Approximate Nearest-Neighbour Fields via Massively-Parallel Propagation-Assisted K-D Trees

Cosmin Eugen Oancea
Department of Computer Science (DIKU)
University of Copenhagen
Copenhagen, Denmark
cosmin.oancea@di.ku.dk

Ties Robroek
ties@trobroek.com

Fabian Gieseke
Department of Information Systems
University of Münster
Münster, Germany
fabian.gieseke@uni-muenster.de

***Abstract*—Nearest neighbour fields accurately and intuitively describe the transformation between two images and have been heavily used in computer vision. Generating such fields, however, is not an easy task due to the induced computational complexity, which quickly grows with the sizes of the images. Modern parallel devices such as graphics processing units depict a viable way of reducing the practical run time of such compute-intensive tasks. In this work, we propose a novel parallel implementation for one of the state-of-the-art methods for the computation of nearest neighbour fields, called propagation-assisted k-d trees. The resulting implementation yields valuable computational savings over a corresponding multi-core implementation. Additionally, it is tuned to consume only little additional memory and is, hence, capable of dealing with high-resolution image data, which is vital as image quality standards keep rising.**

***Index Terms*—nearest neighbour fields, propagation-assisted k-d trees, high-performance computing, computer vision.**

I. Introduction

Finding similarities between two images is a fundamental concept in computer vision, and have been used for a variety of tasks including, but not limited to, inpainting in videos [1], super-resolution [2], or denoising [3]. One way of expressing similarities is based on the concept of *nearest neighbour fields* (NNF) [4]. Due to the heavy computational workload, such fields are usually approximated—leading to *approximate nearest-neighbour fields* (ANNF)—which simply means that for every patch of a given image A we aim to find a patch in another image B, which approximates it well (visually).

In recent years, both the amount of images as well as the resolution of the images have dramatically increased. For this reason, the computation of ANNFs can quickly become a major computational bottleneck for today's data-intensive computer vision tasks. This has sparked the need and interest for the efficient computation of nearest neighbour fields. Various techniques exist to compute ANNFs. A well-known branch of methods is based on spatial search structures such as classical k-d trees [5] that can be used to accelerate the induced nearest neighbour queries. While such data structures often offer a way to significantly reduce the amount of comparisons that need to be conducted, the induced runtime is usually still problematic. For this reason, approximations have been proposed such as the popular *PatchMatch* [4] scheme or the so-called *coherency sensitive hashing* (CSH) framework [6], which yield good approximations based on spatial locality: Here, the main argument is that pixels/patches within a given image and also between two similar images are correlated (e.g., if the two images show an object that has only slightly changed its position). This contextual information can be exploited to efficiently "propagate" good nearest neighbour candidates during the computation of the ANNFs (e.g. row to row when computing nearest neighbours from top to bottom). Finally, He and Sun have proposed a method named *propagation-assisted k-d trees* [7] that combines the concepts of spatial search structure and propagation, by designing an algorithm that takes advantage of both the spatial subdivision induced by a k-d tree as well as of propagation.

The techniques mentioned above generally yield significant computational savings compared to standard (exact) schemes. However, modern computer vision tasks often require the computation of such fields for huge amounts of image pairs with images consisting of millions of pixels. Hence, the computation of ANNFs still remains a computationally challenging problem and subject to ongoing research. Highly-parallel devices such as modern graphics processing units (GPUs) have been used to accelerate compute-intensive tasks in a variety of fields. In this work, we provide a corresponding highly-parallel implementation for propagation-assisted k-d trees, which makes use of a propagation scheme tailored to modern GPUs. The resulting framework is up to one order of magnitude faster than an optimized multi-threaded implementation and provides valuable computational savings over other state-of-the-art competitors. In addition, it consumes a relatively small amount of memory, which enables it to process large image instances on cheap, commodity hardware.

II. Background

For the sake of completeness, we briefly sketch approximate nearest neighbour fields and massively-parallel k-d trees.

A. Approximate Nearest Neighbour Fields

Nearest neighbour search is the centrepiece for computing nearest neighbour fields, yet the adaptation to such fields in-

This research has been partially supported by the Independent Research Fund Denmark under the research projects "Monitoring Changes in Big Satellite Data via Massively Parallel AI" and "FUTHARK: Functional Technology for High-Performance Architectures".

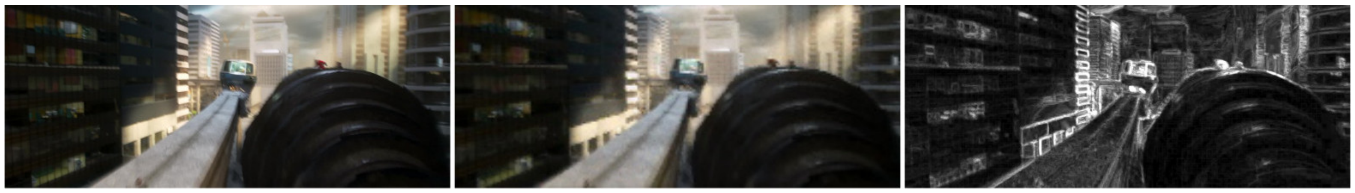


Fig. 1: Illustration of a nearest neighbour field: The left image depicts the first image A and the middle image its reconstruction based on the nearest neighbour indices given for the other image B. It can be seen that, given an appropriate NNF, one can successfully reconstruct the image A. The image on the right visualizes the Euclidean distances (i.e., last layer of the NNF).

troduces some particularities. A *nearest neighbour field* (NNF) captures the similarities between two images $A, B \in \mathbb{R}^{h \times w \times c}$ by storing the nearest neighbour (patch) in image B for every patch in image A [4]. More precisely, let $P_A \subset \mathbb{R}^d$ and $P_B \subset \mathbb{R}^d$ be the set of patches of image A and B, respectively, each containing $(h-p+1) \cdot (w-p+1)$ patches of dimensionality $d = p^2 c$, where p is the patch size (each patch is flattened). The nearest neighbour field NNF(A, B) of A and B contains, for each patch $q \in P_A$, its nearest neighbour $q' \in P_B$ w.r.t. some distance measure. Typically, the Euclidean distance/L2 distance is used in $\mathbb{R}^d$, i.e. $g(q, q') = \left(\sum_{i=1}^{d} (q_i - q'_i)^2\right)^{1/2}$.

The NNF has the same width and height as image A and stores, for each possible patch in image A (location of a patch center), the coordinates of its nearest neighbour patch in image B. Typically, three layers are used, where the first layer contains the x-coordinates, the second the y-coordinates, and the third layer the distances for the nearest neighbour patches. For instance, the NNF$_{[i,j,0]}$ contains the x-coordinate of the nearest neighbour patch in image B of the patch in image A with center (i, j). Correspondingly, NNF$_{[i,j,1]}$ stores the y-coordinate, and NNF$_{[i,j,2]}$ is the L2 distance between the two patches. Figure 1 shows that NNFs can be used to reconstruct an image by means of the recorded neighbours information.

To compute a NNF, the best nearest neighbour patch in image B has to be identified for each patch in image A. This usually requires the computation of millions of nearest neighbours in a relatively high-dimensional search space (e.g., $p = 8$ and $c = 3$ results in a $d = 192$-dimensional space). For this reason, searching for exact nearest neighbours tends to quickly become infeasible, especially if NNFs for many image pairs have to be computed. Hence, a common approach to reduce the computational complexity is to consider only approximate nearest neighbours, which has led to the concept of *approximate nearest neighbour fields* (ANNFs). For instance, a typical preprocessing step is to reduce the dimensionality of the patches by means of dimensionality reduction methods such as principal component analysis (PCA) [8].

