UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor Thesis

GPU Implementations of KD-Tree Assisted Approximate Nearest Neighbour Algorithms

Jonas Lau Kristensen [hvd466]
Daniel Nicholas Mølhave [rjs561]

Supervisor: Cosmin Eugen Oancea

Submitted: June 12, 2026

Contents

Abstract

Finding similar patches between two images is a widely used technique in computer vision. This task, also known as computing the Nearest Neighbour Fields, is computationally expensive. This has paved the way for approximate methods that trade exactness for speed, such as propagation-assisted KD-trees. Data-parallel implementations running on the GPU have shown to achieve large speedups over CPU implementations. However, the existing data-parallel implementation of propagation-assisted KD-trees enforces a regular tree layout with fixed size leaves, which deviates from the original sequential algorithm. In this thesis, we propose a variation[1] that supports irregular leaf sizes, thereby preserving the splitting rule of the sequential algorithm. We integrate our implementation, along with a data-parallel implementation of Randomized Approximate Nearest Neighbours, into an existing framework that already contains the regular data-parallel variant and a CPU variant of propagation-assisted KD-trees. We conduct a comparative analysis along two axes: runtime and L2 distance.

The results show that our implementation offers a competitive runtime/accuracy tradeoff among the propagation-assisted KD-tree variants and, in particular, scales better with leaf size. The randomized variant achieves by far the best accuracy overall, but at a higher runtime cost.

[1]The GitHub repository: https://github.com/diku-dk/annfmp

1 Introduction

Nearest Neighbour Search is an optimisation problem that finds the data point in a set that is closest to a given query point. It is widely used in data science, statistics, machine learning and image processing. The area in which Nearest Neighbour Search is relevant in this thesis is image processing, where subsequent image pairs from different movies are analysed and compared. One way of expressing similarities between two images is the Nearest Neighbour Field (NNF), where the image is divided into overlapping patches of pixels. Given an image pair $A, B \in \mathbb{R}^{h \times w \times c}$, the similarity between them can be expressed by finding, for every patch in A, the patch in B that best matches it under some distance measure.

Brute forcing NNF is computationally expensive since it has complexity of $O(nm)$, where n is the amount of reference data points and m is the amount of query data points, which can quickly become computationally infeasible as n increases. To reduce the number of comparisons, the patches of image B can be organised in a spatial search structure such as a KD-tree. A KD-tree is a k-dimensional tree data structure that for each node splits on the dimension of greatest spread and enables search in only $O(n \log n)$ time. This is efficient in low dimensions, but for larger dimensionalities ($d > 20$) the curse of dimensionality ruins the advantage as more and more subtrees must be visited. In the worst case all subtrees must be visited, meaning it is no more efficient than brute force [16]. To mitigate this, one can instead compute the Approximate Nearest Neighbour Field (ANNF), which aims to find, for each patch in image A, a patch in image B that closely approximates its true nearest neighbour. This can be computed with a KD-Tree and additional techniques that exploit spatial locality, namely propagation. Propagation relies on the heuristic that patches that are neighbours in image A tend to have matches that are neighbours in image B. This allows an algorithm to search for ANN candidates by only visiting a limited number of leaves with a high possibility of finding good candidates.

A data-parallel propagation-assisted KD-tree implementation on GPUs has been proposed by Oancea et al. [16]. The data-parallel implementation has a vastly superior runtime compared to a CPU multi-core implementation. The implementation is written in the functional Futhark programming language that comes with a compiler highly optimised for generating efficient data-parallel code [3]. There is, however, a deviation in how the data-parallel implementation constructs its KD-tree. In order to efficiently map onto GPU hardware, it forces regular nested parallelism by padding the leaves to maintain a constant leaf size and splitting using the middle index. By prioritising constant leaf sizes over respecting the split value, the implementation deviates from the sequential algorithm by He and Sun [5].

Our contributions are comprised of a new data-parallel implementation that adheres to the algorithm proposed by He and Sun, by allowing for irregular sized leaves. The testing framework found in the repository associated with Oancea et al.'s paper is made up of a pipeline from `Futhark` → `C` → `SWIG` → `Python`. We have expanded it with this new irregular data-parallel implementation as well as a data-parallel implementation of Randomized Approximate Nearest Neighbours written by Lynghus [10].
Lastly we conduct a comparative analysis of four different ANNF implementations demonstrating the strengths and weaknesses along two axes: accuracy and runtime, as well as the ability to tune between them.

2 Related Work

This section briefly discusses algorithms that are relevant to the general problem context. In particular, we consider PatchMatch, Locality Sensitive Hashing and Coherency Sensitive Hashing.

