Fast Exact Nearest Patch Matching for Patch-Based Image Editing and Processing

Chunxia Xiao, Meng Liu, Yongwei Nie, and Zhao Dong, *Student Member*, *IEEE*

Abstract—This paper presents an efficient exact nearest patch matching algorithm which can accurately find the most similar patch-pairs between source and target image. Traditional match matching algorithms treat each pixel/patch as an independent sample and build a hierarchical data structure, such as kd-tree, to accelerate nearest patch finding. However, most of these approaches can only find approximate nearest patch and do not explore the sequential overlap between patches. Hence, they are neither accurate in quality nor optimal in speed. By eliminating redundant similarity computation of sequential overlap between patches, our method finds the exact nearest patch in brute-force style but reduces its running time complexity to be linear on the patch size. Furthermore, relying on recent multicore graphics hardware, our method can be further accelerated by at least an order of magnitude ($\geq 10\times$). This greatly improves performance and ensures that our method can be efficiently applied in an interactive editing framework for moderate-sized image even video. To our knowledge, this approach is the fastest exact nearest patch matching method for high-dimensional patch and also its extra memory requirement is minimal. Comparisons with the popular nearest patch matching methods in the experimental results demonstrate the merits of our algorithm.

Index Terms—Nearest patch search, texture synthesis, image completion, image denoising, image summarization.

1 Introduction

Nearest patch matching is a fundamental problem in computer graphics and computer vision, and has a variety of applications in image editing and processing, such as information retrieval [1], [2], texture synthesis [3], [4], [5], superresolution [6], image filtering [7] and image summarizing [8], etc. For these applications, nearest patch matching is defined as finding the nearest patch in a source image Z for each patch in a target image X under a matching-error metric. For a 2D image, to find the nearest patch in the source image Z with m pixels, the brute-force implementation will compare between each pixel of the patch and exhibits $O(mr^2)$ running time for patch of size r. Similarly, for video data which combine 2D images in the temporal domain, the search for nearest 3D patch of size r in a source video Z with m pixels costs $O(mr^3)$ running time. When the patch size r increases, the nearest patch matching becomes very time-consuming and usually becomes the bottleneck of applications.

Currently, there are two major kinds of strategies to accelerate the patch matching. The first one is relying on hierarchical tree structures such as kd-trees [9], TSVQ [4], and ANN [10]. Such approaches can only efficiently handle a low-dimensional patch. For a high-dimensional patch, they are usually combined with dimensionality reduction methods, like PCA, for acceleration. Therefore, their matching result is just approximate. The second kind of strategy is to limit the search space in the source image Z based on a local coherence assumption, such as the local propagation [11], randomized correspondence [12], and k-coherence technique [13]. Therefore, such an assumption could miss some significant information. The matching result is not optimal and might lead to many mismatches [14].

Inspired by Huang's [15] and Weiss's [16] works for accelerating median filtering, we introduce a novel fast exact nearest patch matching method. Our algorithm is based on the following observations: when sequentially performing brute-force nearest patch matching between the source image Z and the target image X, the adjacent patch-pairs overlap to a considerable extent. Making use of this sequential overlaps in each row to eliminate the redundant calculations, the time complexity for 2D patch can be reduced from $O(mr^2)$ to $O(mr)$. Furthermore, based on the sequential overlaps between the columns, processing multiple columns of patches simultaneously can further reduce running time to a constant for patch-pairs matching. Our method can be easily extended to accelerate 3D and even higher dimensional patch matching to be computed in the constant time.

Besides the significantly reduced time complexity, our method also has a much lower memory requirement compared to the existing nearest patch matching methods which rely on hierarchical structure. The hierarchical acceleration structures, such as the kd-tree [9], TSVQ [4], and ANN [17], usually demand memory in the order of $O(r)$ or even higher. However, our method does not require auxiliary data structures and the memory requirement is minimal. Hence, it can be applied for large image and video data precessing. In addition, since our method follows a brute-force comparison routine, the nearest patch matching result is guaranteed to be exact.

Relying on recent many-core platform, e.g., the GPU, our nearest patch matching method can be further accelerated

- *C. Xiao, M. Liu, and Y. Nie are with the Computer School, Wuhan University, Wuhan, Hubei, China. 430072. E-mail: cxxiao@whu.edu.cn, {mengliu.whu, nieyongwei}@gmail.com.*
- *Z. Dong is with MPI Informatik, AG4 CAMPUS E1 4, 66123 Saarbrücken, Germany. E-mail: dong@mpi-sb.mpg.de.*

Manuscript received 20 Nov. 2009; revised 14 May 2010; accepted 20 Sept. 2010; published online 13 Oct. 2010.
Recommended for acceptance by G. Drettakis.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2009-11-0267.
Digital Object Identifier no. 10.1109/TVCG.2010.226.

by at least an order of magnitude. We have efficiently implemented it in various image editing tasks, such as texture synthesis, image summarization, image completion, and image denoising. To our knowledge, this approach is the fastest exact nearest patch matching method for a high-dimensional input and its extra memory requirement is minimal. Moreover, its implementation is straightforward.

The rest of the paper is organized as follows: Section 2 reviews related work. In Section 3, the exact nearest patch matching method for 2D images is presented. In Section 4, the exact nearest patch matching method for video data is further presented. In Section 5, we introduce how to accelerate our methods in parallel using GPU. In Section 6, both the experimental results and the comparisons with previous methods are shown. Finally, we discuss the limitations of our method in Section 7 and conclude our paper in Section 8.

2 Related Work

A complete review of existing works is beyond the scope of this paper and we refer readers to [18], [19], [20] for excellent overviews on the nearest patch matching algorithms.

Nearest patch search algorithms can be roughly classified into two categories: the exact nearest patch matching and approximate nearest patch matching. PCA Trees [21], K-means [22], [23] are often used to achieve exact nearest patch matching. Currently, there are several methods, such as kd-Tree [9], ANN [10], TSVQ [4] and Vantage Point Trees [24], that can perform both exact and approximate nearest patch matching. All these methods apply hierarchical tree structure to accelerate searching. Some other methods such as local propagation method [11], k-coherence technique [13], and randomized correspondence algorithm [12] only perform approximate nearest patch matching. These approximate methods find the approximate nearest patch in local regions based on local coherence assumption. The performance of the nearest patch matching method usually depends on several factors: including image size, patch size, the number of nearest patches, search range, and the number of input images.

The kd-tree-based matching [9] is one of the most widely used algorithms for finding the nearest patch. Although it is easy to create and efficient for range query, the kd-tree only works well for low-dimensional data. Using the kd-tree, the number of searched nodes increases exponentially with the space dimension. When the dimension is large (for example, $N > 15$), its search speed becomes very slow. Another drawback is that the spatial divisions of the tree nodes are always axis-aligned. Hence, it induces an unbalanced tree and results in a poor search performance. Sproull [21] proposes PCA-tree structure which attempts to remedy the axis-alignment limitation of the kd-tree. It first applies Principal Components Analysis (PCA) at each tree node to obtain the eigen-vector which corresponds to the maximum variance, and then splits the points along that direction. Recently, Xiao and Liu [25] applied kd-tree to accelerate mean-shift clustering.

Both kd-tree and PCA-tree [21] are the so-called "projective tree" [20], since they categorize points based on their projection into some low-dimensional space. In contrast, k-means [22], [23] and vantage point tree (vp-tree) [24] are "metric tree" structures which organize points using a metric defined over pairs of points. Thus, they don't require points to be finite-dimensional or even in a vector space. K-means methods [22], [23] assign points to the closest of k centers by iteratively alternating between selecting centers and assigning points to the centers until neither the centers nor the point partitions change. The vp-tree [24] uses a single "ball" at each level. Rather than partitioning points on the basis of relative distance from multiple centers, the vp-tree splits points using the absolute distance from a single center. Although these methods can be used for nearest patch matching, their performances are usually too low to be applied in patch-based image processing and editing.

To reduce running time, instead of finding the exact nearest patches, approximate approaches return patches that are within a factor of $(1 + \varepsilon)$ of the true closest distance, for $\varepsilon \geq 0$. There are many existing approximate matching methods [4], [10], [17]. The ANN method [10] and tree-structured vector quantization (TSVQ) [4] exploit the tree structure to accelerate the search procedure. Therefore, they also suffer from both the large memory requirement and the cost for tree construction and traversal for high-dimensional data like aforementioned methods [9] [21]. To accelerate search procedure and reduce memory requirement, the dimensionality reduction techniques such as PCA can be integrated into the tree-based methods [5], [26]. Such a routine projects high-dimensional patch vector into low-dimensional space. Although performance is improved, a tree-based $\mathrm{ANN} + \mathrm{PCA}$ combination can miss significant information. Moreover, the running time is still long and high memory consumption remains unsolved.

To reduce memory consumption, many other approximate nearest patch matching methods without using tree-structure have been proposed for patch-based texture synthesis [11], [13], and structural image editing [12]. These methods apply the local image coherence assumption to reduce the search space. However, such an assumption could miss significant information. The matching result is not optimal and might lead to many mismatches.

The Fast Fourier Transforms (FFT) methods [27], [28] are also used to accelerate approximate nearest patch matching. Combining summed-area tables [29] and FFT techniques, the complexity for searching the nearest patch for each patch is $(O(n) + O(nlog(n))$, where n is the number of pixels in the image. The computational complexity of these methods is still high for processing the large image. Moreover, these methods cannot guarantee to find the exact nearest patch.