Various approaches to compute ANNFs have been proposed in the literature. As already mentioned above, dimensionality reduction methods along with spatial search structures can be used to obtain approximate answers (e.g. via a k-d tree without backtracking, see below). In addition, schemes based on propagation have been proposed such as the aforementioned PatchMatch approach [4]. More recently, variants of PatchMatch such as CSH or propagation-assisted k-d trees have been used to even further reduce the practical runtime needed to compute ANNFs [6], [7]. In this work, we will provide an efficient massively-parallel implementation for the propagation-assisted k-d trees, which combine the idea of propagation (as employed by PatchMatch) with the concept of spatial pruning via k-d trees.

B. *Massively-Parallel K-D Trees*

A classical *k-d tree* for a point set $P = \{\mathbf{x_1}, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ is a balanced binary tree [5], [9], where the root of the tree corresponds to all points in P. The construction of the tree takes place in a level-wise manner. For a node v at level i, the point set P_v that corresponds to v is split into two (almost) equal-sized subsets, e.g., by using the median in dimension $i \bmod d$ or in the dimension i with widest spread (this scheme resorts to the latter scheme). The recursive construction of the tree ends as soon as any stopping criterion is fulfilled, such as $|P_v| < c$ for some constant $c \in \mathbb{N}$. The tree construction takes $\mathcal{O}(n \log n)$ time, since medians can be found in linear time.

For a given point $q \in \mathbb{R}^d$, one can use such a k-d tree to find the nearest neighbours by traversing the tree in two phases. First, the tree is traversed from top to bottom in order to find the box that (naturally) contains the point q. All points stored in that leaf are compared with the query point and the best k neighbours are stored. Second, the tree is fully traversed from bottom to top, by recursively visiting (checking) neighbouring boxes/subtrees, but only when the distance from the (query) point q to the box is smaller than the (worst) k's nearest neighbour candidate. The nearest neighbour set may be updated whenever a leaf is visited. Given relatively-low dimensional search spaces, k-d trees usually accelerate the search significantly (logarithmic runtime in practice). However, with increasing dimensionality (e.g. $d > 20$), more and more leaves of the tree have to traversed, which can yield a linear runtime per query in the worst case if all leaves have to checked (i.e. no better than a brute-force search).

Hence, given a large amount of queries and search spaces of increasing dimensionality, nearest neighbour search using k-d trees might still become a computational bottleneck. Modern parallel devices such as graphics processing units (GPUs) can help to accelerate the search again in such scenarios. These devices are nowadays also used to accelerate general compute- and data-intensive tasks, which is known as *general-purpose*

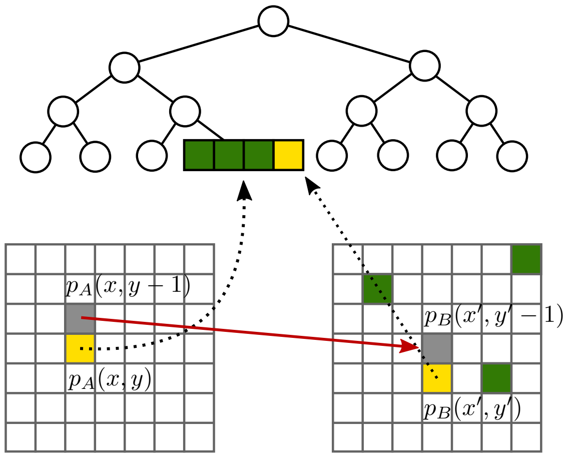


Fig. 2: Propagating patches [4], [7]

computing on graphics processing units (GPGPU). In contrast to multi-core processing units, which feature a relatively small number of complex units (e.g., 2-64), GPUs resort to thousands of simple processing units. Given appropriate implementations, GPUs often yield significantly smaller runtimes than their direct CPU-based competitors. In recent years, efficient GPU implementations have been proposed for a variety of tasks in data mining and machine learning [10]–[13].

Computing ANNFs is a special form of nearest neighbour search. In the past, GPUs have successfully been used for this general task with many approaches coming from the field of compute graphics [14]–[17]. For high-dimensional search spaces, brute-force schemes have been proposed [18], which offer great speed-ups over their multi-core competitors given medium-sized datasets. Other frameworks are based on locality-sensitive hashing [19] or adapted sorting schemes [20], [21]. In addition, implementations harnessing both the power of GPUs as well as spatial subdivisions induced by spatial search structures have been proposed [22]–[26]. Our work is inspired by the buffer k-d tree approach [12], which takes advantage of both k-d trees and massively-parallel computing and which is based on reorganizing the computational workflow into a memory-centric fashion that optimizes temporal locality [27].

III. Parallel Propagation-Assisted K-D Trees

We are now ready to outline the algorithmic buildings blocks of our parallel implementation. We start by providing the main ideas of propagation-assisted k-d trees [7].

A. Propagation-Assisted K-D Trees

Nearest neighbour search based on k-d trees induces a recursive traversal of subtrees of the tree (see above). In the worst case, all leaves will be visited by this traversal. A simple way to guarantee a logarithmic runtime per query is to stop the recursive traversal after a fixed number of leaves have been visited (e.g., 10). However, this modification also leads to approximate answers and restricting the number of leaf visits to a small number can cause a significant decrease in accuracy. Propagation-assisted k-d trees lessen the negative effects of such approximations by also resorting to the concept of propagation [7]. The intuition is that, once previous queries have found successful matches, future "close-by" queries can use this information to their advantage. Specifically, when looking for a nearest neighbour to a specific patch, previously investigated patches within the same area of the image have nearest neighbours that might provide helpful information on where to find good candidates for the current query patch. This modification is inspired by PatchMatch [4], [28], whose propagation scheme is also based on spatial locality, inferring strong candidates from previous searches.

More precisely, propagation-assisted k-d trees first compute the *exact* nearest neighbour (in the dimensionality-reduced space) for the first row of the query image A by applying a (full) k-d tree traversal. For the remaining patches, initial nearest-neighbour candidates are determined from the leaf that contains the query patch, and then the nearest neighbours are refined by selectively visiting new leaves inferred from the analysis of the previous row. Figure 2 demonstrates the propagation for a query patch $p_A(x, y)$. Since the patch $p_A(x, y-1)$ above $p_A(x, y)$ has already been processed, one has access to its nearest neighbour patch, denoted by patch $p_B(x', y'-1)$ in image B. The propagated leaf for patch $p_A(x, y)$ is the (k-d tree) leaf that contains the patch $p_B(x', y')$ (i.e., the patch immediately below $p_B(x', y'-1)$ in image B). Instead of propagating only the first nearest neighbour per patch, He and Sun [7] propagate k neighbours per query patch $p_A(x, y)$, which results in at most $k+1$ leaves to be checked. At the end, the best patch candidate is selected based on the original non-reduced patches (see below for the details).

