High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

Fast ANN for High-Quality Collaborative Filtering

Yun-Ta Tsai[1], Markus Steinberger[1,2], Dawid Pająk[1], and Kari Pulli[1]

[1]NVIDIA [2]Graz University of Technology

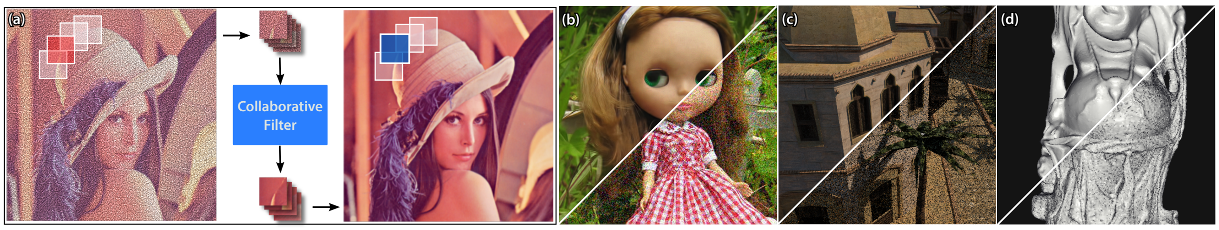

Figure 1: *Collaborative filtering is a powerful, yet computationally demanding denoising approach.* ***(a)*** *Relying on self-similarity in the input data, collaborative filtering requires the search for patches which are similar to reference patch (red). Applying a 3D transform or a weighted average operation on these patches, unwanted noise is removed and the filtered results are scattered back to all patch locations. Our method accelerates the process of searching for similar patches and facilitates high-quality collaborative filtering even on mobile devices. Application examples for collaborative filtering include (left: our output; right: noisy input)* ***(b)*** *denoising a burst image stack,* ***(c)*** *reconstruction of a global illumination scene, and* ***(d)*** *geometry reconstruction.*

Abstract

Collaborative filtering collects similar patches, jointly filters them, and scatters the output back to input patches; each pixel gets a contribution from each patch that overlaps with it, allowing signal reconstruction from highly corrupted data. Exploiting self-similarity, however, requires finding matching image patches, which is an expensive operation. We propose a GPU-friendly approximated-nearest-neighbor algorithm that produces high-quality results for any type of collaborative filter. We evaluate our ANN search against state-of-the-art ANN algorithms in several application domains. Our method is orders of magnitudes faster, yet provides similar or higher-quality results than the previous work.

Categories and Subject Descriptors (according to ACM CCS): I.4.3 [Image Processing and Computer Vision]: Enhancement—Filtering

1. Introduction

Noise removal [BCM05, DFKE06] is an important problem in application domains such as imaging, image synthesis, and geometry reconstruction (Fig. 1). A powerful approach to noise removal relies on self-similarity in the data. Exploiting self-similarity requires finding data points that should have similar values (pixels in 2D images, 3D points in surface scans). This matching is often done by considering an image patch, which gives more context than a single pixel and makes finding the correct matches more robust. Overlapping patches also facilitate collaborative filtering: if the image patches are, for example, of size 8×8, each pixel is part of 64 different patches, and if all those are filtered separately, each pixel receives 64 different results. These 64 results can further be filtered or averaged to obtain strongly denoised estimates. Similar patches could be found from nearby regions in the same image, or in a time sequence, from different images. It is often desirable to find several matching patches instead of finding just the single, best match. This problem can be formulated so that the patch is interpreted as a high-dimensional vector (e.g., 64D for 8×8 patches), and the k closest vectors are found in a k-nearest-neighbor search. Relaxing the problem by requiring only approximate matches allows significant speed-ups, at only a negligible cost on the denoising performance. This leads to a class of algorithms called the approximate-nearest-neighbor, or ANN algorithms.

DOI: 10.2312/hpg.20141094

Many techniques that accelerate ANN search have been proposed. Examples include random KD-trees, K-means clustering, Local Sensitive Hashing, and Principal Component Analysis. While being efficient, the majority of these solutions suffers from a) the curse of dimensionality, where high-dimensional data becomes very sparse and the distance metric loses its discrimination power, b) limited accuracy, reducing the quality of the matches and the filtering result, c) high pre-processing cost, which prohibits the use in interactive applications, and d) poor performance scaling on massively parallel systems, such as GPUs. Importantly, the search for the best matches is completely separated from collaborative filtering, leading to inefficient implementations.

We present an approximate-nearest-neighbor search method that is optimized for both collaborative filtering and an efficient implementation on the GPU. Whereas many other methods first construct a search structure and then repeatedly use it to perform search and filtering, we in essence perform all the queries in parallel as we construct the search structure. The method is general and can be used in different denoising algorithms. We demonstrate the use of our method for 2D image denoising, both, using a single image and denoising an image burst. Furthermore, we show how it can be used to filter the output of ray-traced renderings, and to denoise surfaces recorded with 3D range scanners.

2. Related Work

The key enabler of a collaborative filtering [BCM05, DFKE06] is a fast nearest-neighborhood (NN) search; therefore, we focus the discussion here on various NN methods.

The *KD-tree* is the most widely-used family of algorithms for accelerated NN search [Ben75]. It is very effective for exact search when the data dimensionality is low. For high-dimensional data, several approximations exist.

Randomized KD-trees have been used to look up image features in very large image recognition problems [PCI*07, SAH08]. To avoid excessive backtracking when searching for neighboring elements, dynamically-built priority queues can be used [AM93, BL97]. Randomized KD-trees address this issue by splitting the data among multiple KD-trees generated from randomized overlapping subsets of the data. The trees are smaller and can be searched concurrently, with less backtracking. Pre-processing becomes more expensive, as the data must first be analyzed with PCA.