In contrast to the existing nearest patch matching methods, we propose a fast exact nearest patch matching method based on eliminating the redundant computations of the brute-force matching routine. Huang [15] presented an improved algorithm for median filter by making use of sequential overlaps between adjacent windows to eliminate the redundant calculations. Weiss [16] further improved the performance of the median filter by processing multiple columns at once. Later, Rivers and James [30] extended the method [16] to the fast shape matching. In this paper,

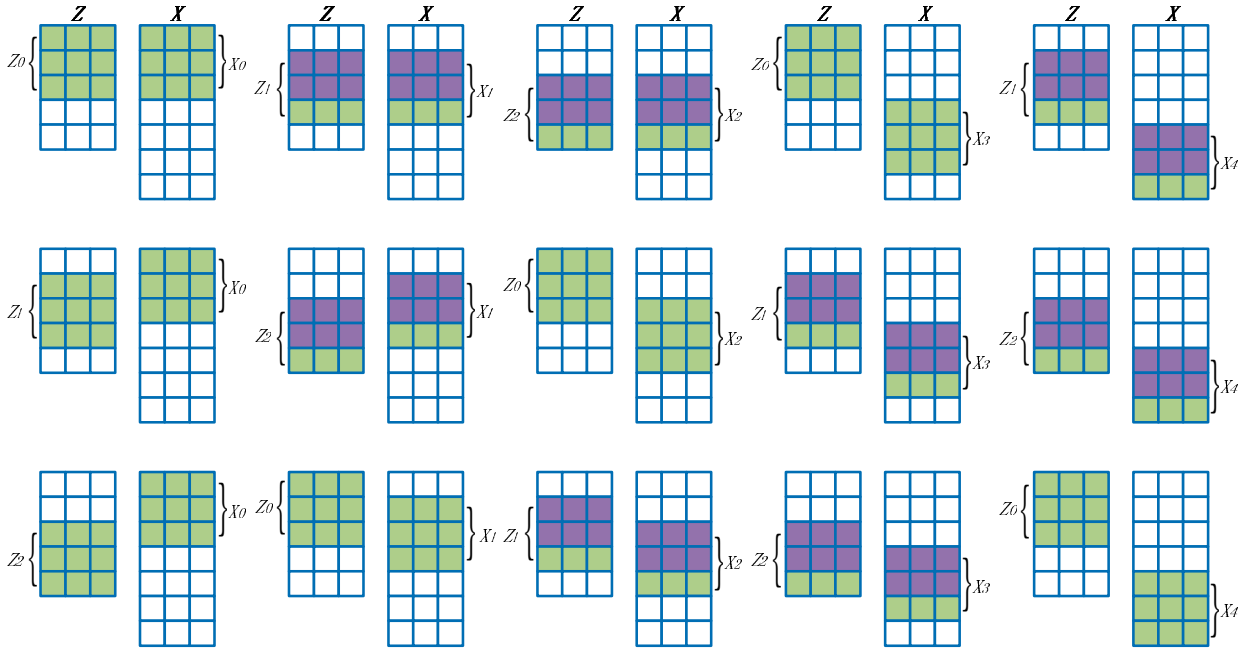


Fig. 1. Nearest patch matching for target image X ($P = 5$), source image Z ($S = 3$), and the patch size $r = 3$. The first row is the first iteration ($L = 0$), the $d(X_0, Z_0)$ is first computed, then its consequent patches (X_1, X_2, X_3, X_4) are compared with the corresponding consequent patches (Z_1, Z_2, Z_0, Z_1) in Z. Specially, based on the overlap between $d(X_0$ and $Z_0)$, $d(X_1, Z_1)$ is computed, similarly, based on the overlap between X_1 and Z_1, $d(X_2, Z_2)$ is computed, then we compute $d(X_3, Z_0)$, based on the overlap, $d(X_4, Z_1)$ is computed. The second row is the second iteration ($L = 1$), the $d(X_0, Z_1)$ is first computed, its consequent patches (X_1, X_2, X_3, X_4) are compared with the corresponding consequent patches (Z_2, Z_0, Z_1, Z_2) in Z. Note that the overlaps are used in the distance computation. The third row is the third iteration ($L = 2$), similarly, the $d(X_0, Z_2)$ is first computed, its consequent patches (X_1, X_2, X_3, X_4) are compared with the corresponding patches (Z_0, Z_1, Z_2, Z_0) in Z.

inspired by [15], [16], we address a different problem. We speed up the brute-force patch matching by eliminating the redundancy of sequential overlaps between adjacent corresponding patch-pairs. Our method guarantees that each patch in the target image X compares every patch in source image Z only once, furthermore, our method eliminates the redundant computations of the adjacent corresponding patch-pairs.

3 Fast Nearest Patch Matching

Using the standard euclidean distance function, the similarity distance (patch distance or patch matching error) between the 2D patch X_m in the target image X and Z_n in the source image Z with size r can be computed as: $d(X_m, Z_n) = \sum_{i=1}^{r}\sum_{j=1}^{r}[X_m(i,j) - Z_n(i,j)]^2$. Here, (i, j) represents the index position for each pixel inside a patch. This is the core computation task of the patch matching and its time complexity is $O(r^2)$ for a 2D image. Our novel algorithm is based on the following observation: when performing the nearest patch matching in sequential order using a brute-force routine, the adjacent corresponding patch-pairs overlap to a considerable extent. By exploiting this sequential overlap, we eliminate the redundant computations, thus the time complexity for the patch matching is significantly reduced. In the rest of this section, we illustrate the key idea of our fast matching method by processing a 2D image.

3.1 The Basic O(r) Algorithm

As illustrated in Fig. 1, the term nearest patch matching means that each patch X_i in the target X should find a nearest patch Z_i in the source Z. Using the brute-force routine, X_i should compare with each patch Z_j in Z to find the nearest patch. The overall time complexity $O(r^2 \times \Phi \times \Psi)$ is huge for large patches, where Φ and Ψ is the size of X and Z, respectively. Notice that adjacent patch-pairs overlap to a considerable extent when performing a sequential brute-force search. As shown in Fig. 1, the similarity difference results in the overlapped region (purple) between pair (X_0, Z_0) and pair (X_1, Z_1) are actually the same. Hence, when computing the similarity for patch-pair (X_0, Z_0), the results in the overlapped region can be kept. Then, the distance for patch-pair (X_1, Z_1) can be computed by summing the distance in the overlapped part and the distance of the corresponding bottom row of patch-pair (X_1, Z_1). As the patch-pairs slide over respective images along each column, redundant calculations become sequential. We can make use of the sequential overlap to significantly reduce the time complexity.

More specifically, we describe how to find a nearest patch in the first column of the source Z for each patch of the first column in the target X. As in Fig. 1, there are P and S patches in each column of the target and source image, respectively. Each patch slides only by one pixel of each step. X_0 and Z_0 are first compared. Then, based on the preserved overlapped results, X_1 and Z_1 are compared. The similarity of adjacent patch-pairs continues to compute until the end of the first iteration when X_{P-1} and $Z_{(P-1)mod(S)}$ are compared. Then, similarly to the first iteration, X_0 is compared with Z_1 and its subsequent patches are compared with the corresponding subsequent patches in Z (namely, X_1 with Z_2, X_2 with $Z_3, \ldots$) until the end of the second iteration. In the final iteration, X_0 is compared with Z_{S-1}, and the subsequent

corresponding patch-pairs are compared until X_{P-1} finishes the comparison with $Z_{(P+S-2)mod(S)}$. Fig. 1 illustrates the above steps for the target image X with $P = 5$ and the source image Z with $S = 3$.

It can be observed that using the sequential overlap, our method guarantees that each patch X_i will only compare with each Z_j once and the redundant calculations are eliminated. Note that when the pair (X_i, Z_j) comes to the occasion that Z_j is the last patch $(Z_j == Z_{S-1})$, then X_{i+1} corresponds to the first patch Z_0 (the pair (X_{i+1}, Z_0)). In this special case, the similarity (X_{i+1}, Z_0) needs to be fully computed since there is no overlap available. After performing the nearest patch matching for each column of X with each column in the source Z, we have finished the nearest patch search in the image Z for each patch in the target X.

Using this method, the 2D patch matching complexity is reduced from $O(r^2)$ to $O(r)$ for each patch. For the large patches (32×32 and larger), the proposed method becomes dramatically fast. The pseudocode of the basic $O(r)$ algorithm for the 2D patch matching is presented in Algorithm 1. In Fig. 3a, the time comparison with the naive brute-force method is given. The results demonstrate that our basic algorithm successfully break the $O(r^2)$ time complexity down to $O(r)$.

Algorithm 1. Pseudocode for $O(r)$ algorithm processing single column.

```
Z: Source image
X: Target image
P,S: Number of patches in each column of Xand Z
Overlap: Distance of the overlapped region
Index[i], Weight[i]: The nearest patch in Z for X_i and
    the corresponding Distance.

for L = 0 to S − 1 do
    for K = 0 to P − 1 do
        let Γ = (K + L)mod(S)
        if (Γ == 0 or K == 0) then
            Dist = Distance between X_K and Z_Γ;
            Compute Overlap;
        else
            Dist = Overlap+Distance of bottom row in
            (X_K, Z_Γ);
        end if
        if Dist < Weight[K] then
            Index[K] = Γ;
            Weight[K] = Dist;
        end if
        Overlap = Dist-Distance of top row in (X_K, Z_Γ)
    end for
end for
```

3.2 Processing N Columns Simultaneously

The basic algorithm in Section 3.1 eliminates the redundant computation of adjacent rows for the single-column processing. To further improve performance, we now perform the nearest patch matching for N columns by comparing N adjacent columns of patches in target image X with N adjacent columns of patches in source image Z. Inspired by the method presented in [16], we propose to process multiple columns simultaneously by eliminating the redundant computation of adjacent columns.

