Coherency Sensitive Hashing

Simon Korman, *Student Member, IEEE,* and Shai Avidan, *Member, IEEE*

Abstract—Coherency Sensitive Hashing (CSH) extends Locality Sensitivity Hashing (LSH) and PatchMatch to quickly find matching patches between two images. LSH relies on hashing, which maps similar patches to the same bin, in order to find matching patches. PatchMatch, on the other hand, relies on the observation that images are coherent, to propagate good matches to their neighbors in the image plane, using random patch assignment to seed the initial matching. CSH relies on hashing to seed the initial patch matching and on image coherence to propagate good matches. In addition, hashing lets it propagate information between patches with similar appearance (i.e., map to the same bin). This way, information is propagated much faster because it can use similarity in appearance space or neighborhood in the image plane. As a result, CSH is at least three to four times faster than PatchMatch and more accurate, especially in textured regions, where reconstruction artifacts are most noticeable to the human eye. We verified CSH on a new, large scale, data set of 133 image pairs and experimented on several extensions, including: k nearest neighbor search, the addition of rotation and matching 3 dimensional patches in videos.

Index Terms—Patch Matching, Image Matching, Nearest Neighbor Fields, Video Matching.

1 Introduction

In the Approximate Nearest Neighbor Fields (ANNF) problem, the goal is to quickly compute a mapping between the (dense) set of patches of one image to that of another, with a minimal L2 distance between the vector representations of the matching patches. Computing ANNFs is an important building block in many computer vision and graphics applications such as texture synthesis [12], image editing [35] and image denoising [8]. This is a challenging task because the number of patches in an image is in the millions and one needs to find Approximate Nearest Neighbors (ANN) for each patch in real or near real time.

In the past, it was customary to compute ANNF with traditional approximate nearest neighbor tools such as Locality Sensitive Hashing (LSH) [21] or KD-trees [1], [28]. These tools perform well in terms of accuracy but are not as fast as one would hope. Recently, a novel method, termed PatchMatch [4], proved to outperform those methods by up to two orders of magnitude, making applications that rely on ANNF run at interactive rate. The key to this speedup is that PatchMatch relies on the fact that images are generally coherent. That is, if we find a pair of similar patches, in two images, then their neighbors in the image plane are also likely to be similar. PatchMatch uses a random search to seed the patch matches and iterates for a small number of times to propagate good matches. Unfortunately, PatchMatch is not as accurate as LSH or KD-trees and increasing its accuracy requires more iterations that cost much more time. In addition, the main assumption it relies on (i.e. coherency of the image) becomes invalid in some cases (e.g. in strongly textured regions), with noticeable influence on mapping quality. It is therefore beneficial to develop an algorithm that is as fast, or faster, than PatchMatch, and more accurate.

• S. Korman and S. Avidan are with the Electrical Engineering Department, Tel-Aviv University.
E-mails: simonkor@mail.tau.ac.il; avidan@eng.tau.ac.il;

Coherency Sensitive Hashing (CSH) replaces the random search step of PatchMatch with a hashing scheme, similar to the one used in LSH. As a result, the process of seeding good matches is much more targeted and information is propagated much more efficiently. Specifically, information is propagated to nearby patches in the image plane, as is done in PatchMatch, and to similar patches that were hashed to the same value. In other words, we propagate information to patches that are close in the image plane or are similar in appearance. The end result is that our algorithm runs faster and gives more accurate results, in terms of root mean square (RMS) error of the retrieved patches, compared to PatchMatch. This increased speed and accuracy comes at a modest increase in memory footprint since we need to store the hashing tables.

An interesting property of our algorithm is that its reconstruction errors are significantly lower than those obtained by PatchMatch. To measure this, we define *incoherency* to measure the number of neighboring patches in one image that are mapped to neighboring patches in the other image. We find that the mappings produced by CSH are much less coherent than the ones produced by PatchMatch. This is because CSH does not rely on the image coherency assumption as much as PatchMatch does. Experiments suggest a strong correlation between the coherency of the mapping and RMS error. The less coherent the mapping, the lower the error. We also characterized the errors by image content and found that CSH works better than PatchMatch in textured regions. We demonstrate the advantages of CSH over PatchMatch on a new data set of 133 image pairs with 2 mega pixel resolution[1]. A prior version of this paper appeared in [24].

1. Code, data-set are available at: www.eng.tau.ac.il/~simonk/CSH

2 Related Work

Patch-based methods have been very successful in a wide variety of computer vision and graphics applications. Efros and Leung [12] introduced a simple non-parametric texture synthesis algorithm. It was quickly followed on and improved by [11], [25], [38]. Non-parametric texture synthesis was then used for various image editing applications by Simakov *et al.* [35] and it also inspired the method of non-local means for image denoising [8].

Common to all these techniques is the need to find, for each patch in image A, a similar (i.e., ANN) patch in image B, where in some cases images A and B can be the same image. Wei and Levoy [37] proposed a Tree Structure Vector Quantization (TSVQ) method to quickly find the necessary matches. Others relied on existing ANN search techniques such as kd-trees [1], perhaps enhancing them with PCA, to reduce dimensionality.

Ashikhmin [2] was the first to introduce the concept of coherency and used it to accelerate non-parametric texture synthesis. This was later extended to k-coherence by Tong *et al.* [36] that pre-computed a set of k nearest neighbors for each patch and used it to accelerate the search for ANN. They have also demonstrated it for texture synthesis. Alexe *et al.* recently proposed efficient algorithms to compute distances between pairs of windows between two images. Their approach exploits a theoretical relation between the windows' appearance distance and their spatial overlap.

Two leading methods for ANN search are kd-trees [1] and Locality Sensitive Hashing (LSH) [21]. Both partition the space, either deterministically (KD-tree) or randomly (LSH) in order to allow for quick query time. In this work we focus on LSH and show how to extend it to deal with coherent data, such as patches in an image.

The work most closely related to ours, and indeed the one that inspired ours, is that of PatchMatch [4]. PatchMatch takes image coherence to the extreme and uses it for various image editing applications. PatchMatch works in rounds. Given a pair of images it randomly assigns each patch in image A to a patch in image B. While most assignments yield poor matches, some are quite good and these are propagated to nearby patches in the image plane. To avoid being trapped in a local minima, it also performs a number of random patch assignments for each patch, keeping the best match after each stage. The algorithm usually converges after a small number of iterations.

The Generalized PatchMatch [5] is a recent extension of Barnes *et al.* that adds rotation and scale to the search space and allows searching for a dense K nearest-neighbor (KNN) fields between an image and itself, or two different images. An attempt to add appearance-guided information PatchMatch's search was reported in [3]. Non-Rigid-Dense-Correspondence [17] is another extension by Hacohen *et al.*, which uses PatchMatch with rotation and scale to transfer color between images. Also, He and Sun [19] successfully use the statistics of patch matching offsets, obtained by PatchMatch, in a graph-cut based solution for image completion.

Several other recent papers have improved on CSH and PatchMatch. The TreeCann algorithm [31] of Olonetsky and Avidan constructs a KD-Tree on a sparse set of image patches which were reduced to a low dimensional space to establish an initial correspondence, which is then refined by a local ,integral-image based, propagation stage. He and Sun introduce Propagation-Assisted KD-Trees [18], which exploits image coherency through a novel propagation search method through the a KD-Tree.

Another recent work, Besse *et al.'s* PatchMatch-Belief-Propagation [7], considers a generalization of the ANNF problem in which the resulting mapping is required to have a controlled level of smoothness. Their approach introduces a smoothness term in a BP formulation, controlling the smoothness of the NNF, and they show that a high level of coherence (or smoothness) of an ANN mapping could be useful for different applications such as stereo matching. In contrast, CSH and the other methods mentioned above, minimize only for the mapping error and do not control mapping coherency. As we claim in the paper, mappings that have high levels of incoherency (such as the ground truth mapping) are preferable for several common applications that are based on patch-based reconstruction (such as denoising and inpainting).

Many patch based methods for video enhancement (e.g. denoising [8], [9], [26], super-resolution [34], inpainting [38]) extensively compute dense matches between 3-dimensional space-time patches. Since the number of space-time patches in a video is enormous and since many of these methods need to run repetitive computations of the NNF (after updating the enhanced video at each round) - these methods are far from being real-time in practice. Different compromises have been done, allowing a certain improvement in runtime. Most common are limiting the search range for similar patches (in both space and time) or working with 2-dimensional patches in a frame-by-frame manner. These approaches may affect the visual quality and the temporal coherency of the enhanced video.

CSH reintroduces the usage of hashing to the ANNF problem. In recent years, there has been a large body of work on similarity preserving hashing, which is used e.g. for approximate nearest neighbor search in the setup of image retrieval. Such methods ([22], [16], [13], [29], [15], to name a few) follow the principle of Semantic Hashing [33], in which the Euclidean distance between hash codewords should correlate with the semantic similarity of the features. These are 'general-purpose' hashing codes in the sense that they can be applied to a general range of feature vectors such as the SIFT [27] or GIST [30] descriptors. In CSH, the hashing scheme is designed for the very unique setting of the ANNF problem where the vectors are patches of natural images (and being so they have certain characteristics), which are also densely overlapped.