2.1 PatchMatch

PatchMatch is a scheme that finds approximate nearest neighbour fields based on the spatial locality of the images. PatchMatch contains two main phases, initialization and iteration. It begins by setting an initial guess for the nearest neighbour field (NNF), which can either be derived from prior information or a random field. Then it applies an iterative phase to the NNF, containing two stages, propagation and random search. The propagation stage uses coherence between and within an image pair to find nearest neighbour candidates. This relies on the notion that if a patch has found a match, then the neighbours of that patch are likely to have a match near the original patch's match. The matches can then be propagated to neighbours, which can propagate them further. Due to the fact that propagation can only distribute matches that exist nearby, it can be trapped in local minima. Random search is used to mitigate this by trying a subset of candidates around the area of the best match for a given patch. [1]

2.2 Locality-Sensitive Hashing

Locality-Sensitive Hashing (LSH) is a scheme to find approximate nearest neighbours in high dimensional space, using hash functions to filter for candidates. LSH contains two main phases, indexing and searching. In the indexing phase a concatenation of `k` hash functions hashes an input point into a hash table. This process is done for a chosen number, `L` hash tables. Concatenating multiple hash functions reduces the amount of false positive hash collisions whilst using multiple hash tables helps reduce the number of

false negatives. The searching phase then hashes a query in the `L` tables, and makes a list of candidates based on all the points found in the query's buckets i.e. the table cell holding all points that hashed to the same value. The result is computed by comparing the candidates distance to the query, returning the closest candidate.[2]

2.3 Coherency Sensitive Hashing

Coherency Sensitive Hashing (CSH) is an approach to find approximate nearest neighbour fields that combines properties from two other approaches, PatchMatch and LSH. CSH is comprised of two main phases, indexing and searching. The indexing phase follows the same structure as LSH, except that the hash family consists of Walsh Hadamard kernels, which allow for more efficient computation and make it possible to lower bound the L2 distance between a query patch and a candidate. The search phase uses the L hash tables constructed during indexing. It follows a similar structure to LSH, but expands it by considering candidates beyond those from hash collisions. The inspiration CSH takes from PatchMatch is the idea of coherence, where a patch inherits candidates from its spatial neighbours matches. These candidates are added to the patch's list alongside those found through LSH. CSH ranks candidates using the Walsh Hadamard kernels. Rather than computing the distance from each candidate to the query, CSH keeps a running best match per patch and compares candidates against it, rejecting a candidate as soon as its lower bound exceeds the distance to the current best match. [9]

3 Background

To understand the contributions in this thesis, it is necessary to first establish the technical context in which the implementation is situated. This section therefore reviews the prior work that the thesis builds upon.
The implementation and experiments build on several closely related ideas. We first describe how nearest neighbour fields can be computed using KD-trees, before considering how propagation can be used to improve candidate quality. We then discuss how these ideas can be expressed in a data-parallel setting, and finally introduce randomized ANNs as an alternative approach used in the comparative analysis.

3.1 Computing Nearest Neighbour Fields Using A KD-Tree

Nearest Neighbour Fields (NNFs) can be formulated as follows:
Let $P_A \subset \mathbb{R}^d$ and $P_B \subset \mathbb{R}^d$ be the sets of patches from images A and B. Each set contains $(h-p+1)\times(w-p+1)$ patches, where each flattened patch has dimensionality $d = p^2c$, here p is the patch size and c is the number of

colour channels, which in the classic case is three for RGB. The NNF for A and B then assigns to each patch $q \in P_A$ its k nearest neighbours $q' \in P_B$ with respect to a distance measure. In this thesis, the Euclidean distance (L2) is used, computed as $\text{dis}(q, q') = \left(\sum_{i=1}^{d}(q_i - q'_i)^2\right)^{\frac{1}{2}}$.
The search can be made cheaper by using KD-trees. The classical KD-tree is a balanced binary tree whose root holds an entire point set $P = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$. It is built level by level. A node v splits its point set P_v into two subsets at the median of the dimension of highest spread, and the construction recurses until a stopping criterion is reached. Finding the nearest neighbour of a query then proceeds in two phases. In the first phase, the tree is traversed top down to find the leaf that naturally contains the query, all points in that leaf are compared against the query and the k best are stored. In the second phase, the tree is traversed bottom up, visiting a subtree and its leaves only when the distance from the query to that subtree's bounding box is smaller than the distance to the current k-th worst neighbour, updating the candidate set whenever a better candidate is found. On higher dimensions this approach suffers from the curse of dimensionality and becomes infeasible.

