Fast Edge-Preserving PatchMatch for Large Displacement Optical Flow

Linchao Bao, Student Member, IEEE, Qingxiong Yang*, Member, IEEE, and Hailin Jin, Member, IEEE

Abstract—The speed of optical flow algorithm is crucial for many video editing tasks such as slow motion synthesis, selection propagation, tone adjustment propagation, etc. Variational coarse-to-fine optical flow algorithms can generally produce high-quality results but cannot fulfill the speed requirement of many practical applications. Besides, large motions in real-world videos also pose a difficult problem to coarse-to-fine variational approaches. We in this paper present a fast optical flow algorithm that can handle large displacement motions. Our algorithm is inspired by recent successes of local methods in visual correspondence searching as well as approximate nearest neighbor field algorithms. The main novelty is a fast randomized edge-preserving approximate nearest neighbor field algorithm which propagates self-similarity patterns in addition to offsets. Experimental results on public optical flow benchmarks show that our method is significantly faster than state-of-the-art methods without compromising on quality, especially when scenes contain large motions. Finally, we show some demo applications by applying our technique into real-world video editing tasks.

I. INTRODUCTION

Optical flow estimation is one of the most fundamental problems in Computer Vision. Since the seminal work of Horn-Schunck global model [11] and Lucas-Kanade local model [2], there have been tremendous progresses in this area. We have algorithms that can handle challenging issues such as occlusions, motion discontinuities, textureless regions, etc. However, there are still outstanding problems in existing optical flow methods, such as large displacement motions and motion blur. This paper addresses the issue of large displacement motions. In particular, we are interested in fast optical flow algorithms as speed is crucial for practical applications.

Large displacement motions have been an issue in optical flow estimation since the beginning. The basic formulation of optical flow is based on a differential form of the brightness constancy equation which is invalid for motions larger than the support of the differential operators. In order to handle larger motions, traditional methods resort to the multi-scale coarse-to-fine framework. However, the coarse-to-fine framework suffers from an intrinsic limitation that it fails for fine scale image structures whose motions are larger than their size. Recently, there are several algorithms proposed to overcome this intrinsic limitation by going beyond the basic differential formulation and incorporating additional correspondence information. For instance, one can directly search for pixel correspondence [3]. But the complexity of the search step scales quadratically with respect to the size of the motion. Robust keypoints are one reliable source of correspondence information that can be matched efficiently across entire images but are only available at sparse image locations. Recently, an algorithm called deep-matching [4] is proposed to produce dense correspondence field efficiently, but its huge memory consumption prevents itself from practical applications. Besides, in order to obtain a dense flow field, one needs a global optimization step which is typically computationally expensive [5], [6].

In this paper, we propose to use approximate nearest neighbor field (NNF) for large displacement optical flow estimation. NNF is a correspondence field indicating pairs of image patches from two images which are closest in terms of some patch distance. There is no limitation on the relative distance between a pair of closest patches which makes NNF a good source of information for handling large displacement motions. Moreover, although exact NNF is expensive to compute, there exist efficient approximate algorithms [7], [8], [9].

In order to obtain optical flow using approximate NNFs, we need to address two fundamental problems. First, there is no spatial smoothness in a NNF which means neighboring patches in one image can have arbitrary matching patches in the other image. This problem is more pronounced in homogeneous regions where matching is ambiguous. Thus most approximate NNF algorithms (such as CSH [8] and KD-Tree [9] based algorithms) will produce messy fields and are not suitable for optical flow estimation. Second, occlusions are not respected in NNF computation, i.e., one will get matching patches in occluded regions even though they are meaningless. The second problem can be resolved by explicitly performing consistency check between forward and backward flow. To address the first problem, one may attempt to use global optimization to incorporate motion candidates from a NNF into an optical flow estimation [10]. However, doing so may lead to a computationally expensive algorithm which has limited practical applicability. Instead, motivated by recent successes of local methods in stereo matching and optical flow [11], [12], [13] where it is shown that carefully crafted local methods can reach quality on par with global ones, we address the problem by increasing the local matching support (patch size). But increasing patch size leads to two new problems...
which are motion boundary preservation and algorithm speed. We address the former problem by introducing a novel edge-preserving version of PatchMatch [12] and the latter one by developing a fast approximate algorithm.

This paper extends its conference version [14] with the following major differences:

1. We provide more explanation of our method (see Section II-A), which is omitted in the conference version due to page limit.
2. We reveal more details about our implementation and experimental results in Section III-B and add the details of the performance of our method on KITTI benchmark.
3. We apply our method to several real-world applications and show some examples in Section IV.