There are other methods for mitigating the cost of backtracking. One approach utilizes spatial coherency to *propagate matches* [OA12, HS12]. If the best candidates for a patch have already been found, and a new search is done for a nearby patch, the good matches found previously can be propagated to help the current search. The bookkeeping adds some overhead, however. *Gaussian KD-trees* [AGDL09] sparsely represent distributions in high-dimensional space. They support spatio-temporal filtering, exploit the commonalities between bilateral, non-local means, and other related filters based on an assumption of Gaussian distributions, and can be implemented efficiently on the GPU. The key difference with respect to a regular KD-tree is that, in addition to the splitting value, it also stores the minimum and the maximum of the data projected onto the cut axis. During search, this can be used to skip branches that are likely to only have few samples. Such design elegantly integrates filtering and NN search into a single data structure. However, it also limits the types of supported filters, whereas our method can work with any type of transformations.

Clustering trees use a different choice for defining how the tree should branch. Fukunaga and Narendra [FN75] proposed *K-means trees*, where a tree structure is constructed via K-means, recursively at each level clustering the data points into k disjoint groups. The trees are constructed by *hierarchical clustering* [ML12], where the branching factor k determines whether a flat or deep tree is built. For clustering, a simple random selection of k points is used. To improve search performance, multiple trees can be built in parallel. Nistér and Stewénius [NS06] proposed to construct trees in the metric space. An advantage of using K-means is its efficiency of clustering. However, the centroid can be easily influenced by outliers. We use FLANN implementation [ML] as our benchmark k-means tree implementation. K-means trees can also be combined with KD-trees to boost search performance [ML09]. This approach has been successfully adopted to noise reduction [BKC08], where Brox et al. perform a recursive k-means clustering with $k = 2$ splits in each node. To increase precision, patches within distance w of a decision boundary are assigned to both sets, which increases the memory footprint and complicates data management.

Vantage point trees [Yia93] split points using the absolute distance from a single center, instead of partitioning points on the basis of relative distance to multiple centers. The number and thickness of these so-called "hypershells" can also be chosen in various ways to improve performance in image processing applications [KZN08].

Locality sensitive hashing [GIM99] is an efficient method for NN search on binary features. Zitnick [Zit10] proposed a similar method using *mini-hash* for the same purpose. While a binary descriptor has a small memory footprint and the Hamming distance can be used as an efficient metric, a fairly large support is required to have enough discriminative power, which makes collaborative filtering more costly.

To accelerate search, *PatchMatch* [BSFG09] uses a random NN search where neighboring patches propagate good matches. The *generalized PatchMatch* [BSGF10] further improves this search strategy to support kNN queries. To avoid brute force search, *PatchGP* [CKYY13], an extension to *pixel geodesic paths* [BS07], only checks subsets of path directions. As the distance measure on these subsets can be unreliable due to noise, PatchGP uses customized multi-scale filters to achieve good denoising results.

Many GPU-accelerated nearest-neighborhood techniques have been proposed. For instance, redundant norm computations can be minimized by exploiting an overlap between search windows [XLYD11], or accelerated insertion sort [GDNB10]. To reduce the search space, the data can also be partitioned into a set of randomly overlapping spheres [Cay10].

All these algorithms suffer from one or more of the following problems:

- complex data structures are needed for managing nodes and search,
- dimensionality reduction lowers filtering quality,
- costly pre-processing is required,
- multiple levels of indirection are not well suited for current GPU architectures (*pointer chasing*),
- suboptimal support for collaborative filtering,
- unreliability with high noise.

Our method addresses all of these problems and we demonstrate its benefits in multiple applications.

Several GPU optimizations techniques inspired our approach. Work queues are an efficient way to recursively construct tree structures on the GPU [CT08, GPM11]. Task descriptors for each are kept in a queue in GPU memory. Threads are running in a loop and consistently draw tasks from the queue until all tasks have been processed. After processing a node, tasks for the child nodes can be added back to the queue. This way of looping until all tasks have been completed is called a *persistent threads approach* [AL09].

3. ANN for Collaborative Filtering

There are several criteria our ANN method has to fulfill. It should work on images (both 2D color images and 3D range images or meshes), and be able to handle fairly large patch sizes (e.g., 8×8). Furthermore, the entire method, i.e., the search structure construction, the search, and the filtering, needs to be fast. Consequently, the method has to map well on the GPU to benefit from its massive parallelism.

We also want to take advantage of the characteristics of collaborative filtering and known properties of the input data. For example, natural images tend to be locally coherent, both spatially, and temporally in case of a video or an image burst. Thus, it is typically sufficient to search for similar image patches in a close proximity [Leb12, BCM05], as a full search over the whole image yields a small quality improvement (see Table 1) with a huge increase in the execution time. It is also known that a relatively small number of similar patches/candidates (e.g., $k = 16$) is sufficient [Leb12], and that even this small query can be approximate.

We exploit these characteristics and limit the search space by dividing the image into a set of tiles (see Fig. 2). As we demonstrate in Table 1, for collaborative filtering applications, the tiled search performs almost as well as a full global or

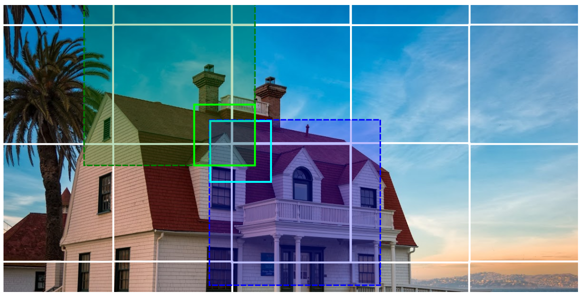

Figure 2: *The implementation of tiled collaborative filtering. The input image is divided into non-overlapping tiles, each $n \times n$ pixels large. Since each patch is centered around certain pixel within a tile, patches that are close to border of neighboring tiles (green and blue in the example figure) overlap and contribute to each others filtering results.*

Search Scope	PSNR [dB]
Global	28.44
Sliding window (15×15)	28.43
Tile (15×15)	28.19

Table 1: *Comparison of non-local means (NLM) filtering for the BM3D dataset [DFKE06] using three different search scopes. Each image is corrupted with zero-mean additive Gausian noise with $\sigma = 20$, PSNR = 22.12dB; patch size 8×8 and number of candidates $k = 16$.* Global *covers the whole image,* Sliding window *uses a symmetric search window around each patch's center, and* Tile *divides the image into non-overlapping tiles within which we look for the patch matches.*

symmetrically centered search. This is because the patches on the border of neighboring tiles overlap, and therefore contribute to each others filtering results.