Algorithm 1 Propagation-Assisted K-D Trees

```
Require: Two images $A, B \in \mathbb{R}^{h \times w \times c}$, patch size $p \in \mathbb{N}$,
    number $k$ of nearest neighbours
Ensure: Nearest neighbour field represented by nearest neighbour indices $I$.
1: $P_A, P_B$ = CreatePatches($A, B$)
2: $m$ = FitPCA($P_A, P_B, n_m$)
3: $r_A, r_B$ = ReduceDim($m, P_A, P_B$)
4: $\mathcal{T}$ = BuildKDTree($r_B$)
5: $(I_k, V_k)$ = SearchContainingLeaf($\mathcal{T}$, $k$, $r_A$)
6: $(I_k, V_k)$ = exactKnnFstRow($\mathcal{T}$, $k$, $I_k$, $V_k$, $r_A[0]$)
7: $(I_k, V_k)$ = ProcessRows($\mathcal{T}$, $k$, $I_k$, $V_k$, $r_A[1:]$)
8: $I$ = SelectBest($I_k, P_A, P_B$)
9: return $I$
```

B. Sequential Implementation

A sequential implementation for propagation-assisted k-d trees is shown in Algorithm 1: To compute the ANNF, corresponding patches are computed in Line 1 for both image A and image B for a given patch size p. This results in $(h-p+1) \cdot (w+p+1)$ patches of dimensionality $d = p^2c$. To facilitate the nearest neighbour search, the dimensionality of the patches is reduced. Instead of resorting to the so-called Walsh-Hadamard transform [7], we conduct a principal component analysis (PCA) on a *random subset* of size n_m of the patches (taken from both P_A and P_B) and use the resulting model m to obtain patches in a reduced feature space with dimensionality d_r. This is achieved via the function ReduceDim, which yields the reduced patches r_A and r_B. Next, a k-d tree is fitted on the reduced patches r_B of image B. Following He and Sun [7], we consider a k-d tree, which iteratively splits the dataset along the axes with widest spread. In addition, a mapping between the (indices of the) original patches and the leaf indices is generated, i.e., each patch in the original image "remembers" the leaf that contains it.

The procedure SearchContainingLeaf extracts for each (reduced) patch in image A (query) an initial set of k nearest neighbour candidates from the leaf to which the patch naturally belongs to (i.e. a simple top to bottom traversal in the k-d tree without backtracking), and procedure ExactKnnFirstRow computes the exact nearest neighbours for the first row of patches via a standard k-d tree traversal (i.e. with full backtracking). The 2D arrays I_k and V_k contain, for each query, the indices and distances of/to the k nearest neighbours.

The propagation-assisted nearest neighbour search is conducted by procedure ProcessRows, which aims to improve the initial candidates computed by SearchContainingLeaf: The patches of image A are processed row-wise in the manner outlined above, i.e., for each patch of a given row, up to k new leaves (to be searched) are identified by looking at the nearest neighbours of the patch just above the current one, and by selecting the leaves containing the (reference) patches just below them.[1] After having processed all rows, one ends up with k nearest neighbour candidates for each patch. The procedure SelectBest processes all the patches and identifies, for each patch, the best among the k available candidates w.r.t. the *original* patches P_A and P_B (this fine-tuning step is called *re-ranking* by He and Sun [7]).

C. Data-Parallel Implementation on GPUs

Next, we describe the data-parallel implementation of Algorithm 1. Since the running time is dominated by the propagation step, we explain the implementation of ProcessRows in detail and only give a bird-eye view of the other steps:

FitPCA: Considering a small random subset is sufficient to obtain a decent PCA model (e.g. using $n_m = 1000$ is generally sufficient). Note that, since n_m is a constant, fitting the PCA model takes only constant time. For this reason, the practical runtime is negligible and we do not parallelize this step.

CreatePatches and ReduceDim: Creating the patches for images A and B is a straightforward data-parallel operation, and reducing the patch dimensionality comes down to code that resembles the matrix-multiplication pattern. In principle, these two steps can be fused together so that the arrays P_A and P_B are not manifested in memory, thus significantly reducing the memory-footprint requirements. However, this prevents locality optimizations for the ReduceDim step—e.g., block and register tiling—resulting in up to 33% application-level slowdown. As such, our implementation manifests P_A in global memory, then reduces the dimensionality of P_A, and repeats the process for image B, while reusing the memory of P_A for P_B. The latter is freed immediately after r_B has been computed.

BuildKDTree: Building the k-d tree is performed entirely in data-parallel fashion. The representation of internal nodes consists of a pair $(d_{spr}^{max}, m_{spr}^{max})$ denoting the index of the dimension (of maximal spread) that is to be split for that node and its median value, respectively. The leaves store the (reduced) patches they contain using a 2D array representation (number of leaves $\times$ leaf size). The computation starts by padding r_B with patches having infinity on all dimensions, such that the resulting binary tree is perfectly balanced. The padding introduces at most one more patch per leaf, and thus results in negligible overhead, e.g., if the leaf size is 100, then the memory overhead is less than 1%.[2] The advantage of padding is that it enables *regular* nested parallelism, which can be efficiently mapped to the GPU hardware. For example, each node at a certain level in the tree will contain the same number of patches, and, sorting the patches can be performed in parallel both at node level and across all nodes at the same level. Then we compute the minimal and maximal element on each dimension, denoted $lb_i^0, ub_i^0, \forall i \in 0..d_r - 1$, where d_r is the number of reduced dimensions. Finally, a sequential loop of count equal to the height of the tree is executed such that nodes at each level are processed in parallel:

1 The lower and upper bounds for each dimension of the set of patches belonging to the current node are computed by making a copy of $lb_i^0, ub_i^0, \forall i$ and updating them according to the median indices and values encountered during a walk from the root of the tree to this node.
2 A map-reduce computation is applied on the result of step 1 to identify the dimension of highest spread (d_{spr}^{max}).
3 The patches corresponding to the current node are sorted w.r.t the values of dimension d_{spr}^{max}—we use radix sort and process 2-bits at a time.[3]
4 Finally the value of the median of the current node m_{spr}^{max} is extracted as the middle element of the sorted sequence, and the patches are split in parallel between the two children of the current node.[4]

SearchContainingLeaf: This procedure processes in parallel all the patches in image A (a.k.a., queries). Since the

[1]Note that this step uses the indirect array that records for each reference patch, the leaf to which the patch belongs to.

[2]Furthermore, the padded nodes will be naturally contained by the rightmost leaves, and if a leaf consists entirely of padded nodes, then it will never be searched since its median will necessarily be infinity.

[3]Please note that this is a batch sort, as it is performed in parallel across the patches belonging to a node and also in parallel across all the nodes at the same level in the tree. This is enabled by the padding procedure that creates a perfectly-balance tree.

[4]This step requires only a parallel write operation (scatter) because the sort is performed on an array of tuples obtained by zipping together the values of the dimension of highest spread with their indices $0 \ldots q-1$, where q is the number of patches belonging to a node at that tree level. Thus what is left to do is to reorganize the patches according to the permuted indices.

total number of queries is larger than the amount of parallelism needed to fully utilize the GPU hardware, we use one thread to process each query, and sequentialize the inner parallelism. The procedure has three (sub-)steps:

1. for each query the tree is walked from the root to find the leaf the query naturally belongs to;
2. the queries are sorted according to their corresponding leaf indices computed in step 1;
3. for each query, an initial set of k nearest neighbours is computed by performing a brute-force search in the corresponding leaf (found in step 1).