A. Related work

It is beyond the scope of this paper to review the entire literature on optical flow. Instead, we will only discuss the closely related papers. In particular, we will focus on the work that addresses large displacement motions. The classical coarse-to-fine framework for large displacement motions that is used by most optical flow algorithms was proposed in [15], [16] and refined in [17]. It generally works well for relatively large objects but performs poorly on fine scale image structures which may disappear in coarse scales. This is an intrinsic limitation of the coarse-to-fine framework. To overcome this limitation, Steinbruecker et al. [9] proposed to incorporate correspondence searches in a framework that avoids warping and linearization. However, the search part is exhaustive for every pixel in the image which makes the algorithm potentially slow for large search ranges. Instead of an exhaustive search at every pixel, the LDOF framework [5] is to only consider robust keypoints which serve as constraints in an energy-based formulation. Because keypoints can be matched across entire images, the algorithm does not suffer from the search range problem. To further improve the reliability of keypoint matching, the MDP-Flow [6] incorporated a discrete optimization step before diving into the variational optical flow solver.

Regarding NNFs, PatchMatch [7] was a seminal work and generated a lot of interests recently because of its computational efficiency and ability to match patches across large distance. But most algorithms in this area are proposed for the NNF problem in terms of reconstruction error [8], [9], which is different from the dense correspondence problem. Exceptionally, the work [13] applied PatchMatch to stereo matching for computing aggregation with slanted support windows, but they did not address the computational efficiency after adopting a weighting scheme on the support windows. A recent work employing NNF for optical flow estimation is [10], which computes an initial noisy but dense matching which is cleaned up through motion segmentation.

Our algorithm is closely related to the local methods in stereo matching and optical flow. Local methods have a long history in stereo matching. They used to be known as fast but less accurate compared to globally optimized methods. But [13] showed that a good local method can perform equally well. Rhemann et al. [11] successfully applied this principle to optical flow and obtained an algorithm that ranks high on the Middlebury flow benchmark. The SimpleFlow [12] followed the same direction but towards a less accurate yet faster solution. The PatchMatch Filter [19] adapted the PatchMatch algorithm onto superpixels and employed the algorithm from [11] to refine the matching correspondence between superpixels.

B. Contributions

The main contribution of this work is a fast local optical flow algorithm that can handle large displacement motions. Our method is local, i.e., it does not involve optimization over the entire image and therefore fast. On the other hand, our method does not sacrifice on quality. We compare our method against existing ones on MPI Sintel, KITTI, and Middlebury benchmarks. Our ability to handle large displacement motions is clearly demonstrated by the top performance on the MPI Sintel benchmark. In terms of quality, our method outperforms all other fast methods without compromising on the speed. In fact, the quality of our method is on par with that of global ones but the speed is significantly faster.

Our main technical novelty is a fast randomized edge-preserving approximate nearest neighbor field algorithm. The key insight is that in addition to similar offsets, neighboring patches have similar self-similarity patterns. Therefore, we can propagate self-similarity patterns in a way similar to propagating offsets as done in [7]. This significantly reduces the computational complexity. We hope this idea to inspire other work in generalizing [7] to other applications.

II. OUR APPROACH

Our method follows the traditional local correspondence searching framework [20] which consists of 4 steps:

1) matching cost computation,
2) cost aggregation,
3) correspondence selection, and
4) refinement.

It is shown that the framework can produce high-quality optical flow [11], but its computational complexity is linear in search range.

While reducing the correspondence search range may be a potential solution, we in this paper address this problem from another point of view. We notice that, if we use squared error as the matching cost and use box filtering to perform the cost aggregation, then steps (1) to (3) are actually equivalent to searching the nearest neighbors for image patches using the patch Euclidean distance, which is known to have fast approximate algorithms that are independent of search range, such as PatchMatch [7]. However, a direct use of PatchMatch to estimate optical flow can handle large displacement motions but tend to introduce visible errors around motion discontinuities as shown in Fig. [15]. To overcome the problems, we propose a new edge-preserving version of PatchMatch (Sec. II-A) and a corresponding fast algorithm (Sec. II-B).
The techniques used in [11] for the refinement step (i.e., consistency check and weighted median filtering [21], [22]) are also employed in this paper except that we suggest to produce subpixel accuracy with a more efficient technique – paraboloid fitting, which is a 2D extension from the 1D parabola fitting – a commonly adopted technique in stereo matching [23]. Details are presented in Sec. II-C and II-D.