3 LSH for Nearest Neighbor Search

The notion of *Locality Sensitive Hashing* (LSH) was first introduced by Indyk and Motwani [21]. Given a set of points in a metric space, LSH function families have the property that points that are close to each other have a higher probability of colliding (under random members of the family) compared to points that are far apart. The first usage of LSH for nearest neighbor search in high dimensions worked in high dimensional binary Hamming space [14]. It this work, we follow the general lines of an LSH-based ANN search scheme, later proposed by Datar *et al.* [10], which uses LSH families suitable for general L_p norms. In the rest of this section we outline their approach.

At the base of the algorithm is a family H of LSH functions and the ANN search algorithm consists of two stages: *indexing* and *search* (query). In the indexing stage, primitive hash functions from H are used to create an $index$ in which similar points map into the same hash bins with high probability. M such primitive hash functions are concatenated to create a *code* which amplifies the gap between the collision probability of far away points and the collision probability of nearby points. Such a code creates a single hash table, by evaluating it on all data-set points. In the search stage, a query point is hashed into a table bin, from which the nearest of residing data-set points is chosen. In order to decrease the probability of falling into an empty bin (with no data-set points), multiple (L) random codes are used to create L hash tables, which are searched sequentially at search stage. Datar *et al.* [10] show that the above scheme results in significantly improved efficiency compared to previous methods in the case of L_2 distances, which are the ones of interest in our case.

The LSH scheme of Datar *et al.* uses the particular family of LSH functions of the form

$$h_{a,b}(v) = \frac{a \cdot v + b}{r} \tag{1}$$

where r is a predefined integer, b is a value drawn uniformly at random from $[0, r]$ and a is a d-dimensional vector with entries chosen independently at random from a Gaussian distribution. The action of such a random function of this distribution (family) on a vector v (or patch) could be described by the 3 following stages: (1) Take a random line, defined by the vector a, divide it into bins of constant width r and shift this division by a random offset of $b \in [0, r)$ (2) Project the vector v on to the line (3) Assign it a hash value, being the index of the bin it falls into. The role of the random offset b is to neutralize the quantization limits of fixed binning. Specifically, it ensures that similar patches (which project to nearby locations on the line) will collide (fall into the same bin) with high probability.

A hash table T is then constructed, based on a function

$$g(v) = h_{a_1,b_1}(v) \circ ... \circ h_{a_M,b_M}(v) \tag{2}$$

which is a concatenation of M random projection-based hash functions h_{a_i,b_i}, independently created as described in the previous paragraph. Figure 2(a), illustrates the spatial decomposition induced by such a function g, built of two projection lines. Each spatial cell (e.g. the gray parallelogram) shows the area of points that are assigned a common hash value.

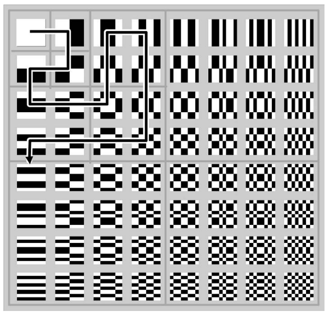


Fig. 1. Adapted from [6]: An array of the 64 Walsh-Hadamard (WH) kernels of dimension 8, ordered with increasing sequency in each column and row. For each kernel, white represents the value 1, and black represents the value -1. The 'snake' ordering (shown) forms a Gray-Code sequence of the kernels.

4 (2D)-Walsh-Hadamard Kernels

We make extensive use of the projection of image patches onto Walsh-Hadamard kernels. They are used (differently) in the two main stages of the hashing scheme. In the indexing stage, they are used in the definition of the underlying hashing functions, while in the search stage - for the efficient approximation of L_2 distances between patches. Even though our hashing scheme is a general framework and these components could have been implemented differently, the use of the WH kernels turns out to be a key ingredient in the good speed/accuracy tradeoff of the algorithm. We therefore elaborate next on the specific properties of the WH projections, which make them suitable for our needs. In Sections 5.1 and 5.2, we give further implementation details.

Figure 1 shows the 64 WH kernels, which form a complete basis of 8×8 pixel patches, alternative to the standard 'pixel-by-pixel' basis. Let's consider how these kernels are suitable for hashing. When designing our locality sensitive hash functions we will use a standard procedure of projecting each high dimensional point (patch) to some 1-dimensional linear subspace (i.e. a line). Clearly, points that were close in the original space, will be projected to close locations on the line. For general sets of points in R^d, much effort has been put into the choice of the distribution, from which such random lines are chosen. However, we would like to take advantage of the specific knowledge, that each point is a vector of pixel values of a square patch. Specifically, when projecting all the patches of an image onto a line, we would like the dispersion to be as large as possible, since this would make this line very discriminative with respect to patch similarity (namely, the distance between the projection of dissimilar patches will be large, while in the case of similar patches - small).

To this end, a natural solution would have been to take the leading eigenvectors of the covariance matrix of the entire set of image patches. In the case of natural images (not letting the choice of lines be image dependent), these turn out to be a sinusoidal basis, ordered in increasing frequency [32]. The 2D WH kernels, when ordered by increasing frequency (from top-left to bottom-right, in Figure 1) form such a sequence of projection lines. Indeed, our hash functions will be based on a fixed (non-random) set of several leading (low-frequency) WH kernels (details in Section 5.1).

The WH projections have been shown by Hel-Or *et al.* [20], [6] to be extremely descriptive and efficient for pattern matching in images. Since the entire WH basis $\{K_j\}$ is a complete and orthogonal basis, the squared L_2 distance between two patches p_1 and p_2, can be accumulated by summing the squared differences between the projections of the two patches on the basis elements. Namely:

$$||p_1 - p_2||_2{}^2 = \sum_j (K_j \cdot p_1 - K_j \cdot p_2)^2 \qquad (3)$$

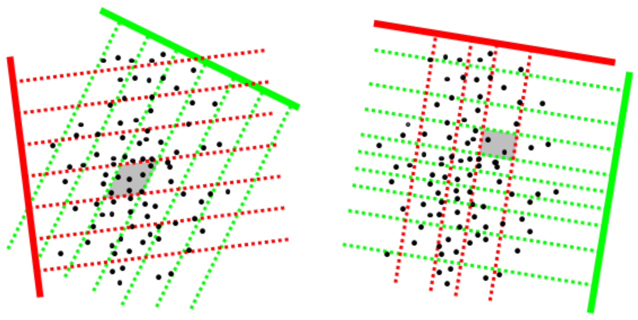
However, while in the standard computation of the L_2 distance (i.e. pixel-by-pixel summation of squared difference) the L_2 accumulates at a constant pace, in the case of the WH kernels with their increasing frequency ordering, the first projections typically capture most of the energy of the distance. This means that evaluating the distance with fairly few projections gives an efficient way to lower-bound the true distance.

Furthermore, assuming that the projections are computed for all the patches of an image - Ben-Artzi *et al.* [6] have shown that it can be done extremely efficiently (with only 2 additions per patch per kernel, regardless of patch dimension), in a specific ordering (see 'snake' ordering in Figure 1), which forms a Gray-Code sequence of the WH kernels. We therefore compute, a priori, a (top-left) subset of WH projections of all the patches of the image. This set of projections is used for L_2 distance approximation, as described above. A much smaller subset of these projections, is used for hashing (details in Section 5.2).

5 Coherency Sensitive Hashing (CSH)

In this section we layout our algorithm for approximate dense nearest patch search. The straight forward way to use the LSH search scheme for image patches would have been to treat each d-by-d patch as a d^2 vector in Euclidian space and the rest follows. However, it wouldn't take advantage of the very particular image patches setup, especially: (a) the (non-uniform) distribution of natural image patches; (b) the wide extent of overlaps between nearby patches.

Instead, we follow the general lines of the LSH scheme, but replace several of its main ingredients with new ones, which are designed to exploit the image patches setup. At the *Indexing* stage, we replace the family of LSH functions with a new set of functions, which make use of the Walsh-Hadamard kernels (details in section 5.1). At the *search* stage, we dramatically extend the set of candidate patches that are considered, compared to the limited set of patches that point to the same index (details in section 5.2). We term the resulting scheme *Coherency Sensitive Hashing* (CSH). The CSH Algorithm is given in algorithm 1, while the details are given in the next subsections.



(a) LSH (b) CSH

Fig. 2. **Spatial decomposition of LSH vs. CSH**. Points (patches) are projected to (solid) lines (green, red). Dashed lines represent the respective binning. Each projection line defines a hash function h_i, which assigns a bin number to the point. The final hash function g is a concatenation of the bin numbers, given by each h_i and points in the same 'cell' (gray area) share a hash code. There are three main differences between LSH and CSH hashing: (i) LSH uses random oriented lines, while CSH uses fixed orthogonal (WH) lines. (ii) LSH uses an equal number of bins per projection, while CSH allocates more bins (code bits) to more 'informative' projections. (iii) LSH bins are evenly spaced, while CSH bins are variably spaced aiming at a uniform spread of samples.

5.1 Indexing

Our indexing scheme is based on that of Datar *et al.* [10], which we introduced in Section 3. In our case, however, each vector is an image patch and we do not project it onto random lines, but rather on a fixed set of the leading (most significant) 2D *Walsh Hadamard* (WH) kernels. The motivation for this choice was given earlier in Section 4 and we now discuss the details, highlighting how this stage differs from the original one (in standard LSH hashing). The differences stems from the very different assumption on the underlying distribution of points - LSH handles general sets of high dimensional points, with an unknown distribution, while CSH deals with patches of natural images. The differences are highlighted in Figure 2 and can be summarized as follows:

Choice of projection lines: LSH uses random lines, according to a p-stable (e.g. gaussian) distribution [39]. In CSH, we use a fixed set of the lowest-frequency WH kernels.