To improve the query performance, we pre-cluster patches in a tile so that similar patches are grouped together. While previous methods first construct a search acceleration structure and then repeatedly use the structure to perform a search, we fuse the data structure construction and search, achieving a significant speed-up.

Below we discuss the main steps of the proposed algorithm: building the cluster list, using it to perform the ANN query, and collaborative filtering.

3.1. Pre-clustering Patches within a Tile

We first divide an image into tiles as shown in Fig. 2. Each tile is processed independently during query, but due to the collaborative filtering, the outputs will overlap. A larger tile allows finding better matches, while a smaller tile fits better into the cache or shared memory, maximizing memory locality for each query, as shown in Table 2.

Our preferred setup uses 15×15 tiles (with 225 potential matches) and 8×8 patches. This patch size is a common choice, as it is large enough to be robust to noise, and small enough for efficient processing [BCM05, DFKE06, Leb12].

Tile Size	Clustering [ms]	Query [ms]	PSNR [dB]
11×11	2.58	2.27	27.94
15×15	3.52	2.07	28.04
19×19	4.73	1.96	28.09

Table 2: *Performance of our method as a function of tile size (BM3D dataset [DFKE06], patch size 8×8, $k = 16$). Each image is corrupted with zero-mean additive Gausian noise with $\sigma = 20$ that yields PSNR = 22.12dB. For tiles larger than 15×15 pixels, the improvement in image quality becomes negligible.*

The patches are clustered hierarchically (see Fig. 3). At each step, the remaining patches (initially all the patches within the tile) are split into two clusters. This is implemented with a variant of K-means++ [AV07] algorithm, which we additionally modified to remove irregular workloads and pseudo-random memory access patterns. The new algorithm performs better on the GPU and is summarized below:

1. Choose the first patch in the node as the first center.
2. Compute ℓ^2 norm n_i between the first patch and every other patch and normalize the distances with $c_i = \frac{\sum_{j \le i} n_j}{\sum_i n_i}$.
3. The first patch with c_i greater than the threshold τ (we use $\tau = 0.5$) is selected as the second center.

To speed up the process, we only perform K-means on a subset of patches, e.g., 8, to find the cluster centers. This sub-sampling only slightly affects the clustering quality, but drastically reduces the computational load. Finally, we assign each patch to the closest of the two centers.

The clustering process continues recursively until the size of the cluster is below a threshold, which usually is twice the number of candidates required for filtering. For instance, for non-local means image denoising we use the top 16 matches for each patch, in which case we stop the recursion when the cluster is smaller than 32 patches.

3.2. Query and Candidate Refinement

After clustering, we can perform the NN query. Because similar patches are grouped within the same cluster, we do not need to perform a traditional tree traversal; instead, for each patch in the cluster we simply find its nearest neighbors by inspecting the patches in the cluster. If higher quality is required, we can search additional clusters. This increases the query time, which grows quadratically with the number of clusters being searched. However, we found that in most cases searching a single cluster is enough.

Figure 4 illustrates the parallel exhaustive search within a cluster. For each patch, we find the indices of the k nearest neighbors within the same cluster. We encode the indices of the nearest neighbors as a bit field, if the maximum number of elements in a cluster is 32, a 32-bit integer suffices. Replacing repeated tree searches with a simple cluster look-up results not only in a tremendous speed-up, but also allows us to efficiently implement collaborative filtering.

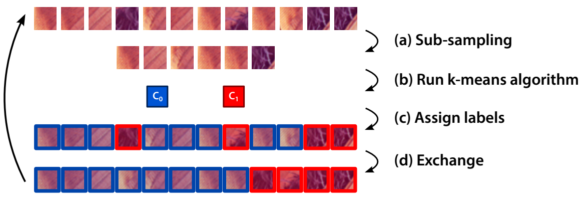

Figure 3: *Hierarchical clustering. **(a)** The input cluster is sub-sampled and used to **(b)** estimate two new cluster centers. **(c)** The patches in the input cluster are associated with the closest center, and then **(d)** reorganized to produce two new sub-clusters. The process is repeated until the output cluster size goes below a certain threshold.*

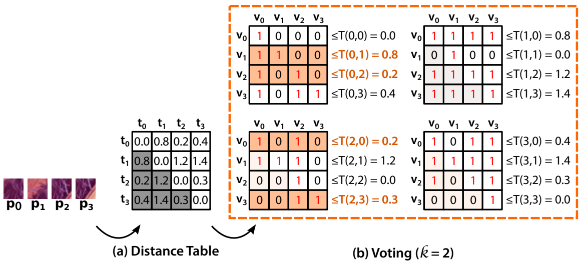

Figure 4: *Cluster-wide ANN lookup. Finding k nearest neighbors (here 2) for all patches in a cluster (here 4) is done in two steps. **(a)** We first compute a symmetric lookup table for all pair-wise distances $\delta_{i,j}$. **(b)** Next, for each patch, we find all patch partitions. The first (top-left) matrix compares distances $\delta_{i,j}$ to $\delta_{0,j}$, the second (top-right) to $\delta_{1,j}$, and so forth. We only need the rows (partitions) that have $k = 2$ ones (high-lighted in orange), the columns indicate the patches closest to the patch of the row. For example, the top-most orange row says that for patch p_1, the two closest patches are patches p_0 and p_1 (with a distance threshold = 0.8). We directly use the rows to binary-encode the nearest neighbors for each patch (e.g., for patch p_0 the encoding is 1010).*

3.3. Collaborative Filtering

After the candidate list is generated, we perform collaborative filtering in parallel for each cluster. For each patch, the nearest neighbors are fetched, the stack of matching patches is filtered, and the results are distributed to each participating patch in the output image. Since all patches within the same cluster are likely to have some common candidates, locality is maximized and computation can be drastically reduced.