A. Edge-Preserving PatchMatch

The main idea of original PatchMatch [7] is to initialize a random correspondence field and then iteratively propagate good guesses among neighboring pixels. In order to avoid trapping into local minima, several random guesses are additionally tried for each pixel during the propagation. The matching cost between two patches is originally defined as

$$d(a, b) = \sum_{\Delta (\Delta x, \Delta y); |\Delta x| \leq r, |\Delta y| \leq r} ||I^A(a + \Delta) - I^B(b + \Delta)||^2,$$

where $I^A$ and $I^B$ denote the CIELab color appearances of image $A$ and $B$, respectively.

In order to make the NNF preserve details of input image, we add bilateral weights [13] into the matching cost calculation. Moreover, similar to the data term employed in variational optical flow estimation [24], [25], we replace the $L_2$ norm in the above formulation with a robust loss function (such as the negative Gauss function or the Lorentzian function [24]) to reject outliers. Furthermore, in addition to color cue, we can add more cues that can better deal with repetitive patterns and textureless regions, e.g., image gradient or the census transform [26]. Specifically, the matching cost in our approach is defined as follows,

$$d(a, b) = \frac{1}{W} \sum_{\Delta (\Delta x, \Delta y); |\Delta x| \leq r, |\Delta y| \leq r} w(a, b, \Delta)C(a, b, \Delta),$$

where $w(\cdot)$ is the bilateral weighting function, $W$ is the normalization factor (sum of all the weight $w(\cdot)$), and $C(\cdot)$ is the robust cost between two pixels (suppose $K$ cues are involved in the cost calculation):

$$w(a, b, \Delta) = \exp(-\frac{||I^A(a + \Delta) - I^A(a)||^2}{\sigma_s^2}) \exp(-\frac{||I^B(b + \Delta) - I^B(b)||^2}{\sigma_r^2}) \exp(-\frac{||\Delta||^2}{\sigma_s^2}),$$

$$C(a, b, \Delta) = \sum_{i=1}^{K} \rho_i(C_i(a + \Delta) - C_i(b + \Delta)),$$

where $\sigma_s$ and $\sigma_r$ are controlling spatial and range influences, respectively (typically, we set $\sigma_s = 0.5r$ ($r$ is patch radius) and $\sigma_r = 0.1$. The cost contributed by each cue $C_i$ is controlled by a robust loss function $\rho_i(\cdot)$ for rejecting outliers and balancing between different cues (for simplicity, we use the same loss function for all the cues used in our experiments, see Sec. III).

Fig. 1 shows a comparison of the NNF results produced by the original PatchMatch and the proposed edge-preserving PatchMatch. The details in input image can be much better preserved in NNF when using our edge-preserving version. Note that in order to perform flow refinement (in particular, the consistency check [11]), we need to compute the NNFs between two images in both directions. Thus we use the symmetric bilateral weight in Eq. (3), so that during the PatchMatch we can symmetrically update both NNFs after calculating one matching cost.

B. Approximate Algorithm

While PatchMatch can effectively reduce computational complexity in terms of search range, its complexity still depends on patch size. In order to produce high-quality flow fields, however, a large patch size is usually preferred for eliminating matching ambiguities (note that the edge-preserving feature plays an important role for maintaining flow accuracy when increasing patch size). In this section, we propose an algorithm that utilizes a self-similarity propagation scheme and a hierarchical matching scheme to approximate the exact edge-preserving PatchMatch.

1) Self-Similarity Propagation: We notice that, due to the range kernel employed in the matching cost computation (Eq. (3)), the major portion of the matching cost is contributed by pixels that are similar to the center pixel. This suggests a natural way to accelerate the matching cost computation which is to simply ignore dissimilar pixels to center pixel. To be more specific, for each pixel, we precompute the $n \ (n \ll M = (2r + 1)^2$ most similar pixels within its neighborhood, store their positions and use them to compute the cost.