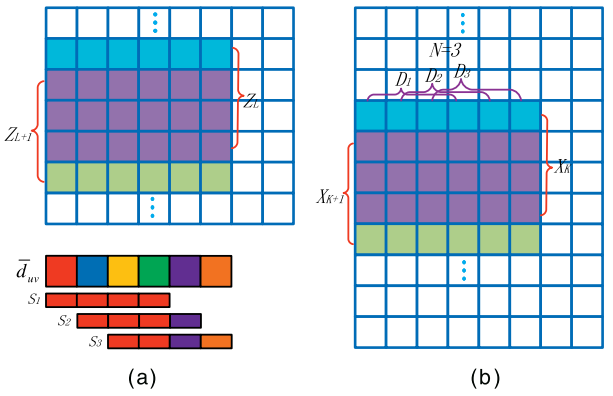


Fig. 2. N columns processing. (a) Source Z, (b) target X. The patch size $r = 4$, and $N = 3$. A bottom row of pixel-pairs (green) are newly added to D*. To compute the distances of patch-pairs, we first sum the first r pixel-pair distances: $S_1 = d(u_0, v_0) + \cdots + d(u_{r-1}, v_{r-1})$ and it is the sum of new row for patch-pair distance $D_1^{'}$. Except for S_0, the following S_i can be computed as: $S_i = S_{i-1} - d(u_{i-1}, v_{i-1}) + d(u_{r+i-1}, v_{r+i-1})$, as illustrated in bottom sketch map in (a). By adding S_i to column D_i and subtracting the old top row (blue) pixel-pairs distances from D_i, we receive the new patch-pair distance $D_i^{'}$.

Processing N columns simultaneously involves the computation of N dependent patch distances (D^*: $D_0 \ldots D_{N-1}$) for each output column. Each distance D_i can be computed efficiently by eliminating the redundant computation of adjacent columns. Assuming that $D_0 \ldots D_{N-1}$ of current N adjacent patches have been computed, now we compute the distances $D_0^{'} \ldots D_{N-1}^{'}$ of the next N adjacent patches. As shown in Fig. 2, the newly introduced data for the next N adjacent patches is just the bottom row of pixels. Therefore, we first compute the distances between each pixel-pair $d(u_i, v_i) = (u_i - v_i)^2$ of the new bottom row $(v_0 \ldots v_{r+N-2})$ from the input Z and its corresponding row $(u_0 \ldots u_{r+N-2})$ in output X. Here, u_i and v_i represent the pixel in X and Z, respectively. Then, the pixel-pair distances $(d(u_0, v_0), \ldots, d(u_{r+N-2}, v_{r+N-2}))$ is added to patch-pair distances $D_0 \ldots D_{N-1}$, as shown in Fig. 2. Finally, by subtracting the old top row pixel-pair distances from $D_0 \ldots D_{N-1}$, we receive the distances $D_0^{'} \ldots D_{N-1}^{'}$ for the next adjacent N patches. Note that old top row pixel-pair distances do not need to be recomputed, since they have already been kept after the previous computation.

Using this mechanism, for N output columns, the number of multiplication operations reduces from $N \times r$ to $N + r - 1$, and the average number of the pixel-pairs distance computation for per patch-pair reduces from r to $1 + (r - 1)/N$. The time for computing distances between each patch-pair is still O(r), but with a much lower constant than the single-column algorithm. Let $N = r - 1$, to process N columns simultaneously for patch-pairs matching, the average number of the pixel-pairs distance computation for each column is the constant 2. Compared with the algorithm of single column in $O(r)$ runtime, a significant improvement in efficiency is achieved. By widening N to fit the widening size of patch r, N columns technique can be adapted to perform patch matching of an arbitrary size. In all the experiments in our paper, we process $N = r - 1$ columns simultaneously.

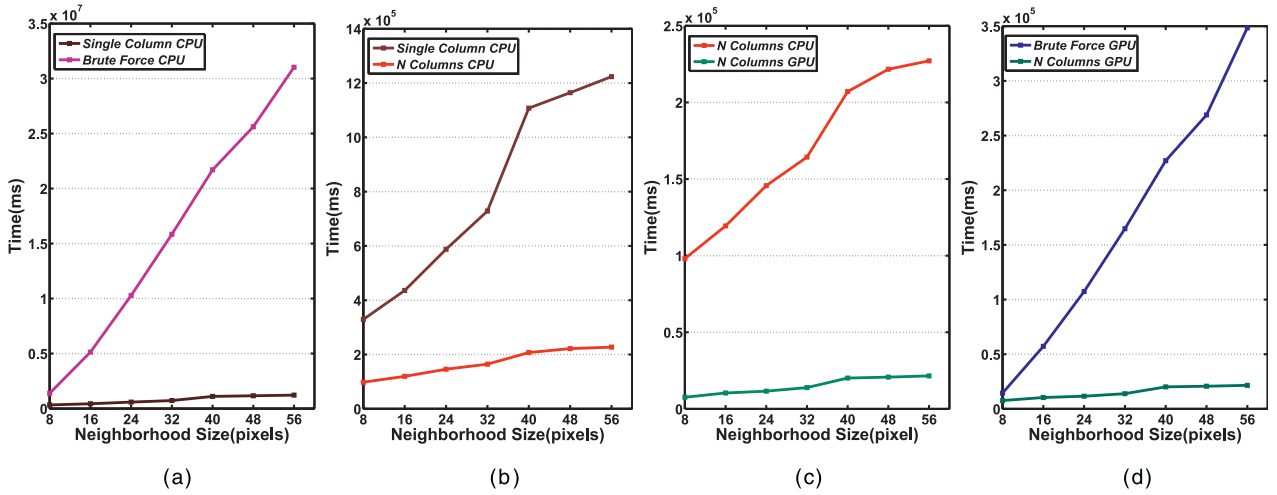


Fig. 3. Time complexity comparison in different patch sizes, (a) proposed basic single-column processing versus naive brute-force patch match, (b) single-column processing versus multiple columns processing, (c) GPU parallel processing for multiple columns versus single-threaded CPU implementation for multiple columns processing. (d) GPU parallel processing for multiple columns versus GPU parallel processing of the naive brute-force patch matching. The size of source image Z is 256×256, and size of target image X is 278×278.

To perform the nearest patch search for each patch in the target image X, of each iteration, we compare N adjacent columns of patches X_N in the target image X with each N adjacent columns of patches Z_N in the source image Z. Then, each Z_N steps one pixel along the image Z, while X_N steps N pixels along the image X. This guarantees that each patch X_i is compared against every patch Z_j only once. For the patches near the boundary, we use the self-adaptive multiple columns processing (N is gradually reduced) to eliminate the redundant computation between the columns.

Compared with the single-column algorithm, the multiple columns algorithm further accelerates the nearest patch search, as illustrated in Fig. 3b. The experimental results prove that using the multiple columns algorithm, the computational complexity on average remains constant for each patch-pair distance computation for different patch sizes. The total computational time is only slightly increased by increasing the patch size.

4 Fast 3D Patch Match Computing

Compared to the 2D image, video data are usually much larger in size. In addition, the time complexity for the standard 3D patch match is $O(r^3)$ with the patch size r. The cubical-complexity time of 3D nearest patch search leads to an intractable scenario for a naive brute-force routine. In this section, we generalize the fast nearest patch matching presented in the image case to accelerate the nearest 3D patch matching for video data.

Similar to the image search case, observing the overlap of the adjacent cube-pairs, we eliminate the redundancy of the adjacent cubes in each column to accelerate the nearest patch matching. As shown in Fig. 4a, we just need to incrementally compute the pixel-pair distances for the new bottom layer of pixels. The time complexity of the 3D cube matching is reduced from $O(r^3)$ to $O(r^2)$. In the remainder of this section, we will present two strategies for 3D cube matching: processing N columns of 3D patches simultaneously and processing $N \times N$ 3D patches simultaneously, which further accelerate the 3D patch matching.

4.1 Processing N 3D Patches Simultaneously

Similar to the image case, we process N columns of 3D patches simultaneously, as shown in Fig. 4b. For the case of 3D patch matching, we add one new bottom layer of pixels $u_{[0..r-1][0..r+N-2]}$ and $v_{[0..r-1][0..r+N-2]}$ to input and output video separately. To compute the patch distances $D_0^{'} \ldots D_{N-1}^{'}$ for the next adjacent N 3D patches, we just need to compute the pixel-pair distances d_{uv} of the bottom layer and add them to $D^* = (D_0, .., D_{N-1})$, where D_i is the similarity of ith 3D patch-pair. Fig. 4a shows how a row of patch-pairs is added to the D*.