Step 2 is an optimization that improves locality of reference: on average, we will have a number of leaf-size queries that correspond to the same leaf. Grouping these queries together makes it likely that they will be executed on the same streaming multiprocessor (SM), which enables the reuse of the leaf data from the L1 cache—because the reference patches of the leaf are accessed in the same order by all such queries. This optimization speeds up the computation of SEARCHCONTAININGLEAF by a $3-4\times$ factor.

EXACTKNNFSTROW: This procedure is used to compute the exact set of k nearest neighbours for the patches (queries) belonging to the first row of image A by means of standard k-d tree traversal. Since the GPU hardware does not support (well) backtracking, we implement the tree traversal with a while loop. The input of an iteration is, for each query, the last visited leaf (initially the one computed by SEARCHCONTAININGLEAF) and a stack represented as an integer (initially zero). The stack uses one bit to record for each internal tree node on the current path whether the node's child that is on the other side of the median w.r.t. the query has been visited:

1. An iteration computes the next leaf to be visited and a new stack, by (i) walking up the current path until it finds a parent whose bit is not set *and* the median test fails, i.e., the corresponding child was not visited and may possibly contain better candidates, and then by (ii) walking down the tree to a new leaf by following the children that are on the same side of the median as the query. For a certain query, the traversal ends when step (i) reaches the root and its stack bit is set (both children were visited).
2. If a new leaf was found in step 1, then a brute-force search is performed on that leaf to improve the k candidates. The implementation of this step is similar to the one used for PROCESSROWS, which will be explained in detail next.

PROCESSROWS: This procedure implements the propagation step in which the nearest neighbours of the patches of a row are improved by searching the leaves inferred from the nearest neighbours of the patch just above the current one on the previous row. This is implemented as a sequential loop, whose iterations process an entire row of patches in parallel by means of two computational kernels.

The first kernel assigns one thread per patch in the current row, which computes the set of (new) leaf indices to be propagated from the patch just above it, denoted `prop_leaves`. The to-be-propagated leaves do not contain duplicates and do not include the leaf that naturally contains the patch, since the latter has been (computed and) searched by the SEARCHCONTAININGLEAF procedure. As such, `prop_leaves` is represented as a (padded) matrix of row size $\times$ k integers, and we store in a separate array, denoted `prop_lens`, the number of to-be-propagated leaves corresponding to each patch.

The second kernel implements the brute-force search of the new leaves. The common way to implement this step would be to assign one thread per (query) patch and to traverse the (reference) patches stored in the leaf while updating at each step its nearest neighbours (if necessary). Such an implementation is inefficient for two reasons: (i) the number of to-be-searched leaves varies across queries, which gives raise to significant thread divergence overhead, and (ii) the maximal size of a row in our experiments is about 4000 patches, which would severely underutilize the GPU hardware, which requires order of tens-of-thousands threads for full occupancy.

```
// Input: i is the current-row index (i > 0)
//    knns: [num_rows][row_len][k](int,float),
//    queries: [num_rows][row_len][d]float,
//    prop_lens: [row_len]int,
//    prop_leaves:[row_len][k]int
//    leaves: [num_leaves][leaf_len][d]float
forall j in 0 .. row_len-1 //CUDA grid parallel
  __shared__ float  dists[leaf_len];
  __shared__ struct{int ind; float dst} knn[k];
  __shared__ float  query[d];
  collective_copy_glb2shr(knn, knns[i,j], k);
  collective_copy_glb2shr(query,queries[i,j],d);
  forseq l in 0 .. prop_lens[j]-1
    int leaf_ind = prop_leaves[j,l];

    forall p in 0 .. leaf_len-1 //CUDA block
      dists[p] = sumOfSquares( query,
                         leaves[leaf_ind,p] );
    end //forall

    int t = k-1; bool cycle = true;
    while (cycle && t >= 0) do
      (m_ind, min_dst) =
          reduce(minind, dists); // CUDA block
      if( min_dst < knn[t].dst ) {
        knn[t].ind = leaf_ind*leaf_len+m_ind;
        knn[t].dst = min_dst;
        dists[m_ind] = ∞; t = t - 1;
      } else { cycle = false; }
    end // while
    sortPartialSortedSeqs(knn);
  end // forseq
  collective_copy_shr2glb(knns[i,j], knn, k);
end // forall
```

Fig. 3: Pseudocode of the GPU implementation corresponding to the brute-force search of the propagated leaves in procedure PROCESSROWS. The outermost **`forall`** loop corresponds to the Cuda grid. The inner **`forall`** and the **`reduce`** are executed in parallel by the threads of a Cuda block, which has the same size as the leaf of the k-d tree.

Figure 3 presents the pseudocode of our implementation, which processes one query in one CUDA block, whose size is chosen to be equal to the number of reference patches in a leaf, denoted `leaf_len`. The code starts by allocating an array named `dists` of `leaf_len` floats in shared memory, and by copying the current query and its current nearest-neighbours candidates (`knn`) from global to shared memory. The sequential loop **`forseq`** processes a new leaf at a time: the nested **`forall`** loop is executed in parallel across the CUDA-block threads and initializes the `dists` array with the distance from the current query to each patch in the leaf. Then a sequential **`while`** loop refines the nearest neighbours:

Each iteration starts by computing the minimal distance and corresponding index of the distances in `dists`—i.e., **`reduce`**`(minind, dists)`. This step is also executed in parallel across the threads of the CUDA block. If the computed distance is smaller than the worse-known initial nearest neighbour, then the first thread in the CUDA block updates (i) the nearest neighbours with the computed candidate, and (ii) the corresponding index in `dists` with infinity, such that the next iteration computes the next-best nearest neighbour of the leaf. The **`while`** loop terminates when the computed candidate has a distance higher than the worst still-alive nearest neighbour from the initial set. Now `knn` consists of two partially sorted sequences: (i) the initial one starting at the beginning and (ii) the one computed from the current leaf starting at the end. These are merged into one ascending sorted sequence by the first thread in the block (`sortPartialSortedSeqs`). After all new leaves have been processed, the resulting nearest neighbour set is (collectively) copied back to global memory.

This implementation has `row_len`×`leaf_len` degree of parallelism, which is enough to fully utilize GPU hardware, and it lifts the (divergent) unbalanced behavior from individual threads to CUDA blocks, where it is supported well by the block scheduler. Finally, the computation is performed mostly in shared memory, which is much faster than global memory.

SELECTBEST: This procedure selects in parallel for each query patch the best of its k nearest neighbours w.r.t. the original-image patches (i.e., full rather than reduced dimensionality). Because this step is applied to all queries in parallel, processing one query per CUDA thread fully utilizes hardware.

IV. EXPERIMENTS

This section assesses the performance of our parallel implementation, in terms of (i) run time and accuracy comparison with existing state-of-the-art implementations, and (ii) the impact of optimizations and the percentage of runtime of computational (sub)kernels.

