) is 18 times faster than PM and 3 times faster than CSH; a more precise setting (re-ranking, m=8) is 4-5 times faster than CSH to approach similar accuracy. Given the same running time, our method reduces the error (*i.e.*, the discrepancy between ground truth) of PM by 70% and the error of CSH by 50% in most settings (*e.g.*, on our curve "re-ranking"). Our algorithm manages to achieve very good accuracy (*e.g.*, re-ranking, m=64) that is not available in PM/CSH after 30 iterations.

When m=8 our method checks about 20 candidates per query. In comparison, this number is $\sim$60 in PM and 50 in CSH after 5 iterations (their default). Our method is more accurate even if it checks much fewer candidates. This is because the accuracy of PM/CSH can only be guaranteed given a sufficiently large number of random samples.

It is worth noticing that our better time-accuracy tradeoff versus PM is not purely due to the shorter representation of WHT. Though PM uses the original representation, the average *effective* dimensionality checked is only 20-40% of $3p^2$ due to "early stop" [4] (when the partial sum exceeds the current best sum), while the WHT hardly benefits from early stop due to its compact energy. Moreover, though the shorter WHT representation leads to faster candidate checking, it also impacts quality. In experiment we find PM+WHT generally has worse time-accuracy tradeoffs than PM alone.

We also compare our method with PM/CSH using various image sizes and patch sizes, as shown in Fig. 6. Similar performance comparisons are observed: our method is about 10-20 times faster than PM and 2-5 times faster than CSH to achieve the same accuracy.

Comparisons with traditional kd-trees We also compare the traditional kd-tree method (no propagation) in Fig. 5. We fix the parameter m=8 when building this tree. The marker "$\otimes$" (leftmost on the curve) shows the performance without any backtracking: it is better than the first iteration of PM/CSH, even if PM/CSH has already exploited propagation. This means that a kd-tree with WHT representations is a good way to organize the candidates. Although CSH also groups the candidates before searching, its binning structure is in general unbalanced and less data-adaptive. On the contrary, a kd-tree is fully balanced and well adaptive to the data.

[5]We have tested $K > 2$ in re-ranking (and also $k > 2$ in enrichment) and found that the gain is not much given the extra running time.

[6]Our method can be parallelized just like PM. In dual/quad-core configurations we observe similar speedup versus PM (parallel CSH has not been provided in [9]).

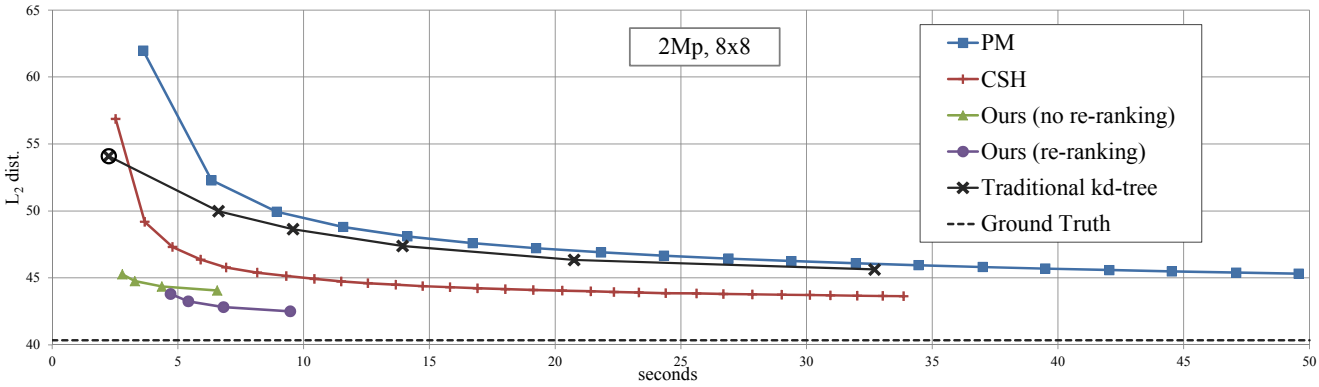

Figure 5. Time-accuracy tradeoffs averaged on 133 image pairs. The image size is 2Mp, and the patch size is 8-by-8. Each marker on PM/CSH's curve represents the performance after each iteration. Each marker on our method's curve represents its performance at each m value (m=8,16,32,64, from faster to slower). Each marker "$\times$" on the traditional kd-tree's curve represents its performance at each C value, where C=16,32,64,128,256 is the maximum candidates visited per query. Specially, the marker "$\otimes$" represents the traditional kd-tree without backtracking. The dash line is the ground truth average L_2 distance.