(a) Input (first frame)  
(b) Original PatchMatch  
(c) Ours  
(d) Ours (refined)  

Fig. 1. PatchMatch results (cropped) on the “Army” dataset from Middlebury benchmark [24]. The proposed edge-preserving PatchMatch can preserve details in the NNF results. Note that the outliers in NNF result can be easily removed by refinement.
Before applying the idea to optical flow estimation, we first performed experiments on joint bilateral filtering [28, 29] to validate it, since the matching cost computation in Eq. (2) is essentially performing a brute-force joint bilateral filtering on cost cues (using input image as guidance). The experiment is conducted as follows: we compute the output of each pixel only by the \( n \) most similar pixels to it according to the guidance image. It is shown in the experiment that \( n = 50 \) to 100 can commonly produce high-quality approximate results for large patch size like \( 35 \times 35 \), which is employed in our optical flow algorithm. Fig. 2 shows one example of the experimental results.

When it comes to optical flow estimation, we performed experiments on the Middlebury training datasets [27] to validate this idea. For each pixel, the neighboring \( n \) most similar pixels are used for computing patch matching cost. Table 1 shows the optical flow accuracy and the corresponding runtime on Middlebury training datasets when \( n \) is with different value

### Algorithm 1 Self-Similarity Propagation Algorithm

**Input:** image \( A \), patch radius \( r \), number of selected pixels \( n \).

**Output:** a self-similarity vector \( S(x, y) \) for each pixel \((x, y)\) containing locations of \( n \) selected pixels.

/* Initialization */

for each pixel \((x, y)\) in \( A \) do

1. randomize a vector \( S(x, y) \) containing the locations of \( n \) neighboring pixels within the patch centered at \((x, y)\);
2. sort pixels in \( S(x, y) \) according to their Euclidean distance to pixel \((x, y)\);

/* This is essentially a sorting over \( 3n \) pixels in the three vectors and select the \( n \) best pixels according to their color distances to pixel \((x, y)\). However, when pixel color at \((x-1, y)\) or \((x, y-1)\) is close to \((x, y)\), the merge process can be done faster with a process similar to a merge sorting since the three vectors can be treated as sorted vectors. */

/* Propagation */

for each pixel \((x, y)\) in \( A \) (scan from top-left to bottom-right) do

1. merge vector \( S(x-1, y) \) and \( S(x, y-1) \) into \( S(x, y) \) according to the pixels’ color similarity to pixel \((x, y)\);

2. for pixel in \( S(x, y) \) that falls outside the patch window, randomize a new location within the patch and re-sort the vector.

end for

/* End */

end for

### Table 1

<table>
<thead>
<tr>
<th>Average Error</th>
<th>EPE</th>
<th>AAE</th>
<th>CPU Timing (sec)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch ((35 \times 35))</td>
<td>0.31</td>
<td>3.35</td>
<td>97.8</td>
</tr>
<tr>
<td>(n = 200)</td>
<td>0.32</td>
<td>3.45</td>
<td>19.1</td>
</tr>
<tr>
<td>(n = 100)</td>
<td>0.33</td>
<td>3.50</td>
<td>10.2</td>
</tr>
<tr>
<td>(n = 50)</td>
<td>0.33</td>
<td>3.56</td>
<td>5.4</td>
</tr>
<tr>
<td>(n = 30)</td>
<td>0.49</td>
<td>5.08</td>
<td>3.5</td>
</tr>
<tr>
<td>(n = 10)</td>
<td>0.91</td>
<td>9.94</td>
<td>1.6</td>
</tr>
</tbody>
</table>

*The time is recorded for running bidirection PatchMatch algorithm (computing two NNPs) on \( 640 \times 480 \) images. Accuracy is evaluated after refinement. Note that the CostFilter takes about 430 seconds on the same CPU to produce bidirectional optical flow with search range \( 61 \times 61 \).
vector into its own vector (according to the stored pixels’ similarity to current pixel); reversely scan and merge. Since we do not intend to select exactly the \( n \) most similar pixels to the center pixel for each patch, the algorithm does not need to interleave additional random guesses during propagation or iterate more. The pseudo-code is in Algorithm 1. The approximate algorithm only needs \( O(n \log n) \) computation for each pixel (the sorting in initialization step and merging in propagation step), which is independent of patch size. Thanks to the propagation between adjacent pixels, the algorithm can produce reasonably good approximate results in a much faster speed (for \( 35 \times 35 \) patch size with \( n = 50 \), it takes about 1.8 seconds and is about \( 6 \times \) faster than the exact selection, the speedup factor grows larger as the patch size becomes larger). For the optical flow accuracy, we do not experience degraded accuracy on the Middlebury training datasets than using the exact selected \( n \) pixels as reported in Table I. Fig. 3 shows an example of the visual results of the selected pixels by our algorithm. More results are provided in Sec. III.