4. Implementation

Our algorithm offers opportunities for extensive parallelization. First, each tile can be processed in parallel. Second, the individual splits during hierarchical clustering can be parallelized. Finally, candidates for each query can be determined in parallel. In a CPU implementation, this parallelism can be exploited in a multi-threaded implementation in conjunction with SSE vectorization. Using the available parallelism in a GPU implementation faces several additional challenges:

C1: Register pressure. Keeping a local copy of a single high-dimensional input vector may exceed the per-thread register file. Computations such as K-means ultimately lead to spilling registers to slower memory.

C2: Memory access patterns. The clustering algorithm groups unrelated patches to nearby memory locations, leading to inefficient, scattered memory access patterns.

C3: Thread divergence. The number of instructions executed for clustering depends on the data. Threads within the same warp but working on different nodes will show varying execution times and divergence hurts performance.

C4: Kernel launch overhead. Launching a kernel at each level of hierarchical clustering imposes a serious overhead. Determining efficient thread setups for unbalanced clusters adds another layer of complexity.

C5: Memory footprint. Computing and storing the candidates for all queries in parallel can result in serious memory and bandwidth requirements when storing the candidate information (particularly important on a mobile SoC).

We present next an efficient GPU implementation addressing all these challenges.

4.1. Clustering

The input data for our algorithm is given by high-dimensional patch data that usually surrounds the current pixel (image data) or the current vertex (3D mesh data). Extracting this patch data from the original input representation would significantly increase memory consumption as it duplicates the overlapping input data. Given that the following stage simply clusters similar patches without altering the patch data, we store and work on references (the pixel coordinates). This way, cache hit rates also increase as neighboring patches access overlapping regions. In video and image stack processing, the data reference can include the frame number; in mesh processing the vertex index can be used as reference.

The major workload of clustering is formed by the 2-means algorithm, which is repeatedly run to generate a hierarchical clustering. Binary clustering is an inherently diverging and irregular task, both at instruction level and in terms of memory. During clustering, distances between arbitrary patches may be computed. Clustering at thread level would impose several problems mentioned earlier (**C1-C4**).

To address these problems, we developed a warp-wide binary clustering algorithm based on shuffle instructions. Shuffle instructions permit exchange of a variable between threads of the same warp without use of shared memory. This allows us to keep only a subset of the high-dimensional data in each thread, reducing register usage. Furthermore, assigning successive dimensions to the individual threads in the warp automatically leads to good memory access patterns since the input dimensions sit next to each other in memory. Using multiple threads to split a single cluster (node) offers the opportunity to alter the roles of individual threads for the different steps of the k-means algorithm. Our warp-wide binary clustering works like this:

1. Each cluster is assigned a warp for splitting it, the first center is set.
2. For each sub-sampled patch in the cluster, the entire warp computes the distance to the first center by executing a parallel reduction using efficient shuffle instructions.
3. Each thread keeps one of the computed distances in a register; the warp computes a prefix sum of these distances to choose the second center.
4. All threads in the warp cooperatively run at most five iterations of the k-means algorithm. At each iteration, the two centers are updated, and the distances are re-computed using parallel reductions.
5. The entire warp determines for each patch the distance to both centers for re-assignment.
6. All threads run through the patch array from the front and back at the same time, marking non-fitting pairs for exchange. As soon as the number of pairs to exchange matches the warp size, all threads perform exchanges concurrently.

These steps address both **C1** and **C2**, and also avoid divergence (**C3**), as an entire warp works on the same problem. Also, the problem of low parallelism on the first levels of hierarchical clustering is reduced, as the number of threads working at each level is multiplied by the warp size.

The only remaining issues are the kernel launch overhead and thread setup when creating the hierarchy (**C4**). To mitigate these issues, we use a task queue [CT08] in combination with a persistent threads implementation [AL09]. A similar technique has been used to generate bounding volume hierarchies [GPM11]. In the task queue, we keep identifiers (*lowerIndex* and *upperIndex*) for each node that still needs to be split. Each worker warp draws such an identifier pair from the queue, splits the corresponding node, puts the identifier for one child back into the queue and starts working on the other child. In this way, only a single kernel launch is needed and nodes at different levels can be worked on concurrently.

The impact of these optimizations is shown in Table 3. Warp-wide execution clearly has the highest impact on performance, increasing execution speed by a factor of 40. Addi-

Strategy	Time [ms]	Speed-up
Naïve implementation	272.06	-
Warp-wide processing	6.54	41.62x
Persistent thread	4.84	56.16x
Parallel exchange	3.46	78.68x

Table 3: *Optimization strategies and speed-up for* clustering *for a* 0.25 *MPix image on a GTX 680.*

tionally avoiding the kernel launch overhead and working on nodes from multiple levels concurrently reduces the execution time by 26%. A further 29% reduction is due to the parallel exchange strategy. Overall, our optimizations reduced execution time by 98.8% compared to a naïve implementation.

4.2. Query

After clustering, similar patches are grouped in the same cluster. The next closest set of patches can be found in the adjacent clusters. This spatial relationship allows us to quickly retrieve potential candidates without costly traversal.

Considering **C1-C2**, we again perform warp-wide computations rather than using a single thread to select the candidates. To determine the candidates for an entire cluster, we use an entire block of threads. Each warp is then used to compute a set of inter-patch distances. Because the distance is symmetrical, we can pre-compute all the pair-wise distances within a cluster, and store them in shared memory, illustrated in Fig. 4. Each entry $T(i,j)$ stores the value of $\delta_{i,j}$ for patches P_i and P_j.