Number of bins per projection: LSH uses bins of a fixed width r (which is tuned for different kinds of probabilistic guarantees of the NN search). Since points are vectors of (bounded) pixel intensities, this translates to an approximately fixed number of bins per projection. In CSH, we assign each projection line (associated with some WH kernel) a number of bins that is proportional to the dispersion of patch projection values. The higher the frequency of the kernel, the less the projections are dispersed, but rather they concentrate around 0. Also, projections of the luminance (Y) component of patches tends to be more wide spread compared to the chroma (Cb/Cr) components. In general, we restrict the number of bins per projection to be a power of 2 (since the hash code is binary).

Spacing of bins per projection: LSH induces uniform spacing, defined by the fixed width r. In CSH, on the other hand, we aim at having an even spreading of samples across the bins. This will enable making efficient usage of the limited number of bits that make up the code.

Following these principles, we needed to determine/configure the following parameters: (i) number of bits in the hash code; (ii) how these bits are distributed between which projections; (iii) For a certain projection - its bin boundaries (edges); In terms of (i) and (ii), we did the bit allocations manually (off-line), and keep them fixed for all images. Increasing the number of bits is equivalent to a denser subdivision of the space. This will guarantee better similarity of points that have the same code, but it comes at the risk of having many points, whose relevant NNs are all assigned different hash values. The fixed values, which were determined to ensure good overall behavior and were used throughout our experiments, are specified in Figure 3. Regarding item (iii), we found that the projection distribution of an image's patches on a certain kernel depends specifically on the nature of the image (and this is especially true for the leading, low frequency, kernels). This is solved by taking a large random sample of the image's patch projections on the certain kernel, using those to place our bins, such that an equal number of samples will be placed in each one. For instance, if we need to place 32 bins, these are placed at the relevant fixed percentiles of the sample.

Like in LSH, a random shift is applied to the binning (and this is in order to maintain the property that close points will be projected to the same bin with high probability), in the sense that percentiles are shifted by a random offset. We therefore expect to have a fixed percentage of samples in each of the bins, except for the first and last ones. Some examples of this automatic bin assignment are shown in Figure 4. Please see caption for details. Note also that the random shifting is the only source of variability between the L hash tables, since as opposed to the original LSH we use a fixed set of projection lines.

5.2 Search

In the Indexing stage we built a set of L hash tables, with the desired property of *local sensitivity* in the appearance plane. Namely, that similar patches (disregarding their image location) are likely to be hashed to the same entry.

patch dim.	$Y_{1,1}$	$Cb_{1,1}$	$Cr_{1,1}$	$Y_{2,1}$	$Y_{1,2}$	$Y_{2,2}$	$Y_{3,1}$	$Y_{1,3}$
16 or 8	32 (5)	4 (2)	4 (2)	8 (3)	8 (3)	2 (1)	2 (1)	2 (1)
4	32 (5)	4 (2)	4 (2)	8 (3)	8 (3)	-	2 (1)	2 (1)
2	32 (5)	4 (2)	4 (2)	8 (3)	8 (3)	-	-	-

Fig. 3. **Bit/Kernel allocations**: The table specifies the number of bins (and the number of bits, in parenthesis) allocated to each of the 8 WH kernels used in the hashing scheme. Each row specifies the allocations for a different size of patch: 8×8 and 16×16 (which use the same allocation), 4×4 and 2×2. Each column represents a specific WH kernel, with the following convention: Y/Cb/Cr stands for the channel the kernel is applied on. The pair of subscripts relate to the location of the WH kernel in the 2d arrangement, like the one shown for 8×8 patches, in Figure 1. Notice that the total length of the hash code will be 18 bits for patch dimensions of 8 or 16 and slightly shorter for the smaller patch sizes.

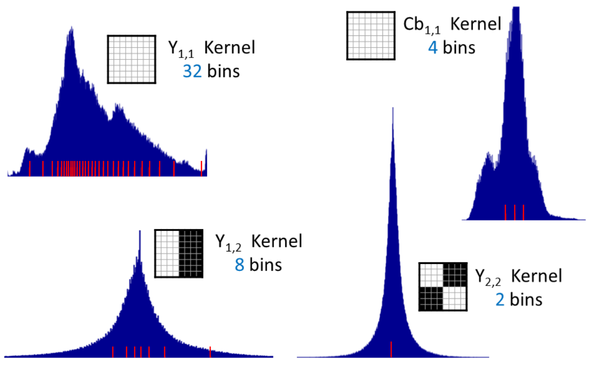


Fig. 4. **Hashing by binning of kernel projections**. The four histograms illustrate the different distributions obtained by projecting all patches of an image on different WH kernels (histograms are scaled to fit the figure). In each, the 8×8 kernel is shown with its name (see Figure 3 for the convention) and the number of bins. Examples - **Top-left**: (DC)-kernel applied on the Y channel produces a very dispersed distribution, which is divided into a large number of 32 bins (defined by the red bin edges, located at fixed percentiles). Besides the extreme 2 bins (due to random shift), each of the 30 other bins contains 1/32 of the image patches. **Top-right**: The DC distribution on a color channel is typically less informative compared to the Y channel. The distribution is more centralized and hence, is divided into less (only 4) bins. **Bottom-right**: As the frequency of the kernel increases, the distribution concentrates around zero.

The straight forward LSH search scheme would have simply implied, for each patch in image A, to consider the set of patches of image B, which are hashed to the same entry in any of the L tables. The resulting set of potential candidates is rather small, does not exploit the known spatial arrangement of the patches and does not allow propagation of information between patches. Instead, CSH runs a search over L iterations (one per hash table), in which it creates a rich set of candidates by combining cues of both *appearance* and *coherence* (of location) in a novel manner.

5.2.1 Candidate Creation

Let g_i denote the hash code (function) used to create the hash table T_i. To simplify the discussion, we'll drop the subscript and refer to a hash function g and the resulting hash table T. Furthermore, the hash function g will be denoted g_A when applied on patches of image A and g_B when applied on patches of image B. Let g_A^{-1} (g_B^{-1}) be the pre-image of g_A (g_B) and $Left(p)/Right(p)/Top(p)/Bottom(p)$ be the patch obtained as a result of shifting a patch p one single pixel to the left/right/top/bottom. In addition, let $Match(a)$ for any patch a in A be its nearest currently known patch in B.

Here are four observations that we use to create a large pool of candidates per patch of image A. Considering patches a, a_1 and a_2 of image A and patches b, b_1 and b_2 of image B:

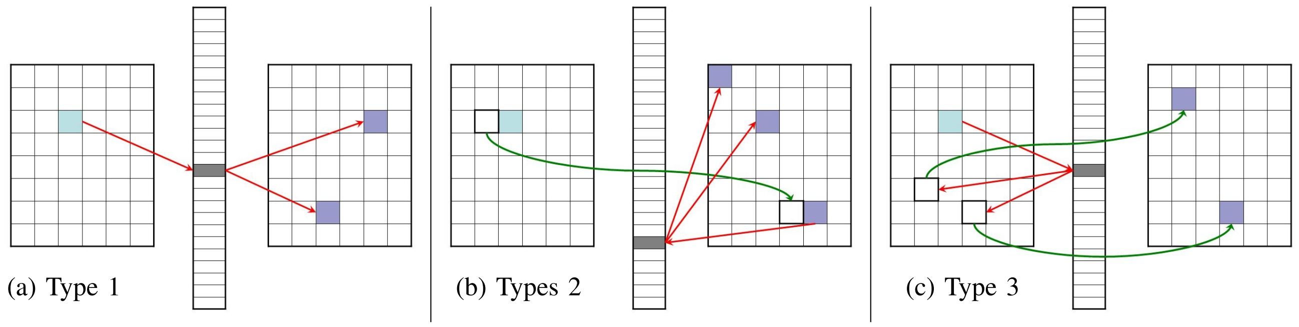


Fig. 5. **Candidate types for a patch.** In each of the sub-figures, Image A is on the left, image B is on the right and the hash table in use is in the center. Arrows relating to a pixel actually relate to the patch whose top left corner is at the pixel. Red arrows represent the hashing (notice their direction), while green arrows point to the patch's current best known representative. The highlighted pixels (patches) in image B on the right are the candidates of the highlighted pixel (patch) in image A on the left. If the $width$ of the hash table is defined to be w (i.e. it stores w representative patches from each of the two images) then the total number of candidates is between $4k$ and $4k+2$ (types 1 and 3 each contribute w candidates, while type 2 appears both in left/right and top/bottom configurations and contributes w or $w+1$ in each configuration). In our implementation (and this illustration) we use $w = 2$.

- observ. 1 (***appearance-based***): If $g_A(a) = g_B(b)$, then b is a (good) candidate for a.
- observ. 2 (***appearance-based***): If b is a candidate for a_1 and $g_A(a_1) = g_A(a_2)$, then b is a candidate for a_2.
- observ. 3 (***appearance-based***): If b_1 is a candidate for a and $g_B(b_1) = g_B(b_2)$, then b_2 is a candidate for a.
- observ. 4 (***coherence-based***): If b is a candidate for $Left(a)$, then $Right(b)$ is a candidate for a. [2]

Observations 1-3 follow from the local sensitivity property of the function g (which follows from the local sensitivity of its parts h). This happens in appearance space. On the other hand, Observation 4 follows from the coherency of patches in the image.