2) Hierarchical Matching: When input image is large, performing PatchMatch on all pixels is a waste of computation. We employ a hierarchical matching scheme to further accelerate the algorithm. Given a pair of input frames, we first downsample the images to a certain lower resolution (for a balance between speed and accuracy, we typically downsample input images twice with a factor 0.5 at each dimension), then we perform the above algorithm to compute the NNF on the downsampled images. After obtaining the NNF on lower resolution, we perform joint bilateral upsampling \( \Delta \) to get a coarse NNF on higher resolution. Then we perform a \( 3 \times 3 \) local patch matching to refine the coarse NNF on the higher resolution images. The pipeline is repeated until we finally get the NNF on the original resolution.

The hierarchical scheme is somewhat similar to that was used in SimpleFlow \cite{12}. However, there are two key differences between our approach and theirs: first, since our edge-preserving PatchMatch does not have restriction on search range, we do not downsample the original frames to very low resolutions and hence it is able to handle large displacements of thin structures (if they still exist in the downsampled resolution). This will also reduce large error accumulation when propagating NNF estimate from much lower resolution to higher resolution. Second, thanks to the edge-preserving ability, the coarser NNF is usually accurate enough and we only need to perform local search within a \( 3 \times 3 \) neighborhood when refining the NNF on higher resolution. This can largely reduce the computation cost, thus our approach is much faster than SimpleFlow (see Sec. III).

Notice that although the hierarchical matching scheme here is similar to the traditional coarse-to-fine framework, the early-cutting pyramid employed in our method is essential to make our method effective for handling large displacement (see Sec. III for the experimental validation for handling large displacement motions).

C. Handling Occlusions and Outliers

After computing bidirectional NNFs (at each resolution) between two images, we explicitly perform forward-backward consistency check \( \Omega \) between the two NNFs to detect occluded regions. Inconsistent mapping pixel is then fixed by nearby pixels according to their bilateral weights. Even so, there will still be some incorrect mapping pixels that cannot be detected by the consistency check, which we treat as **outliers**. A weighted median filtering \( \Delta \) is thus performed on the flow fields to remove the outliers (filtering is performed on all pixels). A second pass consistency check and fixing is then performed to make sure the filtering does not introduce inconsistency. Note that the consistency check and fixing is usually very fast, the computational overhead in this step is mainly the weighted median filtering performed on all pixels.

When occluded region is large, a dedicated hole-filling step is needed in order to fill flow values into such region (e.g., in our implementation a simple scanline-based algorithm is used). Notice that in practical applications, occluded region is not necessarily to be filled with flow values. On the contrary, it is actually a reliable way to detect occluded regions and choose specific handling algorithm with respect to different applications (e.g., the application in Sec. IV-A).

D. Subpixel Refinement

Suppose the discrete correspondence for each pixel \( a \) in image \( A \) is \( \mathcal{NN}_{A \rightarrow B}(a) = b \), and the patch centered at pixel \( a \) is denoted by \( \Omega_a \). We then compute the matching costs between patch \( \Omega_a \) and \( m \) different patches around patch \( \Omega_b \) (see Fig. 4a), respectively, which is denoted as \( \mathcal{D} = \{d_1, d_2, ..., d_m\} \). Note that when computing the matching cost, the fast algorithm in previous section still applies. Assume the cost follows a paraboloid surface on the 2D image grid:

\[
d = f(x, y) = \theta \cdot [x^2, y^2, xy, x, y, 1]^T,
\]

where \( \theta = [\theta_1, \theta_2, ..., \theta_6] \) are the unknowns. Substituting the \( m \) \( (m \geq 6, \text{typically} 25 \text{in our experiments}) \) known points into the equation, we can solve the linear system and figure out the unknowns. Then the \( b^*(x^*, y^*) \) associated with the minimum cost can be computed as follows (by taking derivatives and setting them to zero),

\[
x^* = \frac{2\theta_2 \theta_4 - \theta_3 \theta_5}{\theta_2^2 - 4\theta_1 \theta_2}, \quad y^* = \frac{2\theta_1 \theta_5 - \theta_3 \theta_4}{\theta_2^2 - 4\theta_1 \theta_2},
\]

which is the location of \( a \)’s correspondence with subpixel accuracy. Note that the linear system to be solved is very small,
in practice if we multiply a transposed matrix on both sides, the linear system will have a constant size of $6 \times 6$, no matter how many points are involved (the value of $m$).