Once the matrix is computed, each warp is assigned to generate the candidates for a single patch P_s. Instead of sorting all candidates, we follow a voting scheme, which turned out to be nearly twice as fast as sorting: each patch P_i in the cluster is uniquely assigned to one of the threads in the warp. If the cluster size matches the warp size, every thread is responsible for a single patch. We then iteratively try to find the distance threshold λ w.r.t. P_s, which yields k candidates. Because all the possible thresholds are in the matrix, we only iterate over the stored distances. To compute the number of patches that fall within the threshold, we use *ballot* and *popc* instructions. This is the whole process:

1. Each thread block is assigned to a cluster.
2. Compute distance $\delta_{i,j}$ using warp-wide reduction and store the result in $T(i,j)$ and $T(j,i)$.
3. Each warp is assigned to determine the candidates for a single patch P_s.
4. Find at most k patches whose distance to P_s is less than or equal to λ iteratively via voting, where $\lambda = T(i,s)$.

In our algorithm, candidates are only searched for in the same cluster or within two neighboring clusters with additional expense of shared memory. Thus, all candidate patch references are close in memory after indexing. We can exploit this fact to reduce the memory requirements when encoding the candidates (**C5**). Instead of storing each individual candidate index, we only store the candidate index within the cluster using a bit field. This strategy allows us to use the result of the voting scheme (*ballot* instruction) directly to encode the candidates, reducing the memory requirement to as many bits as there are elements in a cluster.

Strategy	Time [ms]	Speed-up
Naïve implementaion	171.19	-
Warp-wide processing	10.85	15.78x
No tree	5.86	29.21x
Voting	3.33	51.41x
Compressed candandidates	3.17	54.00x

Table 4: *Optimization strategies and speedup for* query *for a* 0.25 *MPix image on a GTX 680.*

The impact of this optimization is shown in Table 4. Warp-wide execution again has the highest impact on performance, speeding up search by a factor of about 16. Avoiding the tree traversal nearly halves the execution time. Another 43% reduction is achieved by the voting scheme in comparison to sorting. Finally, the compressed candidate encoding reduces execution time by merely 5%. However, this optimization reduces the memory required for candidate encoding by one order of magnitude.

4.3. Filtering

While we only covered clustering and query in more detail, most of these techniques can also be used during the filtering stage that follows the query stage in most applications. When working with patch data, we again use an entire warp to work on a single patch to reduce register pressure and per-thread shared memory requirements. All optimizations reducing data load and store can also be used during filtering.

During collaborative filtering we take advantage of the grouping of similar patches. Often, steps in collaborative filtering, such as the transformation in BM3D filtering or the distance computations between patches in NLM, can be formulated as precomputations, In our filtering implementations, we start a block of threads for each cluster and run these precomputations only for the patches in that cluster. Intermediate results can be stored in fast local shared memory.

Our candidate encoding scheme allows further optimizations. In many cases, the same set of candidates is used for multiple patches in a cluster, i.e., if patch b and c are candidates for a, a and c are probably going to be candidates for b. Thus, we can run (at least some) computations only once for all patches that share the same candidate set and use the results for all patches. Due to the bitwise candidate encoding, we can efficiently find equal candidate sets using simple comparisons.

5. Evaluation

We compare our algorithm against other ANN methods, focusing on quality and performance. We break the evaluation into the following tests: a) nearest-neighbor query, b) image quality, and c) performance. For a fair comparison, we only select well-known algorithms that support kNN queries and work with different collaborative filters. If not specified differently, all tests work on 8×8 patches, and use $k = 16$.

Nearest-neighbor Query (NNQ) We use two metrics to evaluate the quality of the NNQ. First, we compute the overlap between the delivered k nearest neighbors with the ground truth determined via exhaustive search, i.e., how many candidates does the ANN method get right. Second, we compute the ratio between the sum of distances of the delivered candidate patches and the ground truth D_{ann}/D_{knn}, i.e., by how much does ANN increase the average patch distance. Table 5 shows the results for randomized KD-trees [PCI*07, SAH08], K-means trees [FN75], composite trees [ML], hierarchical clustering [ML09], generalized patch-match [BSGF10] (for meaningful comparisons, we only use translations, not scale or rotation), random ball cover (RBC) [Cay10] and our approach. We used the images from the BM3D dataset [DFKE06], and performed NNQ for each 8×8 patch within every 15×15 tile as our benchmark.

Method	% of correct	D_{ann}/D_{knn}
Randomized KD-trees	24.87	3.01
K-means	34.86	2.00
Composite	35.21	1.99
Hierarchical clustering	7.18	7.38
Generalized patch-match	0.22	23.91
RBC	97.88	1.01
Ours	39.01	1.32

Table 5: *Quality metrics for different ANN methods for the BM3D dataset. Our approach returns almost 40% of the nearest neighbors. The average distance of the patches returned by our method is 32% worse than the ground truth.*

Image Quality To evaluate the effects of ANN search on collaborative filtering, we ran patchwise non-local means filtering [BCM05] and BM3D filtering [DFKE06] on the dataset from Dabov et al. [DFKE06]. We added zero-mean additive Gaussian noise with $\sigma = 20/255$ to the 8-bit data values. We then ran the ANN algorithms on these input images in tiles and collaboratively filtered the returned candidates as in NNQ evaluation. The results are shown in Table 6. Our method is only slightly worse than RBC, and maintains the highest performance among all ANN methods with BM3D. Note that our method achieves a higher PSNR value than RBC for BM3D filtering. This is because our approximation is less likely to match noise to noise. We also tested with and without searching the neighboring two clusters, as mentioned in Sec. 3.2; the improvement was modest (0.1 dB), and the cost is quadratic. Thus, all the evaluations only consider a single cluster during query.