Here are 3 types of *candidate* patches we generate for a patch a of image A, via compositions of observations 1-4:

type	definition	using observ.
1	$g_B^{-1}(g_A(a))$	1 and 3
2	$g_B^{-1}(g_B(Right(Match(Left(a)))))$	3 and 4
3	$Match(g_A^{-1}(g_A(a)))$	2

These candidate types are further illustrated in Figure 5. In our implementation, we set the *width* of the table w (the number of patches of each of A and B that can be stored in a hash table entry) to be 2. We end up with $4k+2$ candidates (10 in our case) and a rough estimate on the individual type contributions to the final match is 20%, 50%, and 30%, respectively.

We can now compare the candidate patches used by CSH to those used by the different algorithms and notice how CSH generalizes them. LSH uses exactly the candidates of type 1. These candidates on their own are especially limited, mainly since they do not exploit image coherency (which is generally very high), but also since they do not take advantage of appearance similarity (hash collisions) between patches in image A. On the other hand, PatchMatch exploits only image coherency. It uses exactly 2 out of the 4-6 candidates of type 2 (Namely, $Right(Match(Left(a))$ and $Bottom(Match(Top(a)))$, in addition to random location candidates, using no cues of appearance whatsoever.

One clear limitation of PatchMatch, which our algorithm overcomes, is its assumption that mappings that are mostly (spatially) smooth may achieve pleasing approximations. The PatchMatch algorithm looks around the patch's neighbor's nearest patch (propagation) as well as at random patches around the current known nearest patch, with probability dropping exponentially with increase in distance. This approach works well on large contiguous areas that appear in both images, since a proper random guess will propagate to the whole area. However, it has difficulties in textured areas, which are not replicated in both images. In our approach, we intensively relate patches which collide under some hash function. Such collisions occur based entirely on the appearance of the pair of patches without any relation to their spatial arrangement. The spatial layout of our mapping is much less continuous compared to that of PatchMatch. This is evident in the second row of Figure 19, where the x-coordinates of both algorithm's mappings are presented.

5.2.2 *Candidate Ranking*

Given the candidate set (of size $4k+2$), all that remains is to find the nearest one. This step of the algorithm is actually the main overall time consumer. We therefore resort to an approximation of the process, which has a negligible impact on the overall precision but greatly reduces run time.

This is where we make a second use of the Walsh Hadamard (WH) projections, in a different way to how they were used in the indexing stage. We use them here in the way Hel-Or *et al.* used them in their rejection scheme for pattern matching [20]. The idea is that accumulating the squared differences between the projections of a pair of patches on the WH kernels, one at a time, produces an

2. This holds also for Right/Left Top/Bottom and Bottom/Top pairs

2. with the exception that we only store a random set of w of the patches mapped to the entry - w being the *width* of the table, as explained in Figure 5.

Algorithm 1 *Coherency Sensitive Hashing (CSH)*

```
Input: color images $A$ and $B$
Output: A dense approximate nearest patch map ANNF

Indexing (of all patches of images $A$ and $B$)
  1) Project each of the patches of both $A$ and $B$ on the WH kernels (defined in Fig 6) : $\{K_j\}_{j=1}^{N}$, using the Gray Code Kernels technique of [6].
  2) Create $L$ hash tables $\{T_i\}_{i=1}^{L}$. Table $T_i$ is constructed as follows:
     a) Define a code $g_i(p) = h_1(p) \circ ... \circ h_M(p)$ which is a concatenation of $M$ encodings of bin numbers under the projection of the kernels $K_j$ (see Figure 3 and Section 5.1).
     b) Then, each patch $p$ (of both $A$ and $B$) is stored in the table entry $T_i[g_i(p)]$[3].

Search
  1) Arbitrarily initialize the best candidate map ANNF
  2) Repeat for $i = 1, ..., L$ (for each hash table):
     a) For each patch $a$ in $A$
        i) Create a set of candidate nearest patches $P_B$ using the table $T_i$ and the current mapping ANNF (as described in section 5.2.1)
        ii) Let $b$ be the patch from $P_B$ which is most similar to $a$
        iii) If $dist(a, b) < dist(a, \text{ANNF}(a))$ then update: ANNF$(a) = b$ (distances are only approximated, see section 5.2.2)
  3) return ANNF
```

increasingly tighter lower bound on the squared Euclidean distance between the patches, following Equation 3. We use only the leading kernels out of the full basis (in decreasing frequency ordering), which capture a large enough portion of the patch's energy. This method incorporates an early termination mechanism, rejecting a candidate once the sum of projected differences exceeds the current nearest approximation of patch distance. In Figure 6 we specify the exact set of kernels, used for the approximation. Notice that as the patch dimension grows, we need a smaller and smaller portion of projections, enabling the ranking of candidates to be increasingly efficient. The typical case is that a patch p already has a NN patch q and most of the further candidates q' will not be as close to p as q is. When estimating the (squared) L_2 distance between p and q', it will most likely exceed the distance between p and q after very few projections and we will seldom need to evaluate the 'full' set of projections (which is also very partial, e.g. 23 out of 192 in the case of 8×8 patches.

6 Extensions

In this section we propose several extensions to the basic CSH scheme. We suggest two fully-automated data-driven variants of the CSH hash function in Section 6.1; a k nearest neighbor extension (useful for many patch based algorithms) in Section 6.2; a fast variant for computing a bi-directional NNF between an image pair in Section 6.3; an extension to 3D data, used for densely matching videos in Section 6.4; and a variant that adds a third DOF of 2D rotation to the standard 2D-translation mapping in Section 6.5.

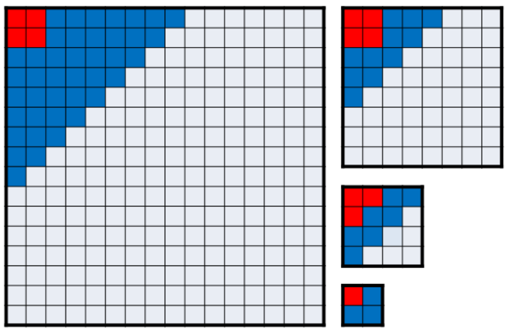


Fig. 6. **WH projections used for L2 distance approximation**. The illustration specifies the subsets of WH kernels, used for lower bounding L2 patches distances according to Equation 3, for patches of dimensions 16, 8, 4 and 2. The matrices depict 2d arrays of WH kernels, where kernels marked red are those used for all patch channels (Y/Cb/Cr), while the blue ones are used for the Y channel only. For example, in the 8×8 case, out of 192 kernels $(8 \cdot 8 \cdot 3)$, we use only 23 (15 for Y and 4 for each of Cb and Cr). Notice that the set of projections used for hash code construction is a strict subset of this selection. The Grey-Code-Kernels method for efficient projection computation, requires that each projection should be computed after *either* of the projection above or to the left of it. This is clearly possible with the proposed set of chosen kernels.

6.1 adaptive hashing

In the indexing stage of the CSH hashing scheme (Section 5.1) the hash functions used do not depend on the images being matched. One exception is the projection bin edge locations, which are determined by sampling the distributions of the projected patches. Unlike those, both the selection of which of the WH kernels to use and the number of bits assigned to each one were determined manually in a way that is proportional to the dispersion of values in each of the projections, averaged over many image-pair instances. These bit/kernel allocations used in CSH are specified in the table of Figure 3.

A different approach would be to make these allocations fully data-driven, possibly allowing better fitting to the instance-specific patch distributions. To this end, we suggest an adaptive bit-allocation scheme, where bits are assigned to projections sequentially in a greedy manner. We first compute the variance of the patch projections on each of the WH kernels. Each projection (kernel) is initially allocated one bin (i.e. assigned with zero bits) and we then iteratively allocate an extra bit to the projection, whose average variance per bin is the largest.

We consider two alternatives to this extension, depending on the way we divide the values (along each projection) into a certain number of bins. In the first option which we term CSH-AP (Adaptive-Percentile) the bin edges are determined in such a way that an equal number of samples fall into each bin. In the second option CSH-AKM (Adaptive-K-Means) the bin edges are computed by a 1-dimensional k-means procedure (into k bins), which by definition minimizes the average within-bin sample variance. In Section 7.2 we compare these two variants against the regular CSH hash-function as well as to other leading hashing schemes found in the literature.

6.2 kNN-CSH

Nearest-Neighbor methods typically support the option of retrieving the k (approximate) nearest-neighbors of a given query point (see, e.g. FLANN [28] and ANN [1]). Such a need may arise in many applications - and a good example in the scenario of patches of images is in the Non-Local-Means [8] denoising algorithm, where a patch is denoised using a large set of patches from its vicinity, and the contribution of a patch exponentially drops with the squared L_2 distance from the source patch. Therefore, the general contribution is typically dominated by the several most similar patches while the contribution of the rest of the patches is negligible. A method for retrieving the k nearest patches would have enabled significantly accelerating the process or alternatively - enabling searching in a wider vicinity of the patch. The Generalized PatchMatch [5] algorithm enables such a kNN search. Note that in this case the goal is to minimize the same error as in the ANN (i.e. 1NN) problem, where and additional average is taken over the k retrieved neighbors.

The extension of CSH to this setup is relatively straightforward. At the beginning of the run, the best-candidate map is initialized (arbitrarily) to contain k random locations per patch. Then, at the search stage, we keep the k best candidates seen so far, by evicting the worst candidate from the list when a good new candidate is found. In addition, in order to enable finding a large variety of NNs (as opposed to a single one), we widen our hash table (at a cost of memory) to keep more patch representatives per table entry.