Processing N columns independently usually requires overall $N \times r^2$ multiplications for distance computation of

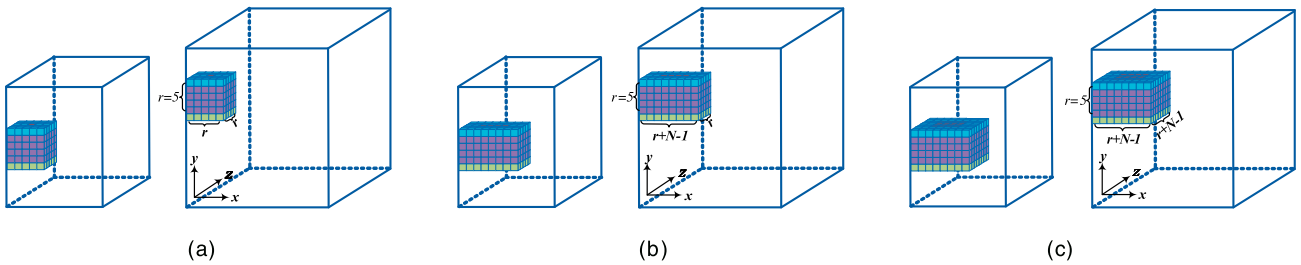


Fig. 4. (a) 3D patch matching algorithm on a single column. (b) Processing N columns simultaneously, adding new ($[0..r-1][0..r+N-2]$) pixel-pairs distances to $N-1$ matching distances $D^* = (D_0, .., D_{N-1})$. (c) Processing $N \times N$ patch-pairs simultaneously, adding ($[0..r+N-2][0..r+N-2]$) pairs of pixel distances to D^* ($D_{i,j}(i, j \in [0, N-1])$).

pixel pair $d(u_{i,j}, v_{i,j})$. In comparison, our scheme for processing N columns simultaneously only involves $(N + r - 1) \times r$ multiplication operations to compute the distance of the $(N + r - 1) \times r$ pixel-pairs. Using this mechanism, the complexity is significantly reduced, for example, if $N = r - 1$, the multiplication operation reduces from $N \times N^2$ to $2 \times N^2$. The average number of the pixel-pairs distance computation for N columns is $2 \times N$ for the cube with size r, and the number of addition operations is also greatly reduced.

Similarly to the image case, we compute the distance $S_i (i = 0, .., r + N - 2)$ for $(u_{[0..r-1][i]}; v_{[0..r-1][i]})$ of the added layer, and add S_i to D* for the 3D patch similarity computation.

4.2 Processing $N \times N$ 3D Patches Simultaneously

Observing that the 3D patches considerably overlap in the direction of the z-axis (Fig. 4c), we further eliminate the redundancy between adjacent patches and accelerate the 3D patch matching. The fundamental idea behind the mechanism is to process $N \times N$ adjacent patches simultaneously: we first process the first row that contains N columns of 3D patches using the multiple columns acceleration algorithm, then, based on the computed sequential patches distance, the rest of adjacent $N - 1$ row patches (the direction of the z-axis) can be computed sequentially.

Specially, to process $N \times N$ adjacent patches simultaneously, the new $[0 \ldots r + N - 2][0 \ldots r + N - 2]$ pixel-pairs are added to the corresponding $N \times N$ patch-pairs, as illustrated in the schematic (Fig. 4c). The overall number of multiplication operations is $(N + r - 1) \times (N + r - 1)$, for computing the distance of each pixel-pair. For $N \times N$ patch-pairs, the average number of distance computations for per patch-pair is $(N + r - 1)^2 / N^2$. Let $N = r - 1$, the average number is the constant 2^2, which greatly improves the performance compared to processing N rows 3D patches independently. The overall computational cost of our method outperforms naive brute-force method by up to a factor of r^3. Similarly as before, we add the $(N + r - 1) \times (N + r - 1)$ pixel-pair distances to the $N \times N$ patch matching distance set D^* $(D_{i,j}(i, j \in [0, N - 1]))$, and obtain the distances of the current $N \times N$ 3D patch-pairs.

To perform the nearest patch search for each patch in X, we compare $N \times N$ patches $X_{N \times N}$ of the output X with each $N \times N$ patches $Z_{N \times N}$ simultaneously. More specifically, each $Z_{N \times N}$ slides one pixel along x-axis direction and one pixel along z-axis direction, while $X_{N \times N}$ slides N pixels along x-axis and z-axis direction, as shown in (Fig. 4c). The number N can be adjusted near the boundary of the video data.

4.3 Computational Complexity Analysis

Our fast nearest patch matching can be generalized to handle a d-dimensional continuous data set R^d. Similar to Sections 3 and 4, to find the nearest d-dimensional patch with size r, we can process $(N \times N \ldots \times N)_{d-1}$ columns simultaneously. The patch matching time complexity for d-dimensional patch-pair can be reduced from $N \times (r)^d$ to $(N + r - 1)^{d-1}$. Let $N = r - 1$, the average multiplication operations for each d dimensional patch-pair is 2^{d-1}. Using our proposed method, the average multiplication operations for each patch-pair is a constant, regardless the patch size r. It should be pointed out that near the boundary regions of the d dimensions data, the number N must be adjusted accordingly. It will make the computational complexity negligibly higher than 2^{d-1}. Since our method requires no auxiliary structure, the memory requirement is always small.

5 GPU Acceleration

Recent many-core graphics processing units (GPUs) exhibit great parallel computation potential and incur performance breakthrough for many time-consuming algorithmic implementations [31]. Based on its SIMD pipeline structure, the GPU is especially appropriate for accelerating an algorithm which requires few data synchronizations and can be implemented in a highly parallel way. The data synchronization somehow serializes the parallel processing routine by introducing data waiting stage for threads execution, hence its cost usually dominates the overall performance of a parallel implementation.

Our exact nearest patch matching method relies on data-reuse to reduce computations. It intrinsically contains data dependences between the matching operation of neighbor patches. The simple GPU translation for its implementation could only achieve limited performance improvement ($\sim 3\times$) over the CPU code. To maximally utilize the parallel computing infrastructure of the GPU, we propose to copy multiple rows of data into the local (shared) memory in the GPU kernel and compute pixel-pair distance for multiple patch-pairs in parallel. We present the GPU implementations for both single-column and N-columns patch matching methods.

We provide pseudocode for the GPU implementation of the single-column method in Algorithm 2 and refer to the line numbers as Lxx in the following text. The illustrations for the GPU implementation for the single-column processing is given in Fig. 5. Initially, we compute $r \times r$ pixel-pair distances between the source patch Z_Γ and target patch X_K in parallel and sum up each row of the pixel-pair distance map to obtain r per-row distances (L13-L14, corresponding to Figs. 5a and 5b). Next, instead of sliding each patch-pair only one pixel along each column, during each following iteration, the r rows of new pixels (nonoverlapping with the previous one) are loaded into the GPU local memory (L22 and Fig. 5c). Again we compute the $r \times r$ per-pixel distances and the r per-row distances (L23 and Fig. 5d). Now there is in memory an array of $2 \times r$ per-row distances which corresponds to contiguous rows of r patch-pairs. Hence, running r threads in parallel, each thread sums r entries of the $2 \times r$ per-row distances (L24 and Fig. 5e). The distance between r patch-pairs can be obtained efficiently. By using such a multirows parallelism strategy, the synchronization latency of single-column patch matching is greatly hidden.

Algorithm 2. Pseudocode for GPU $O(r)$ algorithm single-column processing.

1: Z: Source image, X: Target image, r: Patch size
2: P,S: Number of patches in each column of X and Z
3: $Index[i]$, $Weight[i]$: The nearest patch in Z for X_i and its distance
4: $PixelDis[r][r]$: pixel-pair distances array between two patches