A. Experimental Setup

All experiments described in this section were performed on an AMD system with 16GB DRAM, 8 Ryzen 7 3700X cores, using 2-way multithreading, which is also equipped with an GeForce RTX 2070 NVIDIA GPU with 8GB DRAM, 2304 cuda cores running at 1.41GHz under CUDA 10.1. The CPU-parallel code was hand-written in `C` and compiled with GCC 7.5.0 (`-fopenmp -O2`); the reported CPU run times were validated across 2 runs and are based on 16 threads. The GPU code has been written in the Futhark language [29]–[31]. [5]

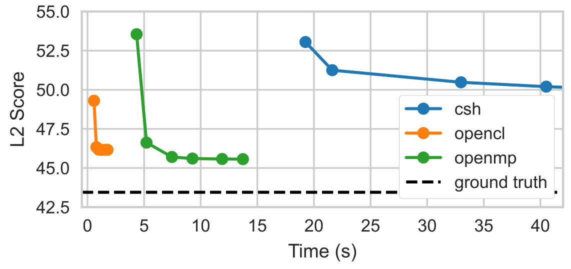


Fig. 4: Comparison between competing algorithms using the VidPairs dataset [6] with all 133 image pairs in native resolution. The patch size was fixed to 8x8. All methods had to find $k = 8$ nearest neighbours. Markers on the `openmp` and `opencl` curve represent different levels of PCA reduction (5, 10, 20, 30, 40, 50), whereas markers on the `CSH` curve represent different amounts of iterations (3, 5, 8, 10).

As baselines, we considered both CSH [6] as well as our multi-threaded version implementation of propagation-assisted k-d trees (denoted by `CSH` and `openmp`, respectively).[6] These baselines have been reported to be superior over other approaches, including PatchMatch [6], [7]. Below, we compare our massively-parallel implementation (denoted by `opencl`) with these baselines and investigate the influence of several parameters. We also provide a one-to-one run time comparison between `openmp` and `opencl`.[7] As datasets, we considered the VidPairs dataset [6], which consists of 133 pairs of similar images taken from 1080 Full HD movie trailers. We additionally resorted to VidPairs4K, a novel dataset consisting of 155 image pairs extracted from ultra high definition film trailers in 4k resolution, see the appendix for more details.

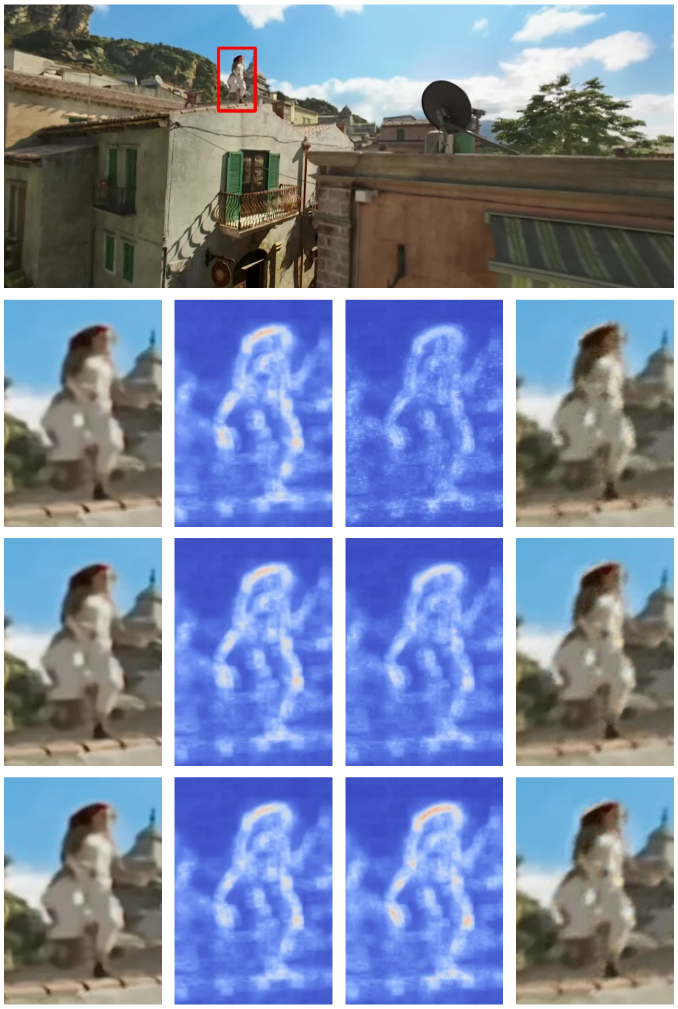
B. Time-Accuracy Comparison

Figure 4 shows the performance of three competitors. The performance of the algorithms is expressed as the (average) time it takes to process an image pair, *and* as the (average) L2 error of the corresponding nearest neighbour field. The test has been performed on the entire VidPairs dataset in native resolution. Both `openmp` and `opencl` outperform `CSH`.

[5]Futhark is used to implement the data-parallel computational kernels, which are then easily interfaced in projects written in mainstream, productivity-oriented frameworks by means of specialized code generators [31]. While Futhark provides good supports for abstraction and for writing generic libraries [30], the current foreign-function interface (FFI) is limited to monomorphic code, but various techniques have studied how to extend the FFI to support type-parameterized components [32]. Our implementation makes heavy use of Futhark's support for adapting the compilation technique to the particularities of the dataset [29], which builds on dynamic analysis techniques used in the area of automatic parallelization [33], [34].

[6]For `CSH`, we resorted to the publicly available single-core implementation that is available under `http://www.eng.tau.ac.il/~simonk/CSH/`. The default parameters were used unless specified.

[7]The implementation in the form of an open-source Python package is available at `https://github.com/diku-dk/annfmp`.



(a) CSH (b) Ours

Fig. 5: Accuracy of generated fields and their respective reconstructions. Top row: Original image. Second row: Reconstruction as well as the Euclidean error of the highlighted area after 3 iterations (`CSH`) and with PCA 5 (`opencl`). Third row: 5 iterations (`CSH`) and PCA 10 (`opencl`). Fourth row: 10 iterations (`CSH`) and PCA 20 (`opencl`).

The multi-core implementation `openmp` achieves a better L2 score as `CSH` in up to a quarter of the time and `opencl` is more than five times faster than `openmp`. In addition, `openmp` and `opencl` achieve significantly higher degrees of accuracy when compared to what `CSH` yields. Figure 5 shows the difference in error distribution between the `CSH` and our methods. More specifically, the L2 error is more evenly distributed over the fields in limited runs when compared to `CSH`, as it can be seen in row 2 and 3 of Figure 5.

Comparing `opencl` to `openmp` shows a definite advantage by utilising the resources of the GPU. `opencl` yields similar error scores in only a fraction of the time required by

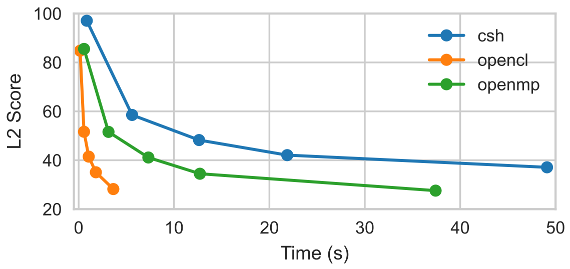


Fig. 6: Comparison between competing algorithms using the VidPairs4K dataset with all 155 image pairs. The patch size was fixed to 8x8 and all methods had to find $k = 8$ nearest neighbours. Markers on the plots denote testing points, where more time demanding tests are on larger images. All image sizes (500, 720p, 1080p, 1440p and 2160p (4K)) from VidPairs4K have been included.

`openmp`.[8] The run time of `openmp` quickly increases with increasing PCA, while `opencl`'s run time remains near static, and thus producing a $1-10\times$ speedup over `openmp`.