6.3 bi-directional CSH

It happens (e.g. in methods that use the principal of Bi-directional Similarity [35]) that given a pair of images A and B, it is required to compute NNFs in both directions. Unlike other methods, our method benefits largely (without any needed adaptation) from the large overlap of calculations and memory structures that are common to the two calculations - the projections of patches on WH codes, the hash coding of both image's patches and the full hashing index (depicted in Figure 5). As a result, the bi-directional NNF can be computed at a major saving in runtime, with a negligible increase in memory.

6.4 video CSH

Many patch based methods for video enhancement (e.g. denoising [8], [9], [26], super-resolution [34], inpainting [38]) extensively compute dense matches between 3-dimensional space-time patches. We extend CSH in this direction and experiment on computing space-time NNFs between pairs of videos. CSH extends naturally to incorporate the additional temporal dimension. We repeated the process of manually: (i) selecting the set of 3D WH kernels that are used for L_2 distance approximations. (ii) selecting the set of 3D WH kernels and the distribution of bits for coding these in the hashing scheme. In addition, the left/top (right/bottom) neighborhood used in the creation of candidates of 'Type 2' (Figure 5(b)) needs to be extended to a richer left/top/pre (right/bottom/post) neighborhood, in order to pass NN information between consecutive frames.

6.5 rotation-CSH

An important generalization of NNFs is in the form of enlarging the search space, by considering patch rotation, in addition to translation. The Generalized PatchMatch [5] is an extension of PatchMatch, which enables adding rotation or even rotation and scale to the search space. Their approach was to handle this kind of flexibility by adding new dimensions to the basic 2-dimensional (translations in x and y) search space. This clearly makes it harder to find good matches, based on the random search and propagations scheme. We take a different approach and handle the addition of rotation by normalizing patches with respect to it[4]. This was the approach taken by SIFT [27] and other methods, which make local descriptions invariant to rotation. Specifically, the normalization is done at each location by taking an interpolated patch in the prominent gradient orientation (which is chosen according to a weighted voting scheme, based on orientations and magnitudes of grayscale intensity gradients in the neighborhood).

Taking rotation into consideration required several adaptations in the CSH scheme: (i) The algorithm expects as inputs 2D arrays of rotation-normalized patches (one per pixel location, together with their rotation angles) instead of a standard pair of images. (ii) Since the patches do not form a consistent image, the computation of the WH kernel projections cannot be done using the Grey-Code-Kernel method, but rather, through direct dot-product projection. (iii) When generating candidate matches of 'Type 2' (please refer to Figure 5(b)), if the previous (green arrow) mapping involves a non-trivial amount of rotation, then a left/right (or top/bottom) relationship from one image does not imply the same relationship in the other. Instead of handling this directly (which is not possible for rotations that are not multiples of $90°$), we look in the four possible directions. This produces many more candidates, some of which might turn out to be useful. (iv) The final output of the algorithm is a translation and rotation value per patch. The rotation value is taken to be the difference between the rotations of the two matching normalized patches.

4. Note that unlike the case of rotation, which patches can be normalized to be invariant to, the case of scale needs to be handled by simulating different scales, i.e. matching can be repeated between different scales of the image.

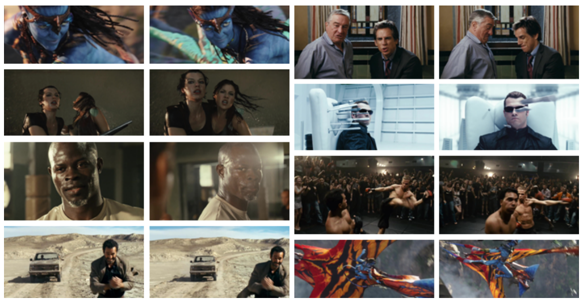


Fig. 7. **The Video Pairs data set** (8 out of the 133 pairs)

7 Experiments

In this section we report on different experiments we performed to validate the algorithm and its different extensions. In Section 7.1, we compare CSH to PatchMatch in terms of the speed-accuracy tradeoff on a large data-set we collected. In Section 7.2 we compare the CSH hash function and its variants to other hashing codes. In Section 7.3 we discuss additional properties of the the CSH mappings, which are shown to be essential for the task of patch-based reconstruction (Section 7.4). The k-NN, video and rotation extensions are demonstrated in Sections 7.5, 7.6 and 7.7.

Data We collected 133 pairs of images, taken from 1080p HD ($\sim$2 megapixel) official movie trailers. Each pair consists of images of the same scene with usually some motion of both camera and subjects in the scene (The images are between 1 and $\sim$30 frames apart in the video). We note that pairs of images with only slight camera and subject motion are not very challenging in the dense patch matching framework and could be handled specifically via registration or optic flow techniques. See Figure 7 for some example image pairs of this database.

Implementation Details Our implementation of CSH is in Matlab, using Mex functions in critical sections. PatchMatch implementation was taken from the PatchMatch website[5]. Both algorithms were run in a single core configuration on a 2.66 GHz machine, with 8 GB of RAM. In terms of memory footprint, aside from the clear need to store the source, target, mapping and error images in memory, CSH requires some extra memory in order to store the hash tables as well as the pre-computed projections of the image patches on the WH kernels. However, instead of constructing the complete index of L hash tables and then searching through them sequentially (as described in algorithm 1), our implementation performs L iterations (cycles) of the index and search steps, using only one table at a time. For further improvement in memory consumption, one could compute the WH projections on the fly, while making a slight change in ordering in the ranking stage. This is possible, since we use them in a sequential order that complies with the Gray Code ordering [6] of these kernels. In all the experiments (except for Section 7.1), both CSH and PatchMatch were run for $L = 5$ iterations.

5. www.cs.princeton.edu/gfx/pubs/Barnes_2010_TGP/index.php

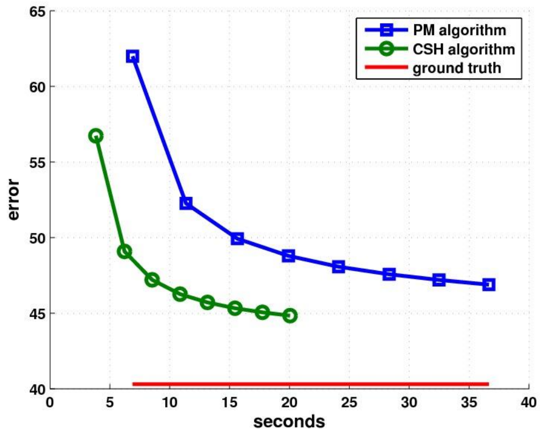


Fig. 8. **Error/Time tradeoffs of PatchMatch and CSH.** Averages are over the 133 image pairs of the data set. Markers on the lines indicate the time it took each algorithm to complete an iteration, and errors are average L_2 distances between patches. Lower error rates (such as those reached by CSH on its third iteration) are reached more than 4 times faster by CSH compared to PatchMatch. Notice that the CSH errors are significantly closer to the ground truth average error (denoted by the solid red line).

7.1 Efficiency

The goal of this experiment is to compare the error-to-time tradeoff of CSH to that of PatchMatch, whose tradeoff was shown [5] to be superior relative to previous methods, in the sense that it reaches reasonable error rates faster.

Our algorithm goes one step forward by being able, on the one hand, to reach reasonable error rates much faster than PatchMatch and on the other - reaching error rates that are out of PatchMatch's reach, as do the (much slower) LSH and KD-Tree algorithms.

We ran both algorithms on the Video Pairs data-set at original resolution using 8×8 patches[6]. The error to time performance of the algorithms was measured by averaging (errors and run-times) over all image pairs. The results are shown in Figure 8. The mapping error (as in [4], [5]) is the average L_2 distance between the matching patches. For comparison, we also computed the exact nearest neighbor match to serve as a ground truth.

In terms of speed, it is clear that our algorithm is much faster than PatchMatch. In order to compare speed, take a certain error rate and compare how long it would take to reach it by each of the algorithms. For instance, the error rate that PatchMatch reaches after 5 iterations (as suggested in [4]) is reached by our algorithm 3 or 4 times faster.

7.2 CSH Hashing

In this experiment we directly measure the efficiency of the CSH hash function, which is compared to several alternative hash functions within the CSH (LSH-based) framework. To do so, for a given image pair we take a random set of source-image query patches as well as the

6. Similar results were observed in different settings, when using lower image resolutions as well as different patch sizes [23]

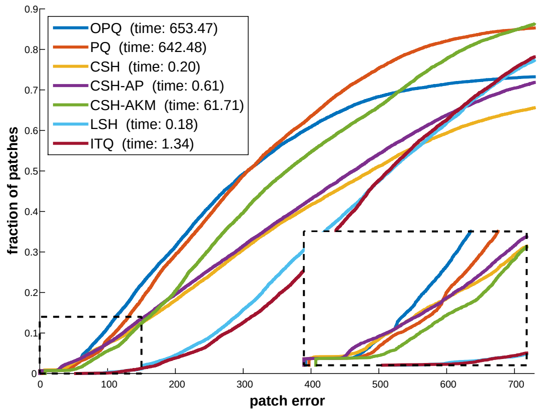


Fig. 9. **Efficiency of different hash functions**. Each curve (one per hash-function) describes the CDF of query-to-bin distances. For a specific error level (x-axis) it shows the fraction of query patches (y-axis), whose average distance from target patches with the same code is below that error level. The 3 CSH alternatives do relatively well in general (the higher the curve the better) and are especially competitive in the lower range of error levels (see enlarged region), where good matches are generated. The K-means based CSH-AKM (green) is generally more accurate than its counterparts (CSH and CSH-AP), however it produces many less low-error matches and is therefore inferior in the CSH framework. The baseline LSH [22] as well as ITQ [22] are clearly less accurate while the state-of-the-art PQ [22] and OPQ [13] are more accurate than the CSH alternatives, but their runtimes (shown in the legend, in seconds, for hashing the patches of a pair of images) are several orders of magnitude slower.

entire set of target-image patches, all which are hashed into 16 bit code words, using a variety of possible hash functions. All hash functions, except those used by CSH, are given the patches in vector form and do not exploit the a-priori image-patch nature of the underlying data.