To further increase the subpixel accuracy, we compute matching cost for the $m$ points on upsampled images instead of the original images (we obtain upsampled image using bicubic interpolation with an upsampling factor of 2 along each dimension in all our experiments). This does not increase the computational overhead since we only need to compute matching cost for all pixels on the original resolution. The main difference is that the $m$ points around pixel $b$ are now already with subpixel offsets to $b$ (see Fig. 4b). Fig. 5 shows the improvement of this strategy.

Finally, an edge-preserving filtering with small parameters (e.g., bilateral filtering [32], [33] with $\sigma_s = 2, \sigma_r = 0.01$ in our experiments) is performed on the flow fields to smooth out small outliers that might be introduced in this step.

**TABLE II**

<table>
<thead>
<tr>
<th>Clean pass</th>
<th>EPE all</th>
<th>EPE $\leq 40$</th>
<th>Runtime (sec)</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepFlow</td>
<td>5.377</td>
<td>33.701</td>
<td>17</td>
<td>CPU</td>
</tr>
<tr>
<td>MDP-Flow2</td>
<td>5.837</td>
<td>39.459</td>
<td>547</td>
<td>CPU</td>
</tr>
<tr>
<td>Ours</td>
<td>6.494</td>
<td>39.152</td>
<td>0.25</td>
<td>GTX 780</td>
</tr>
<tr>
<td>S2D-Match</td>
<td>6.510</td>
<td>44.187</td>
<td>1920</td>
<td>CPU</td>
</tr>
<tr>
<td>Classic+nl</td>
<td>6.731</td>
<td>45.290</td>
<td>888</td>
<td>CPU</td>
</tr>
<tr>
<td>FC-2Layers</td>
<td>6.781</td>
<td>45.962</td>
<td>4525</td>
<td>CPU</td>
</tr>
<tr>
<td>LDOF</td>
<td>7.563</td>
<td>51.696</td>
<td>2.7</td>
<td>GTX 580</td>
</tr>
<tr>
<td>Classic++</td>
<td>7.961</td>
<td>57.374</td>
<td>888</td>
<td>CPU</td>
</tr>
<tr>
<td>Horn-schunck</td>
<td>8.721</td>
<td>60.645</td>
<td>510</td>
<td>CPU</td>
</tr>
<tr>
<td>Classic+nlp</td>
<td>8.739</td>
<td>58.243</td>
<td>156</td>
<td>CPU</td>
</tr>
<tr>
<td>Cla+nl-fast</td>
<td>9.129</td>
<td>66.935</td>
<td>174</td>
<td>CPU</td>
</tr>
<tr>
<td>SimpleFlow</td>
<td>12.617</td>
<td>81.786</td>
<td>2.9</td>
<td>GTX 285</td>
</tr>
<tr>
<td>A.Huber-L1</td>
<td>12.642</td>
<td>77.835</td>
<td>3.2</td>
<td>GTX 280</td>
</tr>
</tbody>
</table>

**Final pass**

<table>
<thead>
<tr>
<th>Clean pass</th>
<th>EPE all</th>
<th>EPE $\leq 40$</th>
<th>Runtime (sec)</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepFlow</td>
<td>7.212</td>
<td>44.118</td>
<td>17</td>
<td>CPU</td>
</tr>
<tr>
<td>S2D-Match</td>
<td>7.872</td>
<td>48.782</td>
<td>1920</td>
<td>CPU</td>
</tr>
<tr>
<td>FC-2Layers</td>
<td>8.137</td>
<td>51.349</td>
<td>4525</td>
<td>CPU</td>
</tr>
<tr>
<td>Classic+nl</td>
<td>8.291</td>
<td>51.162</td>
<td>888</td>
<td>CPU</td>
</tr>
<tr>
<td>Ours</td>
<td>8.377</td>
<td>49.083</td>
<td>0.25</td>
<td>GTX 780</td>
</tr>
<tr>
<td>MDP-Flow2</td>
<td>8.445</td>
<td>50.907</td>
<td>547</td>
<td>CPU</td>
</tr>
<tr>
<td>LDOF</td>
<td>9.116</td>
<td>57.296</td>
<td>2.7</td>
<td>GTX 580</td>
</tr>
<tr>
<td>Classic++</td>
<td>9.153</td>
<td>60.291</td>
<td>888</td>
<td>CPU</td>
</tr>
<tr>
<td>Horn-schunck</td>
<td>9.610</td>
<td>58.274</td>
<td>156</td>
<td>CPU</td>
</tr>
<tr>
<td>Classic+nlp</td>
<td>9.959</td>
<td>64.135</td>
<td>510</td>
<td>CPU</td>
</tr>
<tr>
<td>Cla+nl-fast</td>
<td>10.088</td>
<td>67.801</td>
<td>174</td>
<td>CPU</td>
</tr>
<tr>
<td>A.Huber-L1</td>
<td>11.927</td>
<td>74.796</td>
<td>3.2</td>
<td>GTX 280</td>
</tr>
<tr>
<td>SimpleFlow</td>
<td>13.364</td>
<td>81.350</td>
<td>2.9</td>
<td>GTX 285</td>
</tr>
</tbody>
</table>