C. Scalability

Next, we investigated the scalability of the different approaches given images of increasing sizes, see Figure 6. On tiny image sizes (image with width 500 VidPairs4K), the performance is very similar for all methods. The very small problem size causes overhead to likely be the prime contributor to run time. It is with the larger problem sizes that we see our `opencl` to show its parallel advantage. In particular, it stays under 2 seconds even on the most difficult of tasks and outperforms `openmp` by a $10\times$ factor.

D. Patch Size

The patch size directly impacts the complexity of the problem by defining the dimensionality of the original search space. Figure 7 shows the run time and L2 scores of all three algorithms on VidPairs given varying patch sizes. Both `openmp` and `opencl` consistently maintain an accuracy/time lead over `CSH`. Additionally, both `openmp` and `opencl` achieve similar or lower ultimate L2 scores. This difference is more pronounced on smaller patch sizes, where the L2 score difference is relatively larger. It is worth stressing that `opencl` scales well with the dimensionality of the patches (its runtime is barely increasing).

E. Performance Analysis

Table I presents the total GPU execution time (second row) and the percentage of the execution time spent in each of the stages presented in Algorithm 1 and discussed in Section III-C. The dataset consists of two images, of sizes $800 \times 1920 \times 3$ (left) and $1600 \times 3840 \times 3$ (right), where the

[8]The implementations of the `openmp` and `opencl` implementations vary slightly from a conceptual perspective (e.g. the k-d tree construction is slightly different); hence, the slightly varying L2 scores.

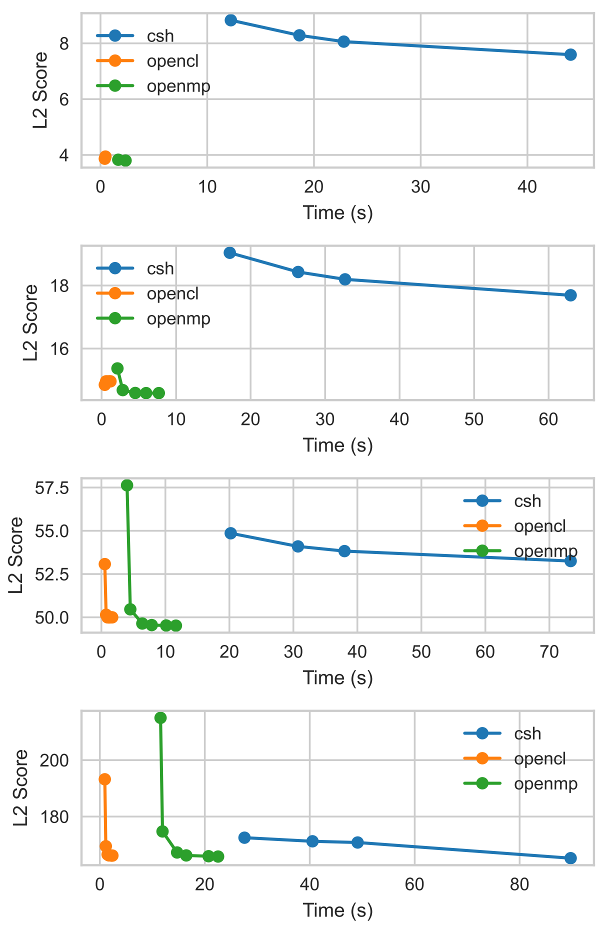


Fig. 7: Adjusting the patch size (2x2, 4x4, 8x8, 16x16). All algorithms had to find $k = 8$ nearest neighbours and the algorithms ran on the first 25 pairs of the VidPairs dataset. For both `openmp` and `opencl`, variations w.r.t. the PCA dimensions (5, 10, 20, 30, 40 and 50) are plotted. For `CSH`, a varying number of iterations are plotted (5, 8, 10, and 20).

reduced dimensionality has been varied from 10 to 40. One can observe that the PROCESSROW accounts for about half of the runtime, and together with EXACTKNNFSTROW they account for $57\% - 70\%$ of total runtime. The k-d tree construction (BUILDKDTREE) accounts for $7 - 16\%$ of runtime and the percentage decreases when increasing the dimensions, while SEARCHCONTAININGLEAF accounts for $2 - 11\%$ of runtime, and the percentage increases when increasing dimensionality.[9]

Table II uses as dataset the same two images as table I. Row `openclunopt` reports the impact of the optimization discussed at the end of section III-C, i.e., using an entire Cuda block (rather than one thread) to execute the brute-force search of a leaf by one query. This optimization results in application-level speedups between $3.8 - 5.2\times$. Row `openmp` reports the speedup of the GPU version in comparison to the multi-threaded CPU version using OpenMP: the average speedup is about $8\times$ and it seems to increase with the size of the image.

[9]For example it increases sharply when `reduced-dim` goes from 20 and 30: sorting the queries optimizes locality, but the higher dimensionality has caused the resident memory set to not fit in cache anymore.

TABLE I: Total run time in seconds and percentage from total run time of the stages described in Algorithm 1 for two images of sizes $800 \times 1920 \times 3$ (left) and $1600 \times 3840 \times 3$ (right). The first row shows that the reduced dimension of the patch is varied from 10 to 40. The large-patch size is 192.

reduced-dim	10	20	30	40	10	20	30	40
run time sec	0.83	1.00	1.34	1.81	2.80	3.62	4.94	6.30
FitPCA %	5.5	2.2	1.9	1.4	.67	3.5	2.8	0.5
RedDim %	8.2	8.3	7.7	8.0	8.7	7.8	7.7	7.2
k-d Tree%	14.5	11.8	9.5	7.0	16.6	13.5	9.9	7.8
ContLeaf%	2.2	3.8	8.1	9.2	2.6	4.1	9.9	11.6
ExactKnn%	13.6	14.9	14.4	13.7	7.9	8.8	9.1	8.9
ProcRows%	44.0	49.2	51.2	55.5	45.1	48.5	50.0	56.2
SelectBest%	9.3	7.5	5.5	4.0	15.3	11.2	8.5	6.4
PythonOV%	2.6	2.2	1.6	1.2	3.0	2.6	2.2	1.4

TABLE II: Speed-up of `opencl` in comparison with (i) the unoptimized one that uses one thread to perform the brute-force search of a leaf (`openclunopt`) and (ii) the `openmp` implementation. We use the same two images as in Table I.

Speed-up vs.	10	20	30	40	10	20	30	40
`openclunopt`	3.8×	4.6	5.2	4.7	3.2	3.9	4.5	4.5
`openmp`	6.7×	8.0	7.8	7.3	7.8	9.3	8.8	9.0

F. *Visual Analysis*

Reconstructions can be built using the nearest neighbour fields and their source images. A representation of the target image is constructed by having all patches contribute to every pixel they overlap on. This procedure is known as “voting” [4]. Figure 8 depicts voting results of both `opencl` and `CSH` on multiple image sizes. Figure 9 compares this representation to the L2 score of the produced fields, whereas Figure 5 compares reconstructions between different iterative and dimensionality reduction settings. In particular, our reconstruction portrays sharp, high contrast lines more accurately (Figure 8) and is more colour accurate (Figure 9).

V. CONCLUSIONS

We propose an efficient highly-parallel implementation for propagation-assisted k-d trees. The resulting framework is about one order of magnitude faster than a corresponding multi-threaded implementation and exhibits a small memory footprint, which renders it capable to computing nearest neighbour fields for large amounts of high-resolution image pairs. We believe that our implementation will be useful for a variety of compute- and data-intensive tasks in computer vision in future that rely on nearest neighbour fields.