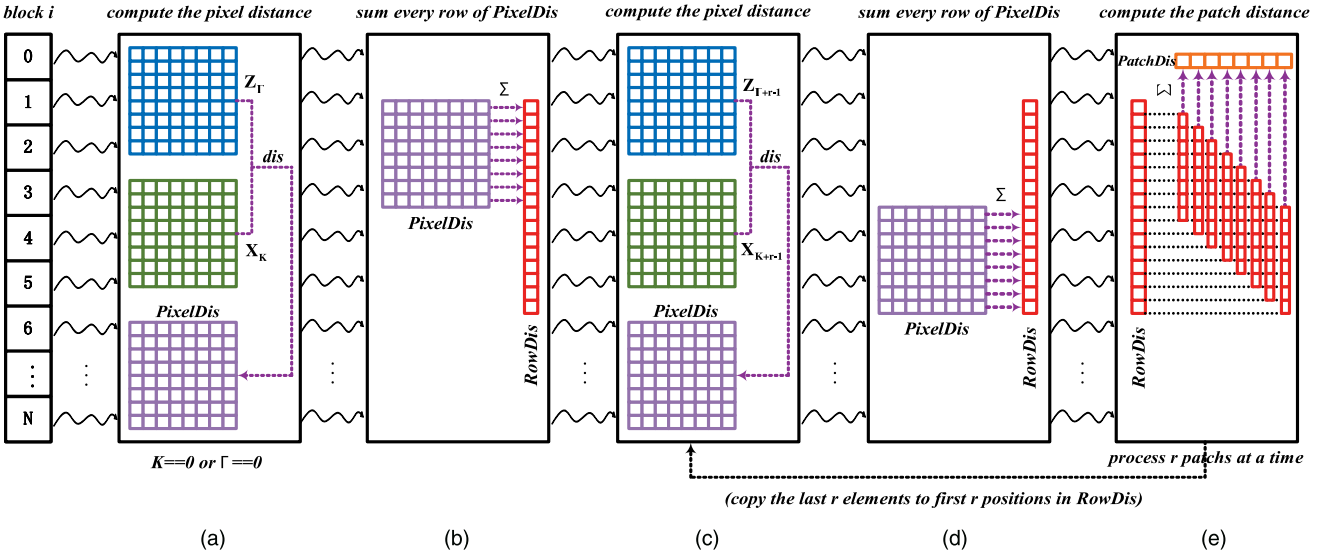


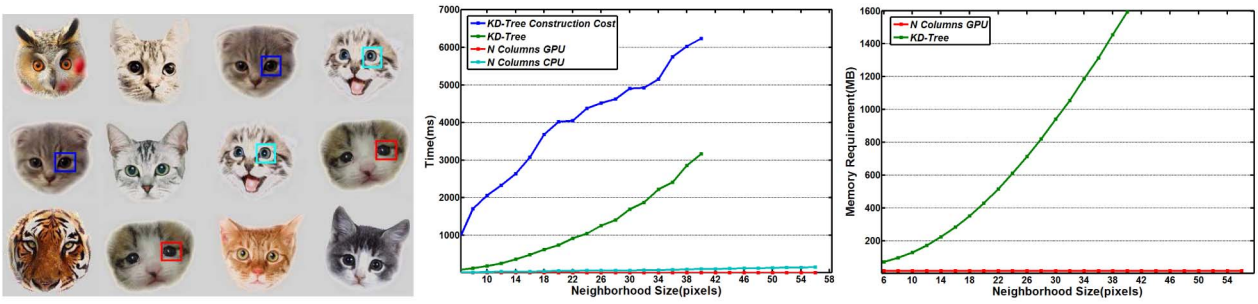
Fig. 5. Illustrations for the GPU single-column processing algorithm ($O(r)$). After r^2 pixel-pair distances between patch Z_Γ and X_K (a), and $RowDis[0, r-1]$ are implemented in parallel using GPU, the r rows data are copied into the GPU's local memory (b). Then, (c) we compute in parallel the pixel-pair distances between patch $Z_{\Gamma+r-1}$ and X_{K+r-1}, (d) compute and store $RowDis[r, 2r-1]$ (d). (e) Relying on the available $RowDis[0, r-1]$, the matching comparison of next adjacent r rows patches can be done in parallel.

5: $RowDis[2r]$: The sum of elements in each row of $PixelDis[][]$
6: $PatchDis[V]$: PatchDistance
7:
8: **for** $L = 0$ to $S-1$ **do**
9: **while** $K \leq P-1$ **do**
10: Let $\Gamma = (L+K) mod(S)$
11: **if** ($K == 0$ or $\Gamma == 0$) **then**
12: Computing $PixelDis[][]$ between Z_Γ and X_K **in parallel**
13: Computing $RowDis[0 \ldots r-1]$ of $PixelDis[][]$ **in parallel**
14: $Dist =$ *Sum* of all the elements of $RowDis[0 \ldots r-1]$
15: **if** $Dist \leq Weight[K]$ **then**
16: $Weight[K] = Dist$
17: $Index[K] = \Gamma$
18: **end if**
19: $K++$
20: **else**
21: Computing $PixelDis[][]$ between $Z_{\Gamma+r-1}$ and X_{K+r-1} **in parallel**
22: Computing $RowDis[r \ldots 2r-1]$ of $PixelDis[][]$ **in parallel**
23: **for** $V = 0$ to $r-1$ **in parallel do**
24: $PatchDis[V] =$ *Sum* of the elements of $RowDis[V+1 \ldots V+r]$
25: **if** $Dist < Weight[K+V]$ **then**
26: $Index[K+V] = \Gamma + V$
27: $Weight[K+V] = PatchDis[V]$
28: **end if**
29: $RowDis[V] = RowDis[V+r]$
30: **end for**
31: $K+ = r$
32: **end if**
33: **end while**
34: **end for**

The proposed N-columns patch matching method can also be implemented in GPU. The strategy is similar to the single-column case. The $r \times (r+N-1)$ pixel-pair distances $PixelDis[0, r-1][0, r+N-2]$ between N columns patches in Z and the corresponding N columns patches in X can be initially computed in parallel. Similarly, the pixel-pair distances sum $ColumnDis[0, r+N-2]$ in each column of $PixelDis[0, r-1][0, r+N-2]$, can also be computed in parallel. During each iteration, $r \times N + r - 1$ new pixels are copied into the GPU local memory. With the available $N+r-1$ columns pixel-pair distance array $ColumnDis[0, r+N-2]$, the N columns patch matching are done in parallel. Compared to the single-column case, implementing the N columns patch matching in GPU further improves the performance.

Our GPU implementation is based on Nvidia's CUDA [32]. As illustrated in Fig. 3c, using the GPU acceleration techniques, our nearest patch matching method can be further accelerated by at least an order of magnitude ($\geq 10\times$) compared to its CPU implementation. Such an efficient performance ensures that our method can be applied in an interactive editing task for a moderate-sized image. For example, in nonlocal image denoising application (described in Section 6) with the image of size 256×256, interactive performance can be achieved for generating the denoised results.

We also compare the performance of the GPU acceleration of our proposed method with the GPU implementation of the naive brute-force patch matching. As shown in Fig. 3d, when the patch size is very small, our method does not show much advantage over the naive brute-force method. However, when the patch size becomes larger, our method becomes increasingly faster. This occurs because the time complexity of each 2D patch-pair matching in our method can be reduced to constant. However, for the naive brute-force method, there is too much redundant computation



(a) (b) (c)

Fig. 6. Comparison with kd-tree. (a) The image (840×600), (b) time complexity comparison for the patch search with different sizes on the kd-tree construction, the kd-tree patch search, our N columns on CPU, and our N columns on GPU, (c) memory requirement comparison for the patch search with different sizes.

when the patch size is large. Therefore, although implemented in GPU, its speed is still much slower than our GPU method and even slower than the single-threaded CPU implementation of our multiple columns processing. With the rapidly developing GPUs equipped with more shared memory in near future, our method can get further performance improvement since more rows of data could reside in the local (shared) memory.

6 Experimental Results and Applications

We apply our fast nearest patch matching method in several applications, including the nearest template patch search, nonlocal filtering [7], optimization-based texture synthesis [5], [33], image and video completion [34], and image summarization [8], [12]. In this section, we show and discuss the results of all these applications. We compare the performance of our method against both the exact nearest patch matching method, like kd-tree, and the approximate nearest patch matching methods, such as ANN [10], TSVQ [4], and FFT method [27]. Furthermore, we show some comparisons with the most recently randomized correspondence algorithm [12] which uses the image local coherence assumption. All the comparisons with these methods focus on the following aspects: memory requirement, time complexity, and the quality of image processing and editing results. Our approach is implemented in C++ on a Pentium Dual-Core CPU E5200@2.50 GHz with 2 GB RAM. The GPU acceleration is based on CUDA [32] and run on a NVIDIA GeForce GTX 285 (1 GB) graphics card.

6.1 Nearest Template Patch Matching

Since kd-tree is one of the most popular methods for the nearest template patch matching, we compare the performance of our method against the kd-tree method for the exact nearest template patch with different patch sizes. As illustrated in Fig. 6, the kd-tree only works well for a low-dimensional case. When the dimension is large (for example, $N > 15$), the kd-tree may become very slow because the number of the searched nodes increases exponentially with the space dimension. Note that the comparison is performed based on the same criterion: the nearest patch is searched for every patch in the image. For the kd-tree method, both the time for the tree construction and the time for performing the nearest patch search with different patch sizes are given in Fig. 6b. The search time means the average time for finding the nearest patch for each patch.

6.2 Nonlocal Image and Video Denoising

We further apply our nearest patch matching technique for accelerating the nonlocal means (NL-means) image denoising [7]. Different from most filtering methods which perform locally, this algorithm is based on a nonlocal average of all pixels in the image. NL-means [7] works well for the image filtering. However, it is also notoriously slow since the similarity for each patch Z_i with each other patch Z_j in the image has to be computed.

To accelerate the NL-means algorithm, the search for similar windows is usually restricted in a "search window" of size $S \times S$ pixels [7] larger than the patch size of Z_i. Using this technique, the method cannot restore the details and fine structure of the noisy images as well as the globally weighted average. Nevertheless, using our method by eliminating redundant similarity computation between the overlap patches, not only the similarity is computed at extremely fast speed, but also the exact result quality is ensured.

In Fig. 7, we compare the results between our method which uses the globally weighted average with NL-means algorithm which restricts the search of similar windows in a "search window" of size $S \times S$ pixels. As shown in the zoom-out results of Fig. 7i, our method gives better filtering results. The image difference $u - NL_h(u)$ is displayed in Figs. 7e and 7g. The performance of our method for nonlocal filtering using patches of different sizes is given in Fig. 7j.

In Fig. 8, we present nonlocal video denoising results using the proposed 3D patch similarity computation. In comparison, it only takes about 21 minutes to filter a video ($400 \times 300 \times 280$) by the GPU implementation of our method. We also give the results with different 3D patch sizes and find that the patch size ($7 \times 7 \times 7$) usually works best.