†: The column “EPE $\leq 40^+$ means the average endpoint error over regions with flow velocities larger than 40 pixels per frame.
‡: The runtime are reproduced from either the original papers or the other benchmark websites for $1024 \times 436$ sized images. Note that due to large memory consumption, DeepFlow [4] is difficult to be implemented on GPU.

**III. EXPERIMENTAL RESULTS**

In this section, we present our experimental results on three public optical flow benchmarks – the Middlebury benchmark [27], the KITTI benchmark [38], and the MPI Sintel benchmark [39]. Note that the Middlebury benchmark only contains...
A. Results on MPI Sintel Benchmark

The MPI Sintel benchmark is a challenging optical flow evaluation benchmark, especially due to the complex elements involved, e.g., large motions, specular reflections, motion blur, defocus blur, and atmospheric effects. The evaluation is performed on two kinds of rendering frames, namely clean pass and final pass, each containing 12 sequences with over 500 frames in total. Table II shows the performance of our method on this benchmark (complete table is available online). Our method are among the top performers but with much faster speed than the competitors. Note that if we only consider regions containing large motions (see column “EPE s40” in Table II), our method ranks even higher. Fig. 6 shows two examples of our results on the training data.

One observation is that our method performs worse on the final pass than on the clean pass. Note that the final pass is rendered with motion blur, defocus blur and atmospheric effects while the clean pass are not. By comparing between the results on the two passes (see Fig. 7), we find that our method is mainly degraded on 3 (out of 12) sequences, namely “ambush_1”, “ambush_3”, and “mountain_2,” when moving from clean pass to final pass. In fact, it turns out motion blur and defocus blur do not affect the quality of the results too much, since adjacent frames are usually blurred similarly. This is also usually true for real-world videos, except when the observed object dramatically changes speed or the camera changes focus. The real reason why the results are degraded on the 3 sequences is actually because of the synthetic atmospheric effects, in particular, the heterogeneous smoke (Fig. 7b) and heavy fog (Figs. 7d). These two kinds of effects seriously disturb image local variances (while this is obvious for smoke, the synthetic fog actually introduces very subtle textures, which can be observed on the detail enhanced input images shown in Fig. 8), and this will cause problems at textureless regions for local method since matching cues...
might be locally dominated by the subtle textures introduced. See Sec. III-D for more discussion.

B. Results on KITTI Benchmark

The KITTI optical flow benchmark contains 194 pairs of grayscale frames (test dataset), which are obtained with a wide-view camera fixed on a moving vehicle. Thus most of the scenes in the dataset are perspective views of streets and the scene motions are caused by camera movements along streets. In this case, the frontal-parallel-approximation of PatchMatch often fails due to the slanted planes (e.g., the road ahead of camera) in the scenes. Our method without plane fitting scheme performs not well on this benchmark (see “Ours (w/o PF)” in Table III).

In order to adjust our method to better handle such kind of scenes, we introduce the randomized plane fitting scheme into our method \(18\). The idea is that, during the patch matching, the shape of the patch is adjusted to fit the optimal plane orientation. Since the optimal plane is unknown, it is parameterized with three unknown parameters for each pixel, which is initialized with random guess and propagated during PatchMatch (just like the unknown flow itself) \(18\). The improvement of this scheme is particularly effective for KITTI benchmark (see the entry “Ours” in Table III). Fig. 10 shows a visual example of the improvement.

Notice that since most of the flow in KITTI benchmark are caused by camera movement and tend to be smooth with few motion boundaries, traditional coarse-to-fine methods (e.g., “Classic++” \(44\)) actually perform better than some other more advanced methods (including ours). However, if one is willing to compromise a little on quality for the sake of speed, our method can provide a good choice in this case.