REFERENCES

[1] Y. Wexler, E. Shechtman, and M. Irani, “Space-time video completion,” in *2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2004), 27 June - 2 July 2004, Washington, DC, USA*. IEEE Computer Society, 2004, pp. 120–127.

[2] D. Glasner, S. Bagon, and M. Irani, “Super-resolution from a single image,” in *IEEE 12th International Conference on Computer Vision, ICCV 2009, Kyoto, Japan, September 27 - October 4, 2009*. IEEE Computer Society, 2009, pp. 349–356.

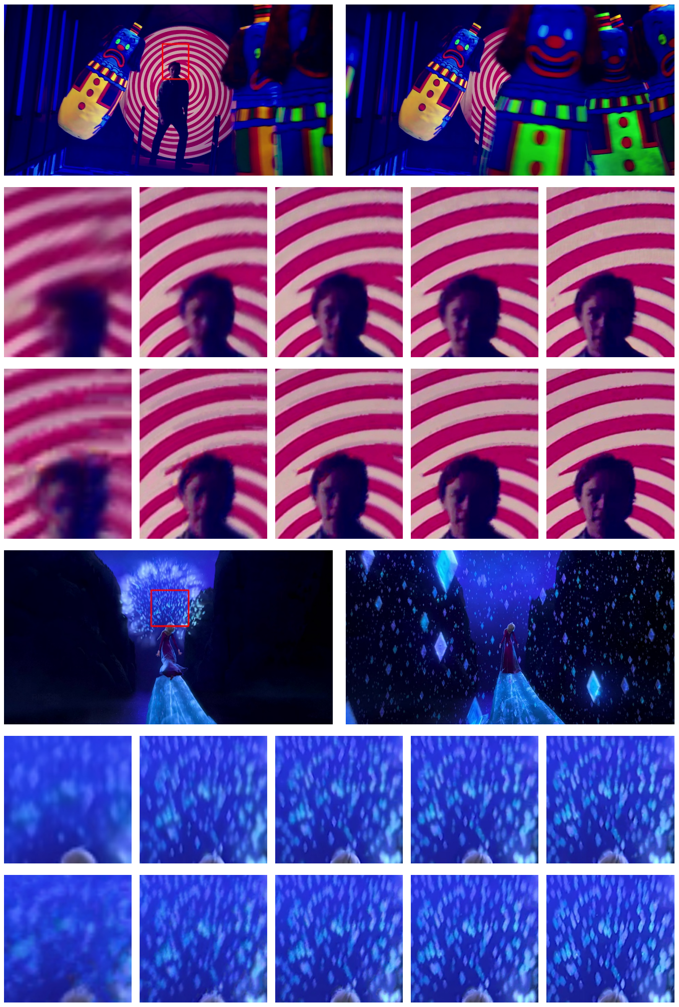


Fig. 8: Increasing accuracy by increasing the source image size. For each of the two examples, the top row has been generated via CSH, while the bottom row has been generated using opencl. Input image sizes from left to right are 500, 720p, 1080p, 1440p and 4K, respectively.

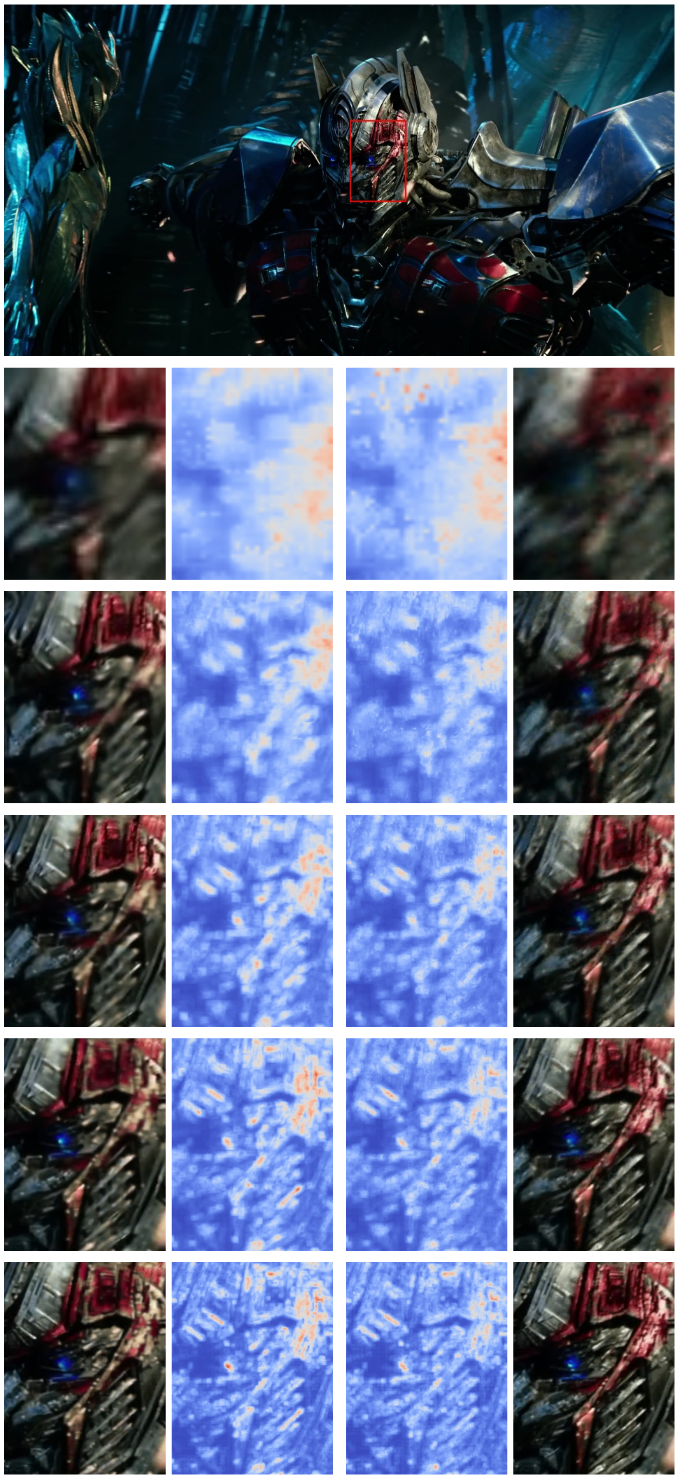


Fig. 9: Accuracy of CSH and opencl given varying image sizes (top to bottom: 500, 720p, 1080p, 1440p, and 4K).

[3] A. Buades, B. Coll, and J. Morel, "A non-local algorithm for image denoising," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), 20-26 June 2005, San Diego, CA, USA*. IEEE Computer Society, 2005, pp. 60–65.

[4] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman, "Patchmatch: A randomized correspondence algorithm for structural image editing," in *ACM Transactions on Graphics (ToG)*, vol. 28, no. 3, 2009.

[5] J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[6] S. Korman and S. Avidan, "Coherency sensitive hashing," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 6, pp. 1099–1112, 2015.

[7] K. He and J. Sun, "Computing nearest-neighbor fields via propagation-assisted kd-trees," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2012, pp. 111–118.

[8] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, 2nd ed. Springer, 2009.

[9] J. Friedman, J. Bentley, and R. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, 1977.