The markers "$\times$" in Fig. 5 show the performance of a traditional kd-tree with different strengths of backtracking. We use the standard backtracking implementation of the ANN lib[1]. We set the "error bound" $\varepsilon = 3$ and vary the maximum number C of candidates visited by any query (C=16 to 256). Fig. 5 shows that its performance is slightly better than PM. This is in contrast to the result reported in [4], where the kd-tree plus PCA is found inferior. Our result is reasonable: the traditional kd-tree exploits the candidate distributions and PM exploits the query dependency, and these two aspects can be equally contributing. All patches in the candidate/query set are from one or another image, so the data relations inside each set should have similar strength. Thus a traditional kd-tree can actually be comparable with PM.

ANN fields and error maps In Fig. 7 we demonstrate the ANNFs and the error maps. The ANNF is given by a map of the coordinates of the matched patches. In Fig. 7 we find that the ANNF of PM is much smoother than the ground truth. Our method and CSH overcome this problem because both methods allow non-local propagation. But our ANNF appears much more similar to the ground truth than CSH (*e.g.*, see the zoom-in regions).

On the error maps, we compute the error e of a query patch p_A by:

$$e(p_A) = \|p_A - p_B\|_2 - \|p_A - \hat{p}_B\|_2, \qquad (2)$$

where p_B is the matched patch given by any algorithm , $\hat{p}_B$ is the ground truth match, and $\|\cdot\|_2$ is computed by the original RGB representation. The error map of the ground truth matching is an all-zero map. Fig. 7 shows our matched patches have smaller error in general, typically on the edges and texture regions.

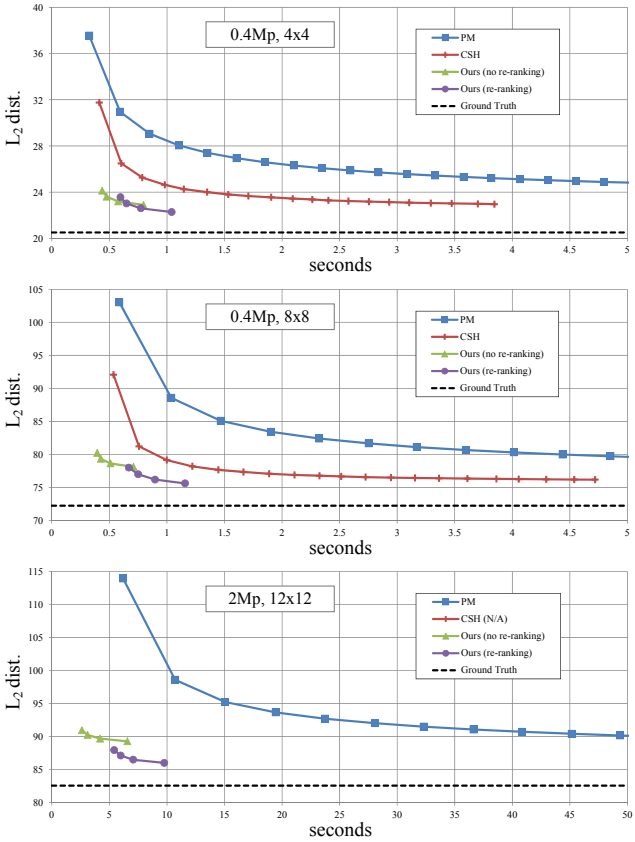

Figure 6. Comparison with PM/CSH in different settings. CSH is not available in the third figure because it does not support patch sizes other than 2^nx2^n.

Reconstruction One can reconstruct the image A by the matched patches found in image B. This is a main step in image inpainting, retargeting, and reshuffling [31, 28]. Any pixel in the reconstructed image is "voted" by all the patch-