6.3 Optimization-Based Texture Synthesis

Optimization-based texture synthesis methods [5], [33], [35] apply an optimization process to iteratively increase the similarity between the output synthesized texture and the exemplar. To accelerate texture optimization, the critical step is to find the best matching patch Z_p in the input exemplar Z for each patch X_p in the output synthesized

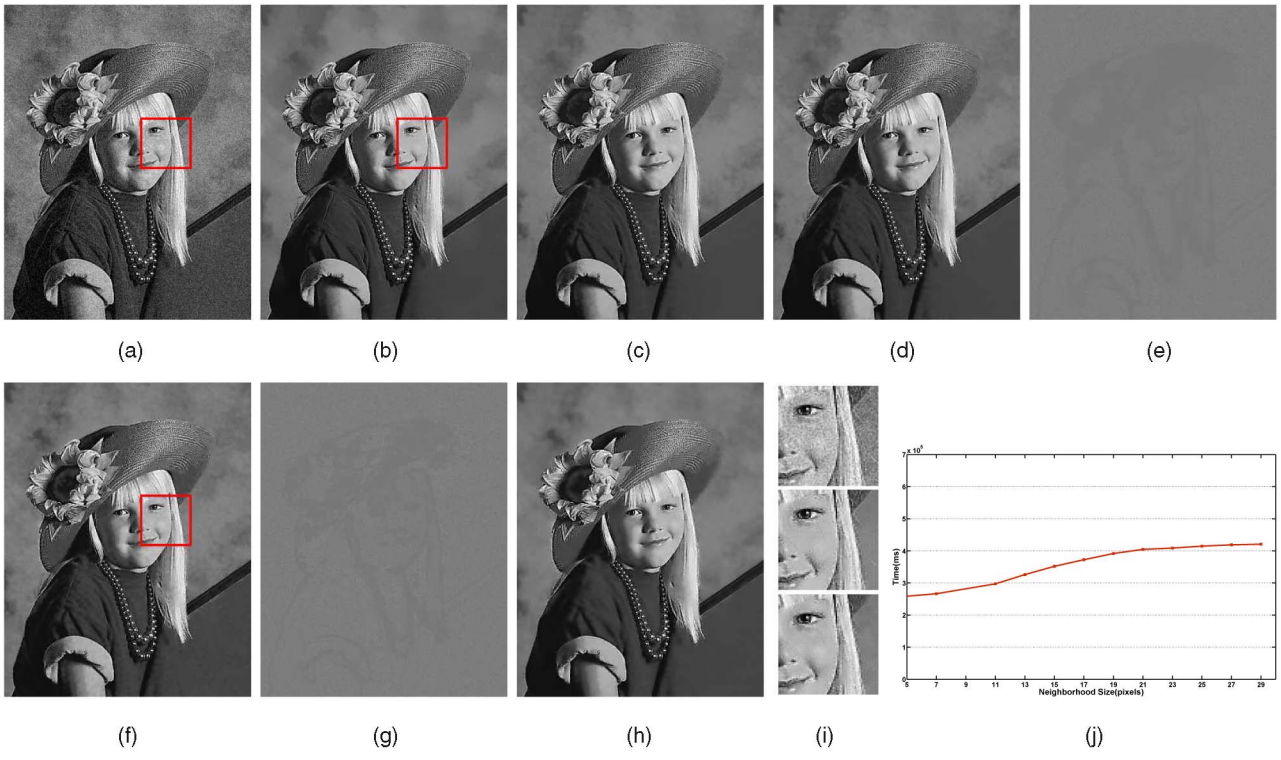


Fig. 7. Nonlocal image denoising. (a) noisy image u (480×612), (b),(c),and (d) are the denoised images $D_h(u)$, we apply similarity square patch Z_i of 7×7 pixels, and fix a search window of 21×21, 71×71, 101×101 pixels, respectively, (e) displaying of the image difference $u - NL_h(u)$ between (a) and (d), (f)and (h) the filtering results using the global weighted averaging, the similarity square patch Z_i is set 7×7 and 9×9 pixels, respectively, (g) the image difference $u - NL_h(u)$ between (a) and (f), (i) the zoom-out results of (a), (b), and (f), (j) time complexity for global weighted averaging using patch with different sizes.

texture X. Using our method, the texture optimization is greatly accelerated.

We compare the performance of our method with the other two approximate nearest patch matching methods: ANN [10] and TSVQ [4]. The timing is given in Figs. 9, 10, and Table 1 for different patch sizes. Note that for comparison, we test ANN and TSVQ and our method with single-threaded CPU implementation. It turns out that our method is already faster. Also note that the time for ANN and TSVQ does not include the preprocessing time of the tree construction, it usually takes several seconds to minutes to build the tree for TSVQ and ANN depending on the input data. Building the tree structure also dominates the memory requirement for ANN [10] and TSVQ [4]. When processing the large patches, the time complexity and storage considerations of ANN and TSVQ incur serious difficulty. In contrast, our method does not share this problem(see Fig. 9d). The ANN method takes the value ε as a parameter, and returns an approximate nearest patch that lies no farther than $(1 + \varepsilon)$ times the distance to the exact nearest patch. We use $\varepsilon = 1.5$ and find that it is a good compromise between the speed and accuracy. The halting criteria for TSVQ method is that the error between the two sequential nearest patches is below 10^{-10}.

In practice, as pointed in [33], it is computationally expensive to compute the energy over all patches in the texture. A subset of neighborhoods $X^{\dagger}$ that sufficiently overlap with each other can be selected. Defining the energy only over this subset will produce desirable results. Since it only needs to find the nearest patches for a subset of patches in X, this may result in the faster performance for the ANN and TSVQ than finding the nearest patches for all patches in X. However, as illustrated in Fig. 9c and Table 1, even choosing $X^{\dagger}$ neighborhood centers that are r/w pixels apart

Fig. 8. Nonlocal video denoising. (a) Noisy video u $(400 \times 300 \times 280)$, (b) denoised image $D_h(u)$, (c) and (d) displaying of the image difference $u - D_h(u)$ with 3D patch size $(11 \times 11 \times 11)$ and patch size $(7 \times 7 \times 7)$, respectively. The 180th frame of the video is illustrated.

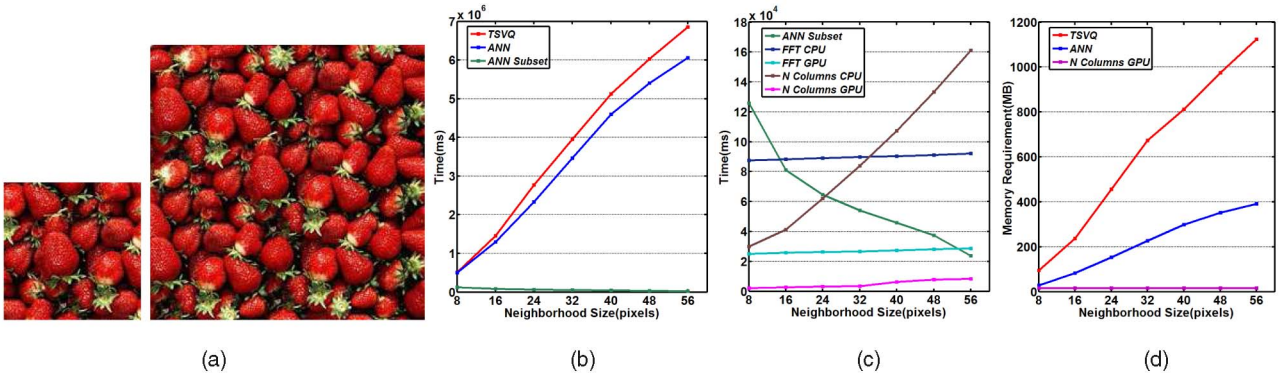


Fig. 9. Image texture synthesis using optimization method. (a) The image exemplar (128×128) is synthesized to a larger texture (256×256), (b) time complexity comparison for patch with different size used in TSVQ [4], and ANN [10], ANN working on a subset of patches in the texture being synthesized, (c) time complexity comparison for ANN working on a subset of patches in the texture being synthesized, FFT method on CPU, FFT method on GPU, our N columns acceleration on CPU, and our N columns acceleration on GPU, (d) memory requirement comparison.

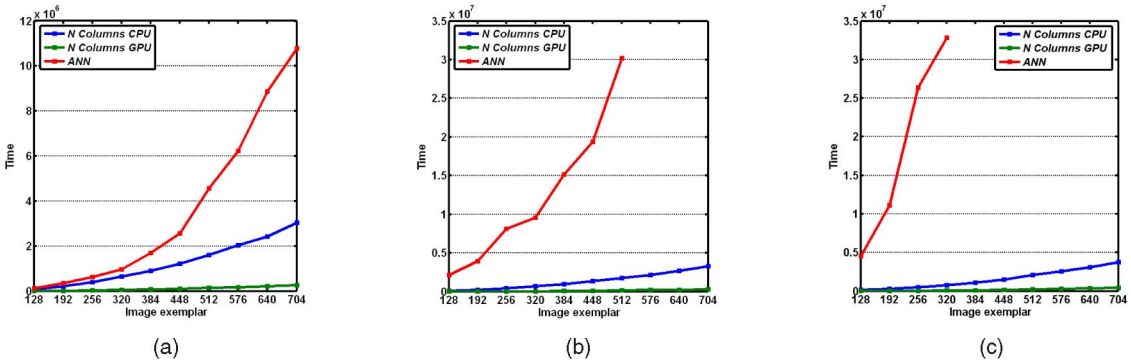


Fig. 10. Time complexity comparison with ANN [10] for synthesizing a texture (512×512) from increasingly larger exemplar images (from 128×128 to 704×704). Three different kinds of patch size are used: (a) patch size is 4×4, (b) patch size is 8×8, and (c) patch size is 16×16.