Method	NLM [dB]	BM3D [dB]
Randomized KD-trees	26.88	30.72
K-means	27.13	30.68
Composite	27.02	30.57
Hierarchical clustering	25.65	29.87
Generalized patch-match	21.24	28.92
RBC	27.83	30.71
Ours	27.79	31.05
Exhaustive Search (GT)	28.55	31.10

Table 6: *Average PSNR for 11 images [DFKE06] corrupted with zero-mean Gaussian noise with $\sigma = 20/255$. Patchwise NLM and BM3D filtering use different ANN methods.*

Performance As most ANN approaches require preprocessing, we measure and report the times for both clustering and query on an Intel i7-950 with 8GB of RAM and an NVIDIA Geforce GTX 680. The FLANN CPU implementations are optimized with multi-threading, and the window search uses SIMD (SSE2). The input resolution is 0.25 MP, $k = 16$.

The results of this test are shown in Table 7. The runtimes of all four FLANN implementations (KD-trees, K-means, composite, hierarchical clustering) are very similar. The time is split fairly evenly between preprocessing and query. All four CPU methods deliver their results in about a second, indicating that the implementations are very similar, only the clustering criteria change. Generalized patch-match does not do any pre-clustering, and, thus, takes nine times longer for queries. However, by using information from neighboring pixels to guide the query process, it is about four times faster than a brute force window search. Implementing the same brute force window search on the GPU, the entire query process is done in less than 600 ms, faster than any approach on the CPU. Applying the same optimization strategies for the window search as we used for our approach (warp-wide execution, voting instead of sorting), we could lower the execution time by 90%. However, our approach is still six times faster, completing clustering in 3.6 ms and query in 4.6 ms. Our approach takes less than 1% of the execution time of the fastest CPU implementation.

We compared our method against two other GPU-based ANN methods: kNN-Garcia [GDNB10], and RBC [Cay10]. Unlike our approach, these methods are not designed to work on small tiles and therefore struggle to perform in this mode (see Table 7). Disabling tiled processing significantly improves their run-time performance – kNN-Garcia took 590.73 ms to complete, and RBC finished in 2565 ms. These numbers, however, are still far from our results. Moreover, a large input patch-set greatly reduces the ANN quality. In case of RBC, disabling tiled processing caused ANN accuracy drop from 97.88% to 33.49%.

Method	Clustering	Query	Total [ms]
Randomized KD-trees	407	380	788
K-means	670	312	982
Composite	666	357	1024
Hierarchical clustering	415	601	1017
Generalized patch-match	0	8930	8930
Window search (CPU)	0	36700	36700
kNN-Garcia (GPU)	25466	398	26359
RBC (GPU)	10837	491	11328
Window search (GPU)	0	594.99	594.99
Window search (GPU opt)	0	48.3	48.3
Ours (GPU)	3.55	4.64	8.19

Table 7: *Run-time for different NN methods. Our method is significantly faster than other methods while still delivering high quality results.*

We also implemented our algorithm for different architectures. Running on the CPU (Core i7-950), it takes 130 ms to retrieve 16 candidates for a 0.25 MPix images. On a mobile GPU (Tegra K1), it takes 122.3 ms, which is even less than the desktop CPU version. Furthermore, our method has a small memory footprint. For a 0.25 MP image, we only require 5 MB of additional storage. As we can process the image in tiles, we can keep the memory requirement constant while supporting arbitrary image sizes. We can increase the number of concurrently processed tiles for future GPU architectures, which may require a higher workload.

6. Applications

We demonstrate our method for image processing, global illumination, and geometry refinement.

Single Frame Noise Reduction is the primary motivation for many collaborative filtering techniques. Many ANN and acceleration methods have been proposed for this domain, but they either have to rely on additional post-processing to improve the quality [CKYY13] or work only in conjunction with a limited number of filters [AGDL09]. Our method is independent of the choice of filters, while providing consistent quality without additional post-processing. In Table 8 we compare our method against the original CBM3D implementation using their dataset [DFKE06]. Again, we added zero-mean Gaussian noise with $\sigma = 20/255$ to all images. Both CBM3D and ours are configured to use a discrete cosine transform as the 2D transform, the Walsh-Hadamard transform in the third dimension, and operate in opponent color space. The only difference is that CBM3D uses a brute-force window search. Our candidate list encoding (Sec. 4.2) enables us to implement filtering very efficiently on the GPU. The results show that our method is very close to the original implementation, yet significantly faster.

	PSNR [dB]	Run-time [ms]
Input	18.58	-
CBM3D	30.44	812000
Our BM3D	30.34	703.18
Our NLM	25.75	39.17

Table 8: *Comparison between the original CBM3D implementation [DFKE06] and our GPU-enabled methods. While our BM3D implementation loses only 0.1 dB in terms of quality, it is more than 1000 times faster. Further improvement in run-time can be achieved by switching to a simpler NLM filter (at the cost of reduced denoising performance).*

Burst Noise Reduction Current digital cameras can operate in a *burst mode* where they quickly capture multiple frames. Simple accumulation of frames from such a burst stack can significantly reduce the noise and improve the overall signal-to-noise ratio. The upper bound of this improvement is proportional to $\sqrt{N}$, where N is the stack size. This approach, however, fails for scenes with motion, where naïve accumulation produces visible ghosting artifacts. To mitigate this issue, we perform single-frame denoising, but look for similar patches not only in the spatial, but also in the temporal neighborhood [DFE07]. This requires a slight modification of the clustering part of our algorithm, which now processes the data at a particular tile location from all frames conjointly. Then we perform non-local means filtering for each patch from the reference image. In Fig. 5 we compare our method to Gaussian KD-Trees [AGDL09], which support both burst noise reduction and GPU acceleration. For single frame denoising Gaussian KD-trees and our approach achieve similar PSNR values, while ours is more than 1500 times faster. For an entire burst stack, our implementation achieves a 3 dB better PSNR while being 2000 times faster. Denoising an entire burst stack is a difficult task for Gaussian KD-trees, as the data becomes high-dimensional and requires PCA preprocessing. As Gaussian KD-trees require multiple parameters and have a very long running time, tuning the approach for optimal image quality is a difficult process.