[10] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th International Conference on Machine Learning*. New York, NY, USA: ACM, 2008, pp. 104–111.

[11] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, "Deep learning with COTS HPC systems," in *Procs. of the 30th Int.*

Conference on Machine Learning. JMLR.org, 2013, pp. 1337–1345.

[12] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer k-d trees: Processing massive nearest neighbor queries on GPUs," in *Proceedings of the 31st International Conference on Machine Learning*, ser. JMLR W&CP, vol. 32, no. 1. JMLR.org, 2014, pp. 172–180.

[13] Z. Wen, R. Zhang, K. Ramamohanarao, J. Qi, and K. Taylor, "Mascot: Fast and highly scalable SVM cross-validation using GPUs and SSDs," in *Proceedings of the 2014 IEEE International Conference on Data Mining*, 2014, pp. 580–589.

[14] S. Popov, J. Günther, H. Seidel, and P. Slusallek, "Stackless kd-tree traversal for high performance GPU ray tracing," *Computer Graphics Forum*, vol. 26, no. 3, pp. 415–424, 2007.

[15] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 126:1–126:11, 2008.

[16] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log N)," in *IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006, pp. 61–69.

[17] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," in *Proceedings of the Symposium on Interactive 3D Graphics and Games*. ACM, 2007, pp. 167–174.

[18] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," in *Procs. of the 17th IEEE International Conference on Image Processing*. IEEE, 2010, pp. 3757–3760.

[19] J. Pan and D. Manocha, "Fast GPU-based locality sensitive hashing for k-nearest neighbor computation," in *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2011, pp. 211–220.

[20] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," in *Computational Science – ICCS 2006*, ser. Lecture Notes in Computer Science, vol. 3994. Springer, 2006, pp. 196–199.

[21] N. Sismanis, N. Pitsianis, and X. Sun, "Parallel search of k-nearest neighbors with synchronous operations," in *IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–6.

[22] L. Cayton, "Accelerating nearest neighbor search on manycore systems," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 402–413.

[23] D. Qiu, S. May, and A. Nüchter, "GPU-accelerated nearest neighbor search for 3D registration," in *Procs. of 7th International Conference on Computer Vision Systems*. Springer, 2009, pp. 194–203.

[24] W. Wang and L. Cao, "Parallel k-nearest neighbor search on graphics hardware," in *Procs. of 3rd International Symposium on Parallel Architectures, Algorithms and Programming*. IEEE, 2010, pp. 291–294.

[25] J. Heinermann, O. Kramer, K. L. Polsterer, and F. Gieseke, "On GPU-based nearest neighbor queries for large-scale photometric catalogs in astronomy," in *KI 2013: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 8077, pp. 86–97.

[26] N. Nakasato, "Implementation of a parallel tree method on a GPU," *Journal of Computational Science*, vol. 3, no. 3, pp. 132–141, 2012.

[27] C. E. Oancea, A. Mycroft, and S. M. Watt, "A new approach to parallelising tracing algorithms," in *Procs. of the 2009 International Symposium on Memory Management*, ser. ISMM '09. ACM, p. 1019.

[28] D. Gadot and L. Wolf, "Patchbatch: A batch augmented loss for optical flow," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4236–4245.

[29] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 53–67.

[30] M. Elsman, T. Henriksen, D. Annenkov, and C. E. Oancea, "Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large," *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 97:1–97:30, Jul. 2018.

[31] T. Henriksen, M. Dybdal, H. Urms, A. S. Kiehn, D. Gavin, H. Abelskov, M. Elsman, and C. Oancea, "APL on GPUs: A TAIL from the Past, Scribbled in Futhark," in *Procs. of the 5th Int. Workshop on Functional High-Performance Computing (FHPC)*. ACM, 2016, pp. 38–43.

[32] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt, "Parametric Polymorphism for Computer Algebra Software Components," in *Procs. 6th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. Mirton Publishing House, 2004, pp. 119–130.

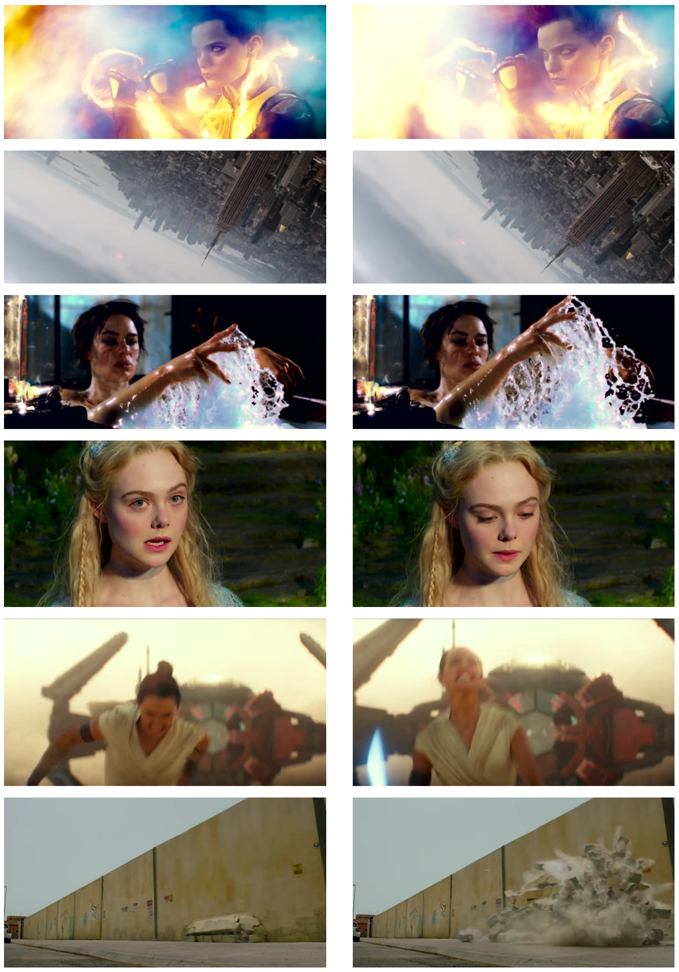


Fig. 10: Six image pair examples of the VidPairs4K dataset, which consists of 155 image pairs.

[33] C. E. Oancea and L. Rauchwerger, "A hybrid approach to proving memory reference monotonicity," in *Languages and Compilers for Parallel Computing (LCPC)*, S. Rajopadhye and M. Mills Strout, Eds. Springer Berlin Heidelberg, 2013, pp. 61–75.

[34] C. E. Oancea and A. Mycroft, "Set-congruence dynamic analysis for thread-level speculation (TLS)," in *Languages and Compilers for Parallel Computing (LCPC)*. Springer-Verlag, 2008, pp. 156–171.

APPENDIX

The VidPairs4K dataset has been introduced for this research. Similar to the VidPairs dataset [6], it has been harvested from official film trailers, see Figure 10. In total, 155 image pairs have been extracted from film trailers that natively support resolutions up to 4K (3840×2160 pixels). There are at least 3 and at most 30 frames in between the images of a given pair. Just as for the VidPairs dataset, images in VidPairs4K vary between widescreen (16:9) and cinematic widescreen (2.40:1); though the width does not vary between image pairs.

The dataset and our custom framework for testing and comparing techniques for ANNF generation are publicly available at `http://trobroek.com/vidpairs4k/`.