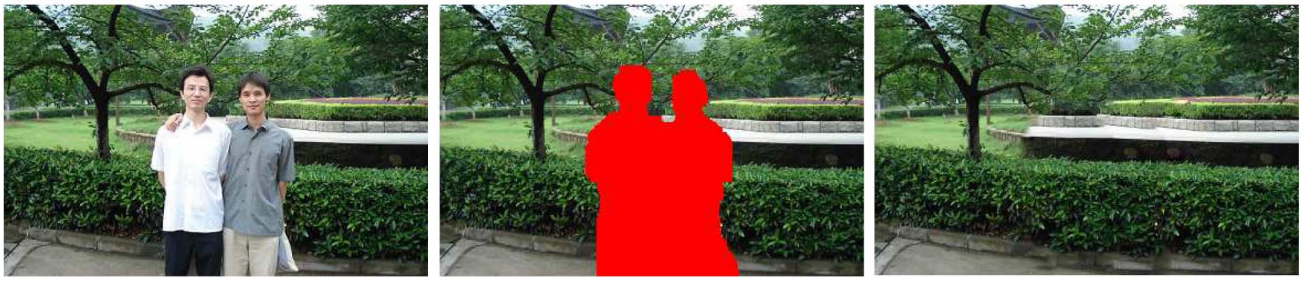
TABLE 1
Performance Comparison on the Texture Synthesis (Milliseconds) Based on Different Patch Sizes for TSVQ, ANN, ANN Working on a Subset of Patches in the Texture Being Synthesized, FFT Method on CPU, FFT Method on GPU, Our N Columns Acceleration on CPU, and Our N Columns Acceleration on GPU

Methods\PatchSize	8	16	24	32	40	48	56
TSVQ	545685	1425640	2765320	3965474	5137210	6015213	6863240
ANN	503869	1296890	2330700	3467480	4593970	5399590	6063950
ANN(Subset)	125967	81055	64741	54179	45939	37497	23687
FFT(CPU)	87625	88364	89137	89945	90463	91278	92146
FFT(GPU)	25035	25796	26238	26574	27259	28032	28763
Our method (CPU)	29987	41393	62221	83834	107344	133279	161166
Our method (GPU)	2103	2594	3218	3405	6247	7741	8461

($w = 4$ in our experiments and r is the width of each neighborhood), our complete search method is still much faster than ANN and TSVQ. Note that for extreme case when patch size $r = 56$, if we set $w = 4$, our complete nearest search method does not show much better performance advantage compared to TSVQ and ANN. It is because in this case, for TSVQ and ANN, only $1/296$ of the patches in the X are needed to be searched for finding the nearest patches. However, in most texture synthesis applications, the preferred patch size is much smaller and our method is fully capable to handle it with the best performance.

We also give the performance comparison with FFT method [27] on both CPU and GPU. As shown in Fig. 9 and Table 1, when performing the nearest patch matching on CPU, FFT method is slower than our method when the patch size is not large ($r < 32$), and is faster when the patch size is large ($r > 32$). This happens because when processing a moderate size image with a large patch size, the number of neighboring patches becomes smaller too. Hence, our method cannot make the best of the sequential overlap between patches. When performing on GPU, our method is much faster for both large and small patch sizes and more experimental data are presented in Table 1. Furthermore, our method guarantees to find the exact nearest patch, while FFT method [27] may lead to a nonoptimal match due to the round-off error produced by the many computations used in the convolution sum.

We also give comparison results with ANN using increasingly larger search space for different patch sizes. For a given epsilon value ε and a fixed patch dimension d, ANN has a single-query complexity of $O(c_{d,\varepsilon} log(n))$ [10], where n is the number of patches in the exemplar image,



(a) (b) (c)

Fig. 11. Image completion using optimization, (a) original image (460×346), (b) masked image, (c) the completion result.

(a) (b) (c)

Fig. 12. Video completion. (a) The input video ($320 \times 130 \times 150$), (b) the masked video, (c) completed video. The 122th frame of the video is illustrated.

and $c_{d,\epsilon}$ is a factor depending on ε and d. If k patches are queried (say, the k patches of synthesized texture), the overall complexity is $O(c_{d,\epsilon}klog(n))$. The proposed algorithm has a complexity of $O(kn)$ (the naive algorithm is $O(knr^2)$ where r is the image patch size). In Fig. 10, we present time complexity comparisons with ANN [10] for synthesizing a texture (512×512) from increasingly larger exemplar images (from 128×128 to 704×704). We present comparison results for three different patch sizes: 4×4, 8×8, and 16×16. The largest input exemplar image presented is 704×704. As for a much larger image, the memory requirement for building the tree structure goes beyond 2 GB RAM. The results in Fig. 10 show that our method (on both CPU and GPU) is faster than ANN. Although the overall trend in complexity shows that ANN may become faster for very large n (for example, $8{,}192 \times 8{,}192$). However, for such a large image, the memory requirement for building tree structure becomes a prohibitive bottleneck. It should be pointed out that, although our method is effective for nearest patch search in texture synthesis, however, the approximate methods such as ANN and k-nearest matching perform very well in terms of speed and quality, and are powerful methods for nearest patch search.

6.4 Image and Video Completion

Compared with the example-based texture synthesis, the patch-based image completion [34] typically involves a large input image so that the matching problem is even more time critical. As an example, in order to complete the missing region H in an image S with some new data H^* such that the resulting image S^* will have a high global visual coherence with some reference image D. Typically, $D = S \backslash H$, which is the remaining image portions outside the hole, is used to fill the hole. Therefore, our target is seeking a patch set S^* which maximizes the following objective function [34]:

$$Coherence(S^*|D) = \prod_{p \in S^*} \max_{q \subset D} sim(W_p, V_q), \quad (1)$$

where p, q run over all points in their respective sequences. $sim(.,.)$ is a local similarity measure between two patches W_p and V_q. We have to find a nearest patch V_q in D for each patch W_p in the hole H.

In Fig. 11, we present a large image completion example using the optimization-based methods [34]. For the special case of image completion, since both the "hole" region and the reference region that is used as the exemplar within the image texture are not regular, we use hybrid method to accelerate the patch matching. We employ not only the N columns matching, but also the single-column matching techniques. In Fig. 11, a patch with size 25×25 is used in the completion process to preserve the large structures of the image. It takes about 15 seconds on CPU for one complete nearest patch matching. We also show video completion results in Fig. 12, which are based on the nearest 3D patch search method (75 seconds on GPU for one complete nearest patch search). Similar to [34], we adopt the motion information for better completion.

6.5 Visual Data Summarizing

Simakov et al. [8] propose an approach for the summarization (or retargeting) of visual data (images or video) based on the optimization of a well-defined bidirectional similarity measure. Two signals Z (input Source signal) and X (output Target signal) are considered to be "visually similar" if as many as possible patches of Z are contained in X, and vice versa. For every patch $Q \subset X$, we need to search for the most similar patch $P \subset Z$, and compute the patch distances, and vice versa.

The nearest patch matching dominates the efficiency of the data summarization processing. We compare our performance and summarization results with the randomized correspondence algorithm [12]. For one complete nearest

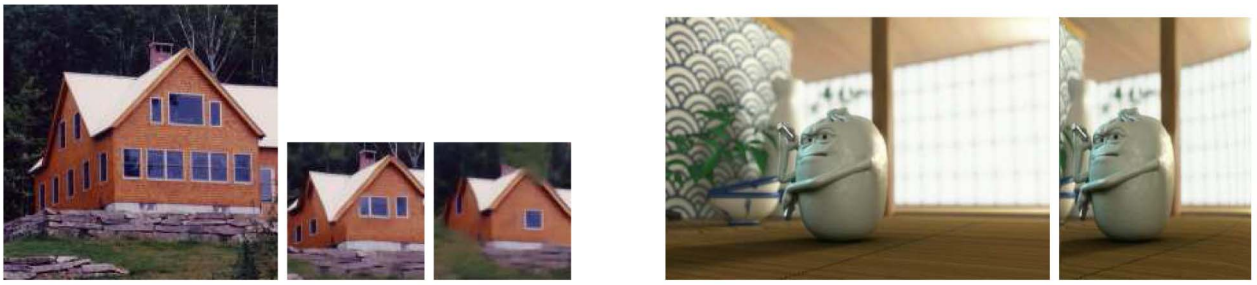


Fig. 13. Image summarizing. (a) The input image (300×300), (b) the input image (a) is summarized to image (150×150) using the proposed nearest neighbor search method, (c) summarization result using the randomized correspondence algorithm [12]. (d) The input video ($352 \times 240 \times 52$), (e) summarized video ($176 \times 240 \times 52$).

patch search using CPU implementation, it takes 4 seconds for our method, and only 1 second for the randomized correspondence method [12] since it depends on the local search scope. However, the randomized algorithm [12] may cause the optimization function to return a local minimal solution. Moreover, using the method [12], nearest patches are found based on the local image coherence information. Hence, the editing results depend heavily on the initialization of the nearest-neighbor field. Using our exact nearest patch search, the editing results do not depend on a good initialization. As illustrated in Fig. 13, when using the same initialization of the nearest patch field, our method generates more convincing result compared with the randomized correspondence method [12].

The video summarization can be done by computing the bidirectional similarity between the source video and target video [8]. Instead of the 2D patch used in image summarization, in the video case, we use 3D space-time patch. In Fig. 13, the video summarization results using our fast nearest 3D patch search are given. It only takes our method about 2.5 minutes on GPU for one complete nearest patch search in this example.