Global Illumination Many modern interactive global illumination techniques apply guided noise reduction on sparsely sampled indirect illumination [BEM11]. We verify the applicability of our ANN method by using the output from a direct illumination forward-rendering pipeline as guidance for performing nearest-neighbor query. Clustering is done on the guidance image only, using an 8×8 patch size. To enhance clustering stability in the shadow regions, we increased the ambient light in the scene. During query we operate on the guidance data, but return samples from the indirect illumination, then combine with direct illumination to generate the final result. Results are summarized in Fig. 6.

Geometry Denoising Range data produced by 3D scanners is usually noisy and requires post-processing [TL94]. Self-

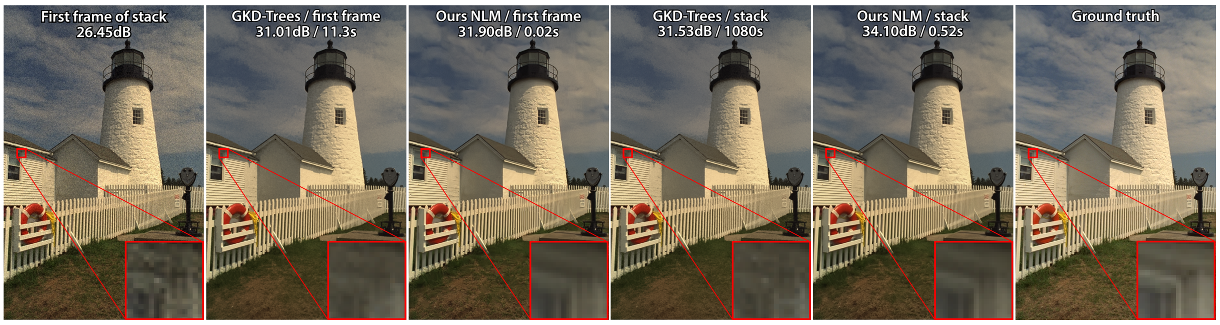

Figure 5: *Burst Image Denoising. The fence dataset (16 images) was corrupted with additive Gaussian noise of $\sigma = 12/255$. Each frame in the stack is 1.4 MP with random warping to simulate camera motion. Both Gaussian KD-Trees and ours run on the same GPU. Parameters for both methods are adjusted for the best image quality. We measured end-to-end processing time from clustering, query, and filtering on a GTX 680. Notice that our method significantly outperforms Gaussian KD-Trees in both cases.*

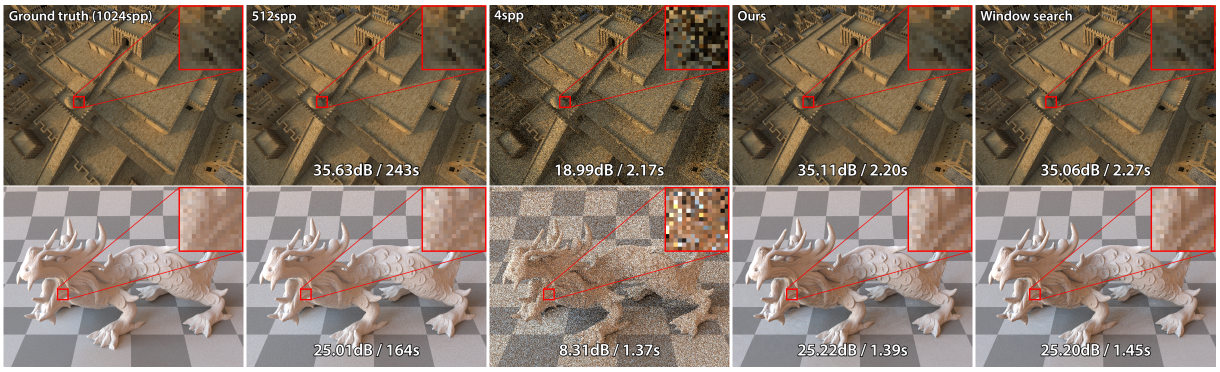

Figure 6: *Global Illumination Reconstruction. ANN methods can be used to speed up the filtering of noisy Monte-Carlo global illumination rendering. Our ANN method achieves nearly the same quality as window search. In terms of PSNR, both approaches are similar to a 512spp rendering.*

similarity in the scan data can also be used to reduce this noise. Gaussian KD-trees, in conjunction with NLM filtering, has been used for this task [AGDL09], extracting a detail layer of the mesh after applying Laplacian smoothing. To evaluate the suitability of our ANN algorithm, we replaced Gaussian KD-trees with our approach to generate filtering candidates. The results of this evaluation are shown in Fig. 7.

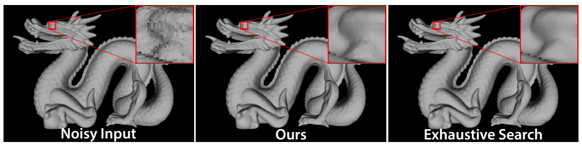

Figure 7: *Geometry denoising. Our ANN algorithm can also be used to find candidates for NLM filtering 3D meshes. Noisy input is generated with σ set to half of the average edge length. The reconstructions of our method and exhaustive search with $k = 256$ are visually indistinguishable.*

7. Summary and Conclusions

We have presented an ANN method building on the combination of tiling, hierarchical clustering using 2-means, and query within a single cluster. According to our evaluation, our approach can be used as input for high-quality, state-of-the-art collaborative filtering in multiple application domains, such as denoising, burst imaging, global illumination post-processing, and geometry reconstruction. While our approach hardly loses any quality in comparison to exhaustive search, it allows for many GPU optimizations.

Using warp-wide execution to work on a patch, avoiding kernel launches, and dynamically changing the work assignment for threads, results in speed-ups between 54× and 79× in comparison to a naïve GPU implementation. In comparison to state-of-the-art ANN methods, we achieve significantly better approximations to the ground truth exhaustive search while being up to 100 times faster. In comparison to Gaussian KD-trees [AGDL09] another GPU method, we are up to 2000× faster while achieving better image quality.