C. Results on Middlebury Benchmark

The evaluation on Middlebury Benchmark is performed on 12 pairs of frames, most of which contain only small displacement motions. Since a matching process is not necessarily needed in the context of small displacements, our method is actually not suitable for this benchmark. Table IV shows the performance of our method on the Middlebury benchmark (complete table is available online). Note that since the evaluation dataset is very small, methods submitted to the benchmark tend to be overfitted (a small difference in EPE can lead to a huge difference in ranking). Our algorithm without the hierarchical matching scheme gets a large promotion on the ranking list (see “Ours (w/o HM)” in Table IV) notice that hierarchical matching scheme is for fast approximation.)
Fig. 10. The improvement by introducing plane fitting scheme. The color coding of flow is from KITTI benchmark. The number in the captions means percentage of bad pixels (flow error larger than 5 pixels). Notice the improvement in the region marked by the white squares.

Fig. 11. Two snapshots of our result on video slow motion synthesis. Please refer to the supplementary materials for video examples.

D. Limitations

As a local method, our approach fails at large textureless regions, where local evidences are not enough to eliminate matching ambiguities. While increasing patch size or adding more cues (such as the census transform) might help relieve the problem, it cannot be completely avoided, especially when the regions are large. In addition, textureless regions can be easily affected by small noise or disturbance (such as the synthetic “textured” fog in Fig. 8), which may lead to incorrect match. In this case, global optimization techniques may be needed. However, notice that mismatch in textureless regions might not be a serious problem for some real-world applications.

IV. Demo Applications

We in this section show some results of applying our optical flow algorithm into real-world video editing applications. Notice that in these applications, a fast optical flow algorithm is the key to provide a user-friendly editing experience.

A. Video Slow Motion Synthesis

Slow motion synthesis is to construct artificial frames between existing video frames to make the video slow down to a certain speed. For example, if the video is going to be slow down to 10% speed, the number of total video frames should be 10x more and thus there should be 9 frames to be constructed between each two existing frames. The natural way to construct an artificial frame is to fetch pixels from the two existing frames between which the frame is to be placed. We compute the bidirectional (forward and backward) optical flow between the two existing frames, and then perform a forward warping on the flow fields to get two intermediate flow fields for a given time position between the two frames.
and then construct two intermediate artificial frames from the two existing frames, respectively. Finally we use the given time position to blend the two intermediate artificial frame to get a result frame, which is more similar to the frame to which the time position is closer. The optical flow computation is the most time-consuming part of this pipeline (the runtime of the other part can be ignored comparing to the runtime of optical flow algorithm). With our optical flow algorithm, we are able to achieve 40 fps (based on the number of output frames) for 640x480 videos when the target motion speed is 10%. Fig. [11] shows one example of our results. More video results can be found in the supplementary materials.

B. Flow-based Video Denoising

Optical flow can be used to establish temporal correspondences across video frames for high-quality video denoising [55]. For each patch in a certain frame, we first compute a few similar patches around it and then use the optical flow fields to collect more similar patches in nearby frames (according to the original patch and its similar neighbors). Finally, an algorithm similar to non-local means is applied to denoise the original patch using all the collected patches. The pipeline is pretty robust for producing temporal coherent video results. The main computation is again the optical flow estimation step. With our optical flow algorithm, the whole pipeline could be accelerated much. Fig. [12] shows an example of our results. More video results can be found in the supplementary materials.

C. Video Editing Propagation

A common way to reduce user efforts in video editing is to propagate user editing results (such as object selection, recolorization, and tone adjustment) from one frame to another according to optical flow field. Fig. [13] shows an example of propagating user selected object boundary with our flow result.

V. CONCLUSIONS

In this paper, we present an optical flow estimation approach that can efficiently produce high-quality results, even when the scene contains very large motions. Our method is local, yet independent of search range, and therefore is fast, thanks to the randomized propagation of self-similarity patterns and correspondence offsets, as well as the hierarchical matching scheme. Evaluations on public benchmarks demonstrate the effectiveness and efficiency of our algorithm. We believe our fast yet effective method will find its place in many practical applications.

REFERENCES


[9] K. He and J. Sun, “Computing nearest-neighbor fields via propagation-


[22] Z. Ma, K. He, Y. Wei, J. Sun, and E. Wu, “Constant time weighted median filtering for stereo matching and beyond,” in Proc. ICCV. IEEE,


Fig. 13. A snapshot of our results on video selection propagation. The red line in the first frame is selected by user and that in the second frame is automatically produced using optical flow.