7 Limitations

Our fast nearest patch search highly depends on the overlapping patches of the continuous input data (image and video) to eliminate the redundant computations. However, when there is no sequential overlap between the neighborhoods of the input data, our method cannot work efficiently. Furthermore, for some applications, like texture synthesis, which do not require the exact patch matching, the approximate method such as ANN [10] may achieve faster results by incorporating PCA techniques. Although our method for video is significantly faster, to process extremely large and long video sequence with a large patch size, the efficiency of our method has to be further improved for the interactive video processing and editing. However, with the rapid development of the graphics hardware, such an acceleration could be achieved in the near future.

8 Conclusion

In this paper, we proposed a novel fast exact nearest patch matching method for image processing and editing. In contrast to most widely used algorithms, our method does not require the reconstruction of any hierarchical data structure. The key idea is to eliminate the redundant matching computations of the adjacent overlapped patches, which results in a constant complexity for the patch similarity matching. Furthermore, we present the GPU-accelerated version of the proposed method, which further improves the performance by at least an order of magnitude. To our knowledge, our algorithm is the most efficient exact approach for the nearest patch matching among the existing methods. In addition, its memory requirement is minimal. We applied our nearest exact patch matching method in several practical image/video applications and all the experimental results are very convincing and promising.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and insightful suggestions. This work was partly supported by NSFC (No. 60803081, No. 61070081), National High Technology Research and Development Program of China (863 Program) (No. 2008AA121603), the Fundamental Research Funds for the Central Universities (6081005), and State Key Lab of CAD&CG (No. A0808).

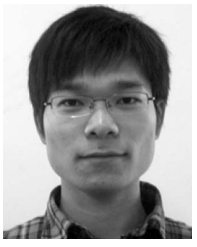
References

[1] D. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *Int'l J. Computer Vision*, vol. 60, no. 2, pp. 91-110, 2004.
[2] J. Sivic and A. Zisserman, "Video Google: A Text Retrieval Approach to Object Matching in Videos," *Proc. Ninth IEEE Int'l Conf. Computer Vision (ICCV)*, vol. 2, pp. 1470-1477, 2003.
[3] A. Efros and W. Freeman, "Image Quilting for Texture Synthesis and Transfer," *Proc. ACM SIGGRAPH*, pp. 341-346, 2001.
[4] L.-Y. Wei and M. Levoy, "Fast Texture Synthesis Using Tree-Structured Vector Quantization," *Proc. ACM SIGGRAPH*, pp. 479-488, 2000.
[5] J. Kopf, C.W. Fu, D. Cohen-Or, O. Deussen, D. Lischinski, and T.T. Wong, "Solid Texture Synthesis from 2D Exemplars," *ACM Trans. Graphics*, vol. 26, no. 3, pp. 21-29, 2007.
[6] W. Freeman, T. Jones, and E. Pasztor, "Example-Based Super-Resolution," *IEEE Computer Graphics and Applications*, vol. 22, no. 2, pp. 56-65, Mar./Apr. 2002.
[7] A. Buades, B. Coll, and J. Morel, "A Non-Local Algorithm for Image Denoising," *Proc. IEEE CS Conf. Computer Vision and Pattern Recognition (CVPR)*, vol. 2, p. 60, 2005.
[8] D. Simakov, Y. Caspi, E. Shechtman, and M. Irani, "Summarizing Visual Data Using Bidirectional Similarity," *Proc. IEEE CS Conf. Computer Vision and Pattern Recognition (CVPR)*, pp. 1-8, 2008.

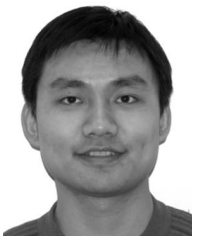
[9] J. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM,* vol. 18, no. 9, pp. 509-517, 1975.
[10] S. Arya, D.M. Mount, N.S. Netanyahu, and R. Silverman, and A. Wu, "An Optimal Algorithm for Approximate Nearest Neighbor Searching," *J. ACM,* vol. 45, no. 6, pp. 891-923, 1998.
[11] M. Ashikhmin, "Synthesizing Natural Textures," *Proc. ACM Symp. Interactive 3D Graphics (I3D),* pp. 217-226, 2001.
[12] C. Barnes, E. Shechtman, A. Finkelstein, and D. Goldman, "Patchmatch: A Randomized Correspondence Algorithm for Structural Image Editing," *Proc. ACM SIGGRAPH,* 2009.
[13] X. Tong, J. Zhang, L. Liu, X. Wang, B. Guo, and H. Shum, "Synthesis of Bidirectional Texture Functions on Arbitrary Surfaces," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 665-672, 2002.
[14] J. Hays and A.A. Efros, "Scene Completion Using Millions of Photographs," *Proc. ACM SIGGRAPH,* 2007.
[15] T. Huang, *Two-Dimensional Signal Processing II: Transforms and Median Filters,* Springer-Verlag, 1981.
[16] B. Weiss, "Fast Median and Bilateral Filtering," *ACM Trans. Graphics,* vol. 25, pp. 519-526, 2006.
[17] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," *Proc. 30th Ann. ACM Symp. Theory of Computing,* pp. 604-613, 1998.
[18] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín, "Searching in Metric Spaces," *ACM Computing Surveys,* vol. 33, no. 3, pp. 273-321, 2001.
[19] G. Shakhnarovich, T. Darrell, and P. Indyk, *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice.* MIT Press, 2005.
[20] N. Kumar, L. Zhang, and S. Nayar, "What is a Good Nearest Neighbors Algorithm for Finding Similar Patches in Images?" *Proc. European Conf. Computer Vision (ECCV),* pp. 364-378, 2008.
[21] R. Sproull, "Refinements to Nearest-Neighbor Searching Ink-Dimensional Trees," *Algorithmica,* vol. 6, no. 1, pp. 579-589, 1991.
[22] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," *Proc. Symp. Math. Statistics and Probability,* pp. 281-297, 1967.
[23] Y. Linde, A. Buzo, and R. Gray, "An Algorithm for Vector Quantizer Design," *IEEE Trans. Comm.,* vol. 28, no. 1, pp. 84-95, Jan. 1980.
[24] P. Yianilos, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces," *Proc. Fourth Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 311-321, 1993.
[25] C. Xiao and M. Liu, "Efficient Mean-Shift Clustering Using Gaussian KD-Tree," *Computer Graphics Forum,* vol. 29, no. 7, pp. 2065-2074, 2010.
[26] S. Lefebvre and H. Hoppe, "Appearance-Space Texture Synthesis," *ACM Trans. Graphics,* vol. 25, no. 3, pp. 541-548, 2006.
[27] S. Kilthau, M. Drew, and T. Moller, "Full Search Content Independent Block Matching Based on the Fast Fourier Transform," *Proc. Int'l Conf. Image Processing (ICIP),* vol. 1, pp. 669-672, 2002.
[28] C. Soler, M. Cani, and A. Angelidis, "Hierarchical Pattern Mapping," *Proc. ACM SIGGRAPH,* pp. 673-680, 2002.
[29] F. Crow, "Summed-Area Tables for Texture Mapping," *Computer Graphics,* vol. 18, no. 3, pp. 207-212, 1984.
[30] A. Rivers and D. James, "FastLSM: Fast Lattice Shape Matching for Robust Real-Time Deformation," *ACM Trans. Graphics,* vol. 26, no. 3, 2007.
[31] A. Lefohn, M. Houston, C. Boyd, K. Fatahalian, T. Forsyth, D. Luebke, and J. Owens, "Beyond Programmable Shading: Fundamentals," *Proc. ACM SIGGRAPH: ACM SIGGRAPH '08 classes,* pp. 1-21, 2008.
[32] C. NVIDIA, "Compute Unified Device Architecture Programming Guide, Version 2.2," NVIDIA, 2009.
[33] V. Kwatra, I. Essa, A. Bobick, and N. Kwatra, "Texture Optimization for Example-Based Synthesis," *Proc. ACM SIGGRAPH,* pp. 795-802, 2005.
[34] Y. Wexler, E. Shechtman, and M. Irani, "Space Time Completion of Video," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 29, no. 3, pp. 463-476, Mar. 2007.
[35] C. Xiao, Y. Nie, W. Hua, and W. Zheng, "Fast Multi-Scale Joint Bilateral Texture Upsampling," *The Visual Computer,* vol. 26, no. 4, pp. 263-275, 2010.

Chunxia Xiao received the BSc and MSc degrees from the Mathematics Department of Hunan Normal University in 1999 and 2002, respectively, and the PhD degree from the State Key Lab of CAD & CG of Zhejiang University in 2006. He is currently an associate professor at the School of Computer, Wuhan University, China. During October 2006 to April 2007, he worked as a postdoc at the Department of Computer Science and Engineering, the Hong Kong University of Science and Technology. His research interests include image and video precessing, digital geometry processing, and computational photography.

Meng Liu received the BS degree from the School of Computer, Wuhan University in 2009. He is currently a master student at School of Computer, Wuhan University, China. His research interests include image and video editing, GPGPU applications.



Yongwei Nie received the BS degree from the School of Computer, Wuhan University in 2009. He is currently a master student at School of Computer, Wuhan University, China. His research interest includes computational photography.



Zhao Dong received the BS and MS degrees from Zhejiang University in 2001 and 2005, respectively. He is currently a PhD candidate in the Computer Graphics Group of Max-Planck-Institut fuer Informatik, Germany. His research interests include real-time global illumination rendering, GPGPU applications, and volume graphics. He is a student member of the IEEE.