The first alternative hash-function, is the baseline standard LSH hash function (LSH), in which 16 random projection lines are used, each allocated a single bit. Note that this hash functions is totally agnostic to the input data. Next, several state-of-the art quantization methods which all largely exploit the specific distribution of the input vectors. These are Product Quantization (PQ) [22], Iterative Quantization (ITQ) [16] and the more recent Optimized Product Quantization (OPQ) [13]. We also report results for three variants of the CSH hash function. The basic version (CSH) as well as its two 'adaptive' extensions that were discussed in Section 6.1, where 1D binning is done according to distribution percentiles (CSH-AP) or by K-Means (CSH-AKM).

In general, CSH keeps several 'representative' patches of each hash code in the hash table and therefore an indicative measure for the error of a candidate found through the hash table is the average distance between a query patch and all patches with the same hash code. Such results, collected using 2000 random query 8×8 source-image patches from each of the Vid-Pairs collection, resized to 0.5 MPs, and are shown in Figure 9.

The plots show the distribution of the obtained distances, for the different hash functions. Notice in the legend the average coding times (in seconds) for encoding the entire set of patches of a pair of images. The main observation is that the CSH hash function (light orange) finds large amounts of low-error matching patches (see enlarged area) at a very competitive runtime. We focus the attention to patch matches of lower error, since these are the ones that describe 'real' matches between corresponding patches, which have a high influence on the propagation of good matches. On the other hand, a large fraction of query patches do not have similar patches in the target and minimizing their error is of lower importance. In this sense, considering the K-means based variant CSH-AKM (green) - its curve is generally above that of the original CSH (orange), however, it is lower in the important low-error range. Therefore, it is less efficient in our setup. The other proposed (adaptive) variant CSH-AP (purple) only slightly outperforms the original version and is therefore less attractive due to its runtime overhead. Finally, the state-of-the-art PQ [22] and OPQ [13] product quantization methods are more accurate than the CSH alternatives, however their runtimes are impractical for this dense matching scenario.

7.3 Other Properties

Aside from its good error to time tradeoff, CSH possesses other pleasing properties, which are of high importance (not less than the error rate itself), in the common usages of such dense patch mappings. In this section we will review these properties, in comparison to the PatchMatch mapping and ground-truth (exact) mappings.

7.3.1 Image Energy and Mapping Quality

PatchMatch and CSH differ in the way the quality of a match depends on the energy level of the patch (i.e. how textured is the patch). Generally speaking, PatchMatch copes slightly better with flat areas, while CSH does better in the mid range and going towards textured, edgy patches. This is, again, due to the locality of the PatchMatch search and propagation, which will work well in large homogeneous areas, but will fail in high energy areas where usually nearby patches might only be well matched to patches that are very distant in the target image.

For our experiment we used the same 133 image pairs. For each such pair, we ordered the source image's patches according to their spatial energy (mean of gradient magnitudes) in increasing order and divided them into ten equal sized deciles. For each such decile of patches we calculated the mean error of the patch matches, produced by each of the algorithms. In Figure 10, we plot the difference between the PatchMatch error and the CSH error for each of the deciles. The general trend of the plot is clear and consistent across the range of patch energies. We argue that the distribution of errors produced by CSH is preferable to that of PatchMatch, since it is known that errors along edges and textured areas have a much stronger visual impact compared to inaccuracies in textureless areas. This is the reason that CSH is able to avoid many artifacts along edges

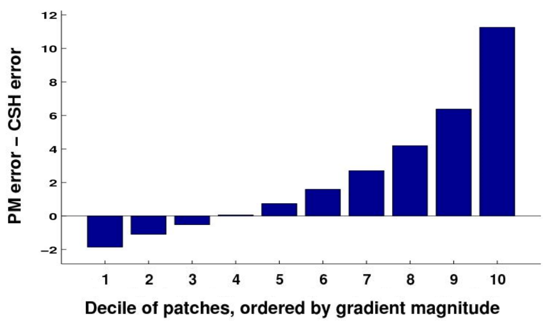


Fig. 10. **Mapping errors ordered by patch energy**. x-axis: Patches of the source image are divided into 10 deciles, according to their energy level (mean gradient magnitude). y-axis: the difference between PatchMatch and CSH mapping errors, averaged over each of the deciles. On the lower end, the first decile represents patches with low energy in the range $[0, 14]$ on which PatchMatch error (mean L2 patch distances) is slightly lower (2 graylevels), while at the tenth decile (high energy in the range $[155, 255]$) - CSH error is significantly lower (over 11 graylevels).

(compared to PatchMatch) when reconstructing a source image from a target image patches using the dense mapping between them (this is shown in section 7.4).

7.3.2 Incoherence of the Mapping

Given a dense patch mapping from image A to image B, we define the *incoherence* of the mapping at each pixel a of A to be the number of *different* pixels in B that a is mapped to under all of the patches that contain it. For instance, incoherence of 1 (the minimum possible) at a pixel, means that all the patches containing it map coherently (by a constant translation). The maximal coherence is the patch size. This definition is illustrated in Figure 11.

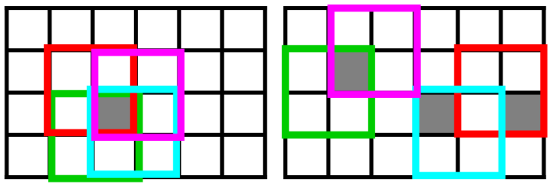


Fig. 11. **Incoherence of a pixel**. In this example patches are 2-by-2. There are 4 patches containing the pixel on the left. Each of these patches is mapped to the patch of the corresponding color on the right. The incoherence of the mapping at the pixel is 3.

The higher incoherence of the CSH mapping (compared to PatchMatch) is due to the different way in which the patches are found. In PatchMatch, the vast majority of final matches are ones that were directly propagated from neighboring patches or randomly found extremely close to them. In CSH, different good quality matches that are spatially spread in the target image have a fair chance to be found by the algorithm. This is especially true for regions that do not appear as a whole in the target image.

Large incoherence of a dense mapping is a crucial property, when it comes to some of the applications that make use of dense patch mappings. This is true for applications, where an image area is reconstructed, pixel by pixel, according to ’votes’ that come from patches in the target image of an ANN mapping. The reason being simply that the incoherence measures the number of votes a pixel gets. Therefore, for mappings of the same error level, regardless of how the votes are integrated into a single decision (e.g. by taking the median or some weighted average) - the precision of the estimate increases with the incoherence. This (negative) correlation between incoherence and reconstruction is shown experimentally in section 7.4. The average incoherence over the entire data-set was found to be 15% higher in CSH compared to PatchMatch.

7.4 Image Reconstruction

The combination of these CSH properties is useful in various image editing and denoising applications. We demonstrate this in the most direct manner, using the reconstruction of a source image A, given a target image B and a dense patch map from A to B. This kind of reconstruction is the main ingredient of the patch based versions of the above mentioned applications. We use the code supplied with PatchMatch to calculate the image reconstruction and its quality. It simply replaces each pixel with the average of the corresponding pixels that it is mapped to by all patches that contain it. This kind of averaging was shown [35] to maximize the (Bi-)Directional Similarity from A to B. For this experiment we used all images from the Video Pairs data-set, resized to 0.4 MP.

	PatchMatch	CSH	Ground Truth
reconst. RMSE	7.62	6.29	5.81

Fig. 12. **Average reconstruction errors** - PatchMatch vs. CSH, relative to using ground truth mapping. Averages are over the 133 image pairs data-set, at 0.4 MP. CSH achieves reconstruction error rates that are only 8% higher than those produced using the ground truth mapping. In comparison, the errors of PatchMatch are more than 30% higher.

We use as a baseline the ground-truth (exact) mapping, which results in the best possible reconstruction under the Bidirectional Similarity framework. The results are summarized by the table in Figure 12. The RMSE error is the square root of the mean (over pixels in all images) of the squared L2 (in RGB) norm between original and reconstructed pixels. It can be seen from the table that the CSH average error is more than 20 percent lower than that of PatchMatch.

Figure 13 clearly shows the correlation we discussed in Section 7.3.2 between mapping incoherence and reconstruction error. A typical reconstruction example[7] is shown in Figure 19, in which the reconstructions produced using PatchMatch and CSH mappings are compared with the reconstruction produced using the ground truth mapping.