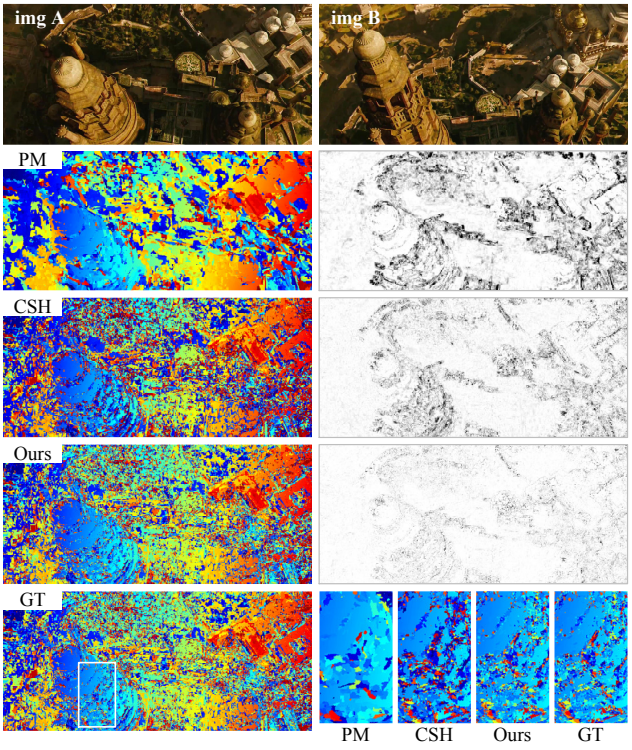

Figure 7. ANN fields and the error maps. **1st row**: inputs. **2nd-4th rows**: the ANN fields (left) and the error maps (right) of PM, CSH and our method. **5th row**: ground truth ANNF (left) and zoom-in of all the ANNFs. The ANNFs only show the x-coordinates, visualized in hue. In the error maps a darker pixel means larger error (a GT error map is all white). Here we use 0.4Mp images and 8x8 patches. The PM and CSH are after 5 iterations, and ours is using m=64 with re-ranking (running time: PM 2.3s, CSH 1.4s, and ours 1.2s). The images are from the VidPairs set.

es covering this pixel. In Fig. 8 we show the reconstruction results. The "GT" reconstruction is the result voted by the ground truth matched patches. Our results are visually better than PM/CSH in various cases like thin structures (Fig. 8 top), textures (Fig. 8 middle), edges (Fig. 8 bottom), and faces (Fig. 8 bottom, note the eyes and mouth). Our results are visually comparable to the ground truth.

When the image A is iteratively reconstructed using the same image B (as in [31, 28]), we only need to build the kd-tree using the patches in B *once*. Thus our speedups versus PM/CSH can be even greater in these applications.

Scalability Fig. 9 shows our running time at 10 different image sizes. The time is almost linear in image sizes, since the candidate checking time is dominant.

5. Discussions and Future Work

We have shown our method is faster and more accurate than existing methods including PatchMatch. But it is worth

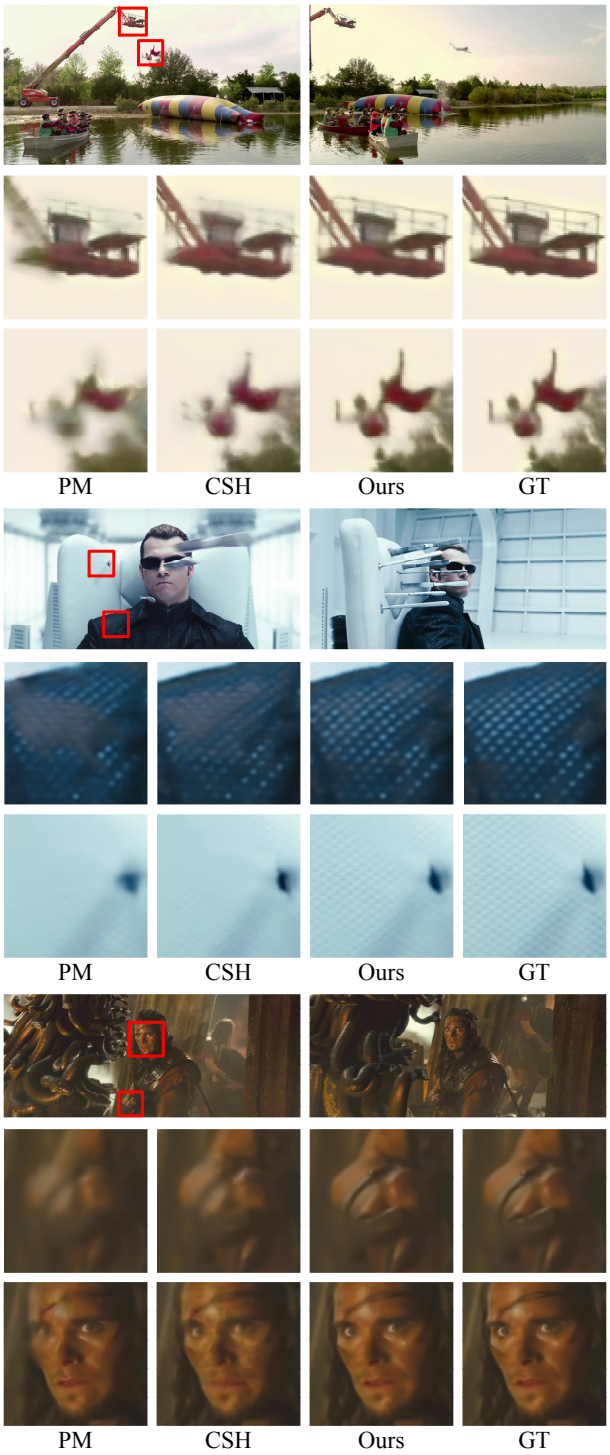

Figure 8. Visual comparisons of the reconstructed images. On the top of each group are the images A and B. The images are 0.4Mp and the patches are 8x8. PM and CSH are run 1 iteration, and our method is m=8 without re-ranking. Thus the running time of all methods is nearly the same ($\sim$0.5s). The images are from the VidPairs set. This figure is best viewed in the electronic version.