While our method is designed to work with any collaborative filtering approach, our implementation enforces some restrictions. Our implementation works very well in certain parameter ranges, e.g., patch size 4×4, 8×8, or 16×16. Parameter setups that conflict with the GPU warp size or require too much shared memory can reduce performance by up to an order of magnitude. In the future we want to explore the acceleration of complex computation chains where collaborative filtering is the bottleneck, such as end-to-end camera pipelines, video noise reduction, super-resolution, and image editing.

References

[AGDL09] ADAMS A., GELFAND N., DOLSON J., LEVOY M.: Gaussian kd-trees for fast high-dimensional filtering. *ACM Transactions on Graphics 28*, 3 (2009). 2, 8, 9

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *HPG* (2009). 3, 5

[AM93] ARYA S., MOUNT D.: Algorithms for fast vector quantization. In *Data Compression Conference* (1993). 2

[AV07] ARTHUR D., VASSILVITSKII S.: k-means++: The advantages of careful seeding. In *SODA* (2007). 4

[BCM05] BUADES A., COLL B., MOREL J.-M.: A non-local algorithm for image denoising. In *CVPR* (2005). 1, 2, 3, 4, 7

[BEM11] BAUSZAT P., EISEMANN M., MAGNOR M.: Guided image filtering for interactive high-quality global illumination. *Computer Graphics Forum 30*, 4 (2011). 8

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975). 2

[BKC08] BROX T., KLEINSCHMIDT O., CREMERS D.: Efficient nonlocal means for denoising of textural patterns. *IEEE Trans. Image Processing 17*, 7 (2008). 2

[BL97] BEIS J., LOWE D.: Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR* (1997). 2

[BS07] BAI X., SAPIRO G.: A geodesic framework for fast interactive image and video segmentation and matting. In *ICCV* (2007). 2

[BSFG09] BARNES C., SHECHTMAN E., FINKELSTEIN A., GOLDMAN D. B.: PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics 28*, 3 (2009). 2

[BSGF10] BARNES C., SHECHTMAN E., GOLDMAN D. B., FINKELSTEIN A.: The generalized PatchMatch correspondence algorithm. In *ECCV* (2010). 2, 7

[Cay10] CAYTON L.: A nearest neighbor data structure for graphics hardware. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures* (2010). 3, 7

[CKYY13] CHEN X., KANG S. B., YANG J., YU J.: Fast patch-based denoising using approximated patch geodesic paths. In *CVPR* (2013). 2, 8

[CT08] CEDERMAN D., TSIGAS P.: On dynamic load balancing on graphics processors. In *Symposium on Graphics Hardware* (2008). 3, 5

[DFE07] DABOV K., FOI A., EGIAZARIAN K.: Video denoising by sparse 3d transform-domain collaborative filtering. In *EUSIPCO* (2007). 8

[DFKE06] DABOV K., FOI A., KATKOVNIK V., EGIAZARIAN K.: Image denoising with block-matching and 3d filtering. In *SPIE Electronic Imaging* (2006). 1, 2, 3, 4, 7, 8

[FN75] FUKUNAGA K., NARENDRA P. M.: A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers C-24*, 7 (1975). 2, 7

[GDNB10] GARCIA V., DEBREUVE E., NIELSEN F., BARLAUD M.: K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *Image Processing (ICIP), 2010 17th IEEE International Conference on* (2010), IEEE, pp. 3757–3760. 3, 7

[GIM99] GIONIS A., INDYK P., MOTWANI R.: Similarity search in high dimensions via hashing. In *VLDB* (1999). 2

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster hlbvh with work queues. In *HPG* (2011). 3, 5

[HS12] HE K., SUN J.: Computing nearest-neighbor fields via propagation-assisted kd-trees. In *CVPR* (2012). 2

[KZN08] KUMAR N., ZHANG L., NAYAR S. K.: What is a good nearest neighbors algorithm for finding similar patches in images? In *ECCV* (2008). 2

[Leb12] LEBRUN M.: An analysis and implementation of the BM3D image denoising method. *Image Processing On Line 2* (2012), 175–213. `doi:10.5201/ipol.2012.l-bm3d`. 3, 4

[ML] MUJA M., LOWE D. G.: Flann - fast library for approximate nearest neighbors. `http://www.cs.ubc.ca/research/flann/`. 2, 7

[ML09] MUJA M., LOWE D. G.: Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP* (2009). 2, 7

[ML12] MUJA M., LOWE D. G.: Fast matching of binary features. In *Computer and Robot Vision* (2012). 2

[NS06] NISTÉR D., STEWÉNIUS H.: Scalable recognition with a vocabulary tree. In *CVPR* (2006). 2

[OA12] OLONETSKY I., AVIDAN S.: TreeCANN – k-d tree coherence approximate nearest neighbor algorithm. In *ECCV* (2012). 2

[PCI*07] PHILBIN J., CHUM O., ISARD M., SIVIC J., ZISSERMAN A.: Object retrieval with large vocabularies and fast spatial matching. In *CVPR* (2007). 2, 7

[SAH08] SILPA-ANAN C., HARTLEY R.: Optimised kd-trees for fast image descriptor matching. In *CVPR* (2008). 2, 7

[TL94] TURK G., LEVOY M.: Zippered polygon meshes from range images. In *SIGGRAPH* (1994). 8

[XLYD11] XIAO C., LIU M., YONGWEI N., DONG Z.: Fast exact nearest patch matching for patch-based image editing and processing. *IEEE Trans. on Visualization and Computer Graphics 17*, 8 (2011). 3

[Yia93] YIANILOS P. N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA* (1993). 2

[Zit10] ZITNICK C. L.: Binary coherent edge descriptors. In *ECCV* (2010). 2