7. Please refer to the CSH web page [23] for additional examples.

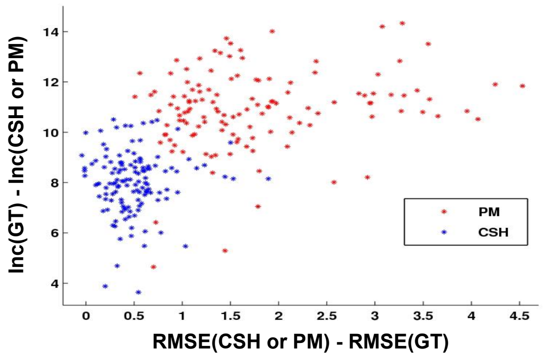


Fig. 13. **Incoherence and Reconstruction Error.** Each point denotes one of the 133 image pairs. The x-axis shows the difference in reconstruction error between each algorithm and the ground truth (GT) mapping. Similarly, the y-axis shows the difference in incoherence between the GT and each of the algorithms. Being close to the origin, means being close to the GT. The two separate clusters emphasize the negative correlation, between incoherence and reconstruction error (see discussion in section 7.3.2).

7.5 CSH for KNN

In this experiment, we run CSH for finding k NNs per image patch (as described in Section 6.2). We run on the entire VidPairs dataset, using four different values of k: 1, 5, 10 and 20. In order to evaluate the error distribution of the computed K nearest-neighbors per patch in each of configurations, we take a random sample of 200 patches from each image. For each such patch we measure the 25th and 75th error percentiles as well as the median (50th percentile) error for each of KNN-CSH, the KNN version of PatchMatch and the ground truth result, which we computed by brute force search (only on the evaluation sample). We report the averages of each of these 3 percentiles over all sample patches from all the images. In all configurations - patches are 8×8 and images are scaled to 1 Mega-Pixel. The error statistics are summarized in Figure 14.

Typically, the CSH errors are much lower than those of Patchmatch and closer to those of the ground truth search. Still, one can notice that the spread of errors in CSH is larger than that of Patchmatch. This can be explained by the use of the hashing scheme. This scheme allows CSH to recover the leading candidates, which are close to the ground-truth solutions. However, the variety of good candidates that can be reached from one patch is limited in the hashing scheme, as opposed to the random search scheme applied by Patchmatch.

7.6 Video Matching

In this experiment we demonstrate the ability of CSH to densely match space-time patches from a dataset of videos (as described in Section 6.4). We use four different video clips (4 shots from the movie trailers, from which the VidPairs data-set was collected), containing 44, 55, 73 and 89 frames, resized to 0.1 mega-pixels. The first, middle and last image of each of the sequences can be seen in Figure 15. In a first experiment, the NNF is computed between the first half and the second half of each clip.

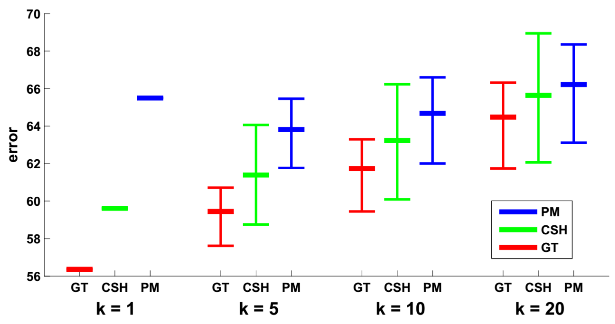


Fig. 14. **KNN CSH error statistics.** We ran the KNN version of CSH and Patchmatch for $L = 5$ iterations on the entire VidPairs dataset for different values of k (1, 5, 10 and 20). The error percentiles of 25, 50 (median) and 75 are shown (average values over a large random sample of patches). The results are similar in fashion to the full evaluation in Figure 8, however the errors are generally higher since the images are smaller (1 MP rather than 2 MP). The CSH errors are typically closer to those of the ground truth (see text for details).

The video frames are scaled to a size of 210×500 and we use $4 \times 4 \times 4$ patches though out. In order to compare against the 'ground-truth' best possible NNF, we picked 100 patches from the first half of each clip, uniformly at random, compute the optimal solution for these (in brute-force manner) and report our results relative to those. The results can be seen in Figure 16.

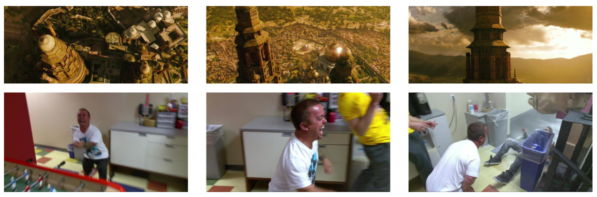


Fig. 15. **Example images from the video clips used for the video matching experiment.** First, middle and last frame from two of the sequences. See further details in text.

Similarly, we experiment on computing a KNN field between a video and itself. This would be a common building block in many block-based methods for video editing applications, which are out of the scope of this

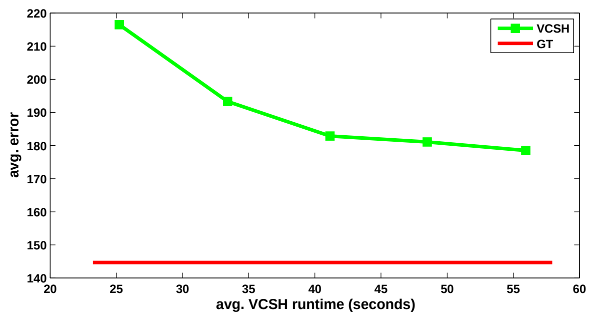


Fig. 16. **1NN VCSH (Video CSH) between two halves of each video clip.** This plot shows, per iteration of VCSH, the average computation time (over the four clips) in seconds. In addition, we report average match errors over a random sample of 100 space-time patches per clip, compared to the average ground-truth error computed on these samples.

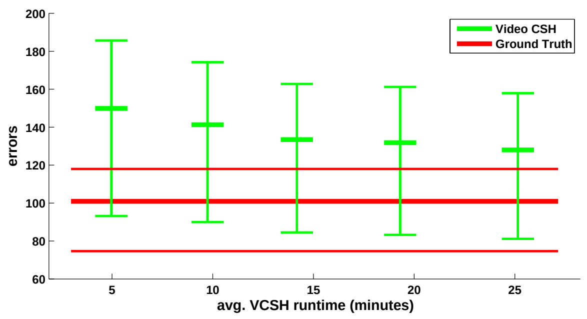


Fig. 17. **KNN VCSH (Video CSH) with K = 25 between each video clip and itself.** Average computation times in minutes (over 4 clips) for 5 iterations of KNN VCSH. In addition, in green are the 25th, 50th (median) and 75th percentiles (over the K=25 NNs) of the match errors over a random sample of 100 space-time patches per clip. These are compared to the respective percentiles of the ground-truth errors, shown in red.

article. Here again, we use the same four scaled video clips and report average and standard deviation of the computed $k = 25$ NNs. These are compared relative to ground-truth errors, which were measured on 100 random patches. The results are summarized in Figure 17.

7.7 CSH with Rotation

In this experiment we run CSH, allowing rotations (as described in Section 6.5) and compare against PatchMatch, which allows including rotation or even rotation and scale. The running was executed on the entire VidPairs dataset, at four different combinations of patch and image dimensions. Figure 18 summarizes the average errors for each of the four configurations, for the different modes of CSH and PatchMatch respectively. As can be seen, in the case of CSH, allowing the additional degree of freedom enables a reduction of the error by around 10%. PatchMatch is unable to benefit from the extra degrees of freedom, since the significant increase in the search space size makes its random search component less efficient.

8 Conclusions

We proposed an algorithm for computing ANN fields termed Coherency Sensitivity Hashing, which follows the concepts of LSH search scheme, but combines image coherency cues, as well as appearance cues in a novel manner. It was shown to be faster than previous methods and more accurate, especially in textured areas. In addition, its high incoherence improved reconstruction results, which are at the basis of many patch based methods. Finally, some natural extensions of the basic search scheme were developed in order to facilitate their usage in many common patch-based algorithms.

Acknowledgments: This work was partially supported by Israel Science Foundation grant 1556/10 and European Community grant PIRG05-GA-2009-248527. We thank Yonatan Hyatt, Guy Shwartz and Efrat Glikstein for their assistance.

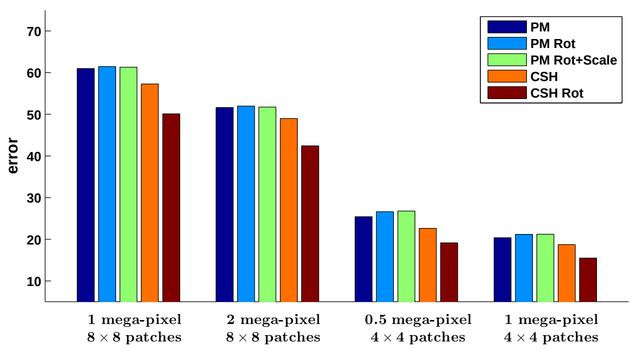


Fig. 18. **CSH with Rotation.** We evaluate the contribution of adding rotation to the CSH scheme. We report average errors, over the entire VidPairs dataset, for CSH with and without rotation and in comparison with regular, rotation and rotation+scale versions of PatchMatch. We report on 4 different setups in terms of image resolution and patch dimensions, as specified in the x-axis. It is consistently evident that the addition of rotation enables reducing the error by more than 10% in the case of CSH, while the versions of PatchMatch using rotation or even scale do not outperform the basic search scheme on this challenging data-set.