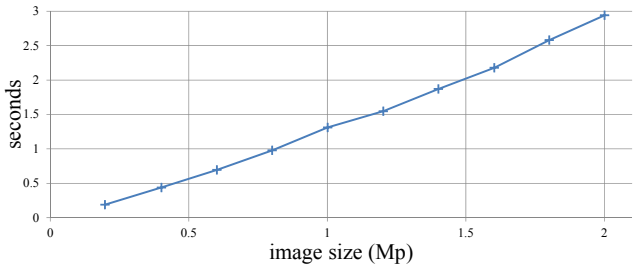

Figure 9. Scalability of our method (m=8, no re-ranking).

mentioning that PatchMatch has other advantages. First, PatchMatch has been generalized to search across scales and rotations [5] or color transformations [18]. Second, the similarity function (L_2) in PatchMatch can be extended to special forms other than L_p norms, such as those tailored for image matting [19] or stereo vision [7]. Besides, PatchMatch is more memory efficient. In the future we will improve our method in these directions.

The key idea of our method is that querying can be performed collaboratively in traditional ANN methods if the queries are dependent. It motivates us to study other similar scenarios. For example, in object categorization [14] overlapping patches are quantized by searching nearest-neighbors in a codebook. It may be accelerated by propagation search. We will study this topic in the future.

References

[1] ANN library. `www.cs.umd.edu/~mount/ANN/`.

[2] S. Arya and D. M. Mount. Algorithms for fast vector quantization. In *Proc. DCC '93: Data Compression Conf*, pages 381–390, 1993.

[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 1998.

[4] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. Patchmatch: a randomized correspondence algorithm for structural image editing. In *SIGGRAPH*, 2009.

[5] C. Barnes, E. Shechtman, D. B. Goldman, and A. Finkelstein. The generalized patchmatch correspondence algorithm. In *ECCV*, pages 29–43, 2010.

[6] G. Ben-Artzi, H. Hel-Or, and Y. Hel-Or. The gray-code filter kernels. *TPAMI*, pages 382–393, 2007.

[7] M. Bleyer, C. Rhemann, and C. Rother. Patchmatch stereo - stereo matching with slanted support windows. In *BMVC*, 2011.

[8] A. Buades, B. Coll, and J.-M. Morel. A non-local algorithm for image denoising. In *CVPR*, pages 60–65, 2005.

[9] CSH website. `www.eng.tau.ac.il/~simonk/CSH/`.

[10] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian. Image denoising by sparse 3-d transform-domain collaborative filtering. *TIP*, pages 2080–2095, 2007.

[11] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, 2008.

[12] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[13] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *ICCV*, page 1033, 1999.

[14] L. Fei-Fei and P. Perona. A bayesian hierarchical model for learning natural scene categories. In *CVPR*, pages 524–531, 2005.

[15] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, pages 209–226, 1977.

[16] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comput.*, pages 750–753, 1975.

[17] D. Glasner, S. Bagon, and M. Irani. Super-resolution from a single image. In *ICCV*, 2009.

[18] Y. HaCohen, E. Shechtman, D. B. Goldman, and D. Lischinski. Non-rigid dense correspondence with applications for image enhancement. In *SIGGRAPH*, 2011.

[19] K. He, C. Rhemann, C. Rother, X. Tang, and J. Sun. A global sampling method for alpha matting. In *CVPR*, pages 2049–2056, 2011.

[20] Y. Hel-Or and H. Hel-Or. Real time pattern matching using projection kernels. In *ICCV*, pages 1430–1445, 2003.

[21] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin. Image analogies. In *SIGGRAPH*, 2001.

[22] S. Korman and S. Avidan. Coherency sensitive hashing. In *ICCV*, 2011.

[23] N. Kumar, L. Zhang, and S. Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? In *ECCV*, pages 364–378, 2008.

[24] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, 2009.

[25] PatchMatch website. `http://gfx.cs.princeton.edu/pubs/Barnes_2009_PAR/`.

[26] E. Shechtman and M. Irani. Matching local self-similarities across images and videos. In *CVPR*, pages 1–8, 2007.

[27] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, pages 1–8, 2008.

[28] D. Simakov, Y. Caspi, E. Shechtman, and M. Irani. Summarizing visual data using bidirectional similarity. In *CVPR*, pages 1–8, 2008.

[29] A. Vedaldi and B. Fulkerson. Vlfeat: an open and portable library of computer vision algorithms. In *ACM Multimedia'10*, 2010.

[30] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH*, 2000.

[31] Y. Wexler, E. Shechtman, and M. Irani. Space-time video completion. In *CVPR*, pages 120–127, 2004.