References

[1] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45(6):891–923, 1998.
[2] M. Ashikhmin. Synthesizing natural textures. In *Proc. symposium on Interactive 3D graphics*, pages 217–226, 2001.
[3] C. Barnes. *PatchMatch: A Fast Randomized Matching Algorithm with Application to Image and Video*. PhD thesis, Princeton University, 2011.
[4] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *In SIGGRAPH*, 28(3), 2009.
[5] C. Barnes, E. Shechtman, D. B. Goldman, and A. Finkelstein. The generalized PatchMatch correspondence algorithm. In *European Conference on Computer Vision*, 2010.
[6] G. Ben-Artzi, H. Hel-Or, and Y. Hel-Or. The gray-code filter kernels. *In PAMI*, pages 382–393, 2007.
[7] F. Besse, C. Rother, A. Fitzgibbon, and J. Kautz. Pmbp: Patchmatch belief propagation for correspondence field estimation. *International Journal of Computer Vision*, 110(1):2–13, 2014.
[8] A. Buades, B. Coll, and J.-M. Morel. Nonlocal image and movie denoising. *International journal of computer vision*, 76(2):123–139, 2008.
[9] K. Dabov, A. Foi, and K. Egiazarian. Video denoising by sparse 3d transform-domain collaborative filtering. In *Proc. 15th European Signal Processing Conference*, volume 1, page 7, 2007.
[10] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of annual symposium on Computational geometry*, pages 253–262, 2004.
[11] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. *SIGGRAPH*, pages 341–346, 2001.
[12] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *ICCV*, pages 1033–1038, 1999.
[13] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 2946–2953. IEEE, 2013.
[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *International Conference on Very Large Data Bases*, pages 518–529, 1999.
[15] Y. Gong, S. Kumar, H. A. Rowley, and S. Lazebnik. Learning binary codes for high-dimensional data using bilinear projections. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 484–491. IEEE, 2013.
[16] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 817–824. IEEE, 2011.

[17] Y. HaCohen, E. Shechtman, D. B. Goldman, and D. Lischinski. Non-rigid dense correspondence with applications for image enhancement. *ACM Transactions on Graphics (TOG)*, 30(4):70, 2011.
[18] K. He and J. Sun. Computing nearest-neighbor fields via propagation-assisted kd-trees. In *Computer Vision and Pattern Recognition (CVPR)*, pages 111–118. IEEE, 2012.
[19] K. He and J. Sun. Statistics of patch offsets for image completion. In *Computer Vision–ECCV 2012*, pages 16–29. Springer, 2012.
[20] Y. Hel-Or and H. Hel-Or. Real-time pattern matching using projection kernels. *In PAMI*, pages 1430–1445, 2005.
[21] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Symposium on Theory of Computing*, pages 604–613, 1998.
[22] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(1):117–128, 2011.
[23] S. Korman. CSH webpage. www.eng.tau.ac.il/~simonk/CSH.
[24] S. Korman and S. Avidan. Coherency sensitive hashing. In *ICCV*, pages 1607–1614. IEEE, 2011.
[25] V. Kwatra, A. Schdl, I. Essa, G. Turk, and A. Bobick. Graphcut textures: Image and video synthesis using graph cuts. *SIGGRAPH*, 22(3):277–286, 2003.
[26] C. Liu and W. Freeman. A high-quality video denoising algorithm based on reliable motion estimation. *Computer Vision–ECCV 2010*, pages 706–719, 2010.
[27] D. G. Lowe. Object recognition from local scale-invariant features. In *Computer Vision. The Proceedings of the IEEE International Conference on*, volume 2, pages 1150–1157. Ieee, 1999.
[28] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISSAPP*, pages 331–340. INSTICC Press, 2009.
[29] M. Norouzi and D. J. Fleet. Cartesian k-means. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 3017–3024. IEEE, 2013.
[30] A. Oliva and A. Torralba. Building the gist of a scene: The role of global image features in recognition. *Progress in brain research*, 155:23–36, 2006.
[31] I. Olonetsky and S. Avidan. Treecann-kd tree coherence approximate nearest neighbor algorithm. *ECCV*, 2012.
[32] D. Ruderman. Statistics of natural images. *Network: Computation in Neural Systems*, 5(4):517–548, 1994.
[33] R. Salakhutdinov and G. Hinton. Semantic hashing. *RBM*, 500(3):500, 2007.
[34] O. Shahar, A. Faktor, and M. Irani. Space-time super-resolution from a single video. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 3353–3360. IEEE, 2011.
[35] D. Simakov, Y. Caspi, E. Shechtman, and M. Irani. Summarizing visual data using bidirectional similarity. In *CVPR*, pages 1–8. IEEE, 2008.
[36] X. Tong, J. Zhang, L. Liu, X. Wang, B. Guo, and H. Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. *ACM Trans. on Graphics*, 21(3):665–672, 2002.
[37] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH*, 2000.
[38] Y. Wexler, E. Shechtman, and M. Irani. Space-time completion of video. *PAMI*, 29:463–476, 2007.
[39] V. Zolotarev. *One-dimensional stable distributions*. American Mathematical Society, 1986.

Simon Korman is currently a PhD student at the school of electrical engineering, Tel-Aviv University, Israel. He received a B.Sc. degree in Computer Science and Mathematics from the Hebrew University, Israel, in 2001, and then received an M.Sc. degree in Computer Science and Mathematics from the Weizmann Institute, Israel, in 2005. He also spent the summer of 2012 as an intern in Microsoft Research, Redmond. His research interests include image processing, computer vision, shape analysis and pattern recognition.

Shai Avidan received the Ph.D. degree from the School of Computer Science, Hebrew University, Jerusalem, Israel, in 1999. Currently, he is an Associate Professor at the Faculty of Engineering, Tel-Aviv University, Israel. In between he worked for Adobe, Mitsubishi Electric Research Labs (MERL) and Microsoft Research. He published extensively in the fields of object tracking in video and 3-D object modeling from images. He is also interested in Internet vision applications such as privacy preserving image analysis, distributed algorithms for image analysis, and image retargeting.

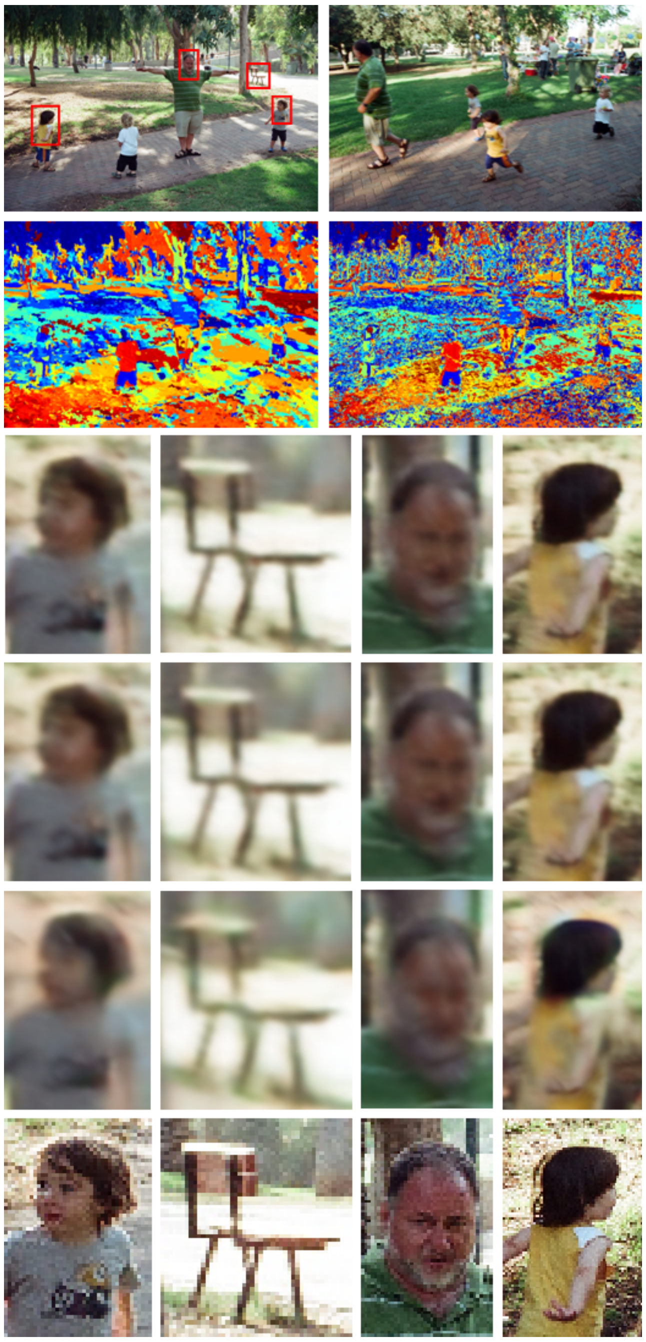


Fig. 19. **Reconstruction Example.** We visually compare reconstruction results using PatchMatch, CSH and Ground truth mappings on a typical pair of 0.5 MP images. **Row 1**: The dense mappings are computed from A (left) to B (right). **Row 2**: x-coordinates of PatchMatch mapping (left) and CSH mapping (right). Blue/red areas in A are mapped to the left/right side of B. These images illustrate the lower coherency of the CSH mapping compared to that of PatchMatch. As discussed in the text - this enables better reconstruction. **Rows 3-5**: Enlarged areas from reconstructed image A, using ground-truth, CSH and PatchMatch mappings (in this order). In this example, reconstruction RMS errors are: 19.4 (ground-truth), 20.1 (CSH) and 22.0 (PatchMatch). Visually, the PatchMatch reconstruction is less accurate (especially around edges), introducing blur and color distortion. **Row 6**: The original (image A) areas, shown for reference.