3.2 Propagation-Assisted KD-Trees

Propagation-assisted KD-Trees, proposed by He and Sun, reduce the computational cost by exploiting the spatial coherence of images, patches that are neighbours in image A tend to have matches that are neighbours in image B. The algorithm therefore performs an exact search only for the first row of query patches and then propagates candidate matches. To avoid the curse of dimensionality when computing the exact KNN for the first row, the dimensionality of all patches is reduced. Oancea et al. uses Principal Component Analysis in their implementations. The exact k nearest neighbours are computed for the first row of query patches in image A via a full KD-tree traversal. For every subsequent patch, an initial candidate set is taken from the leaf that naturally contains the query patch, and these candidates are then refined by selectively visiting a number of additional leaves inferred from the row above. An illustration of propagation can be found in Figure 1.

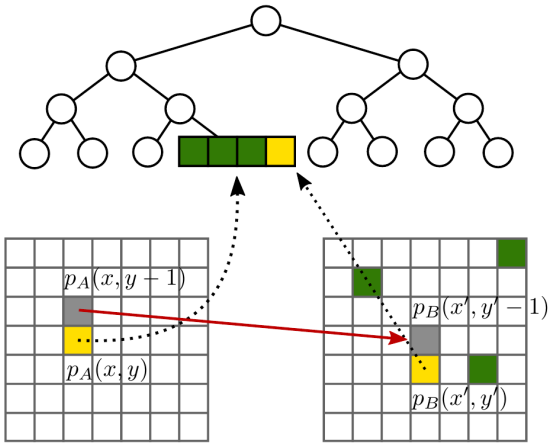


Figure 1: For a query patch $p_A(x, y)$, the patch directly above it, $p_A(x, y-1)$, has already been processed, so its nearest neighbour match $p_B(x', y'-1)$ in image B is already known. Adapted from [16, 5].

Rather than propagating only this single best match, He and Sun propagate all k neighbours of the patch above, together with the query's own natural leaf. This means at most $k+1$ leaves are visited per patch. Because all searching is done in the reduced space, the final match for each patch is reranked against the original, non reduced patches at the end.

The pseudocode for such an implementation is as follows: The particular representation comes from Oancea et al. [16]

Algorithm 1 Propagation-Assisted KD-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: $P_A, P_B = \text{CreatePatches}(A, B, p)$
2: $m = \text{FitPCA}(P_A, P_B, n_m)$
3: $r_A, r_B = \text{ReduceDim}(m, P_A, P_B)$
4: $T = \text{BuildKDTree}(r_B)$
5: $(I_k, V_k) = \text{SearchContainingLeaf}(T, k, r_A)$
6: $(I_k, V_k) = \text{ExactKNNFirstRow}(T, k, I_k, V_k, r_A[0])$
7: $(I_k, V_k) = \text{ProcessRows}(T, k, I_k, V_k, r_A[1:])$
8: $I = \text{SelectBest}(I_k, P_A, P_B)$
9: **return** I

As can be seen in the pseudocode, the algorithm follows a nine step process:

1. Patches are created from the two images.
2. A PCA model is computed using a subset of the patches.
3. The dimensionality of the patches is reduced using the PCA model from the previous step.
4. A KD-tree is fitted on image B's reduced patches, splitting on the dimension of highest spread.
5. SearchContainingLeaf finds KNN candidates for each query in the leaf it naturally belongs to, by a top down traversal.
6. ExactKNNFstRow computes the exact KNN for the first row of patches, via a bottom up traversal.
7. ProcessRows is the propagation step, it aims to improve the initial k candidates found in the SearchContainingLeaf step. The patches of image A are processed row by row, and for each query up to k new leaves are searched by taking the nearest neighbours of the patch directly above the query and selecting the leaves containing the reference patches below them.
8. SelectBest processes all patches and identifies, for each patch, the best of its k candidates with respect to the original, non reduced patches P_A and P_B.
9. Return Nearest Neighbours.

3.3 Data Parallel Propagation-Assisted KD-Trees

A regular data parallel version of He and Sun's algorithm has been proposed by Oancea et al. [16]. The overall structure mirrors He and Sun's sequential implementation, with several of its steps parallelized.

Before the KD-Tree is built, the reduced patches are padded with patches holding infinity in every dimension. This keeps the tree perfectly balanced and enables regular nested parallelism meaning the tree can be built in parallel at each level, since every leaf contains the same number of patches. Padding does not affect the L2 score, because the padded patches always sort into the rightmost leaves and are never searched since their median is infinity. Each level is processed in parallel by first computing, for every node at that level the lower and upper bound of each dimension from the values encountered on the walk from the root to that node. The dimension of highest spread is then determined, and the node's patches are sorted with respect to this. The patches are split between the node's two children at the middle element of the sorted sequence, with the value at that position

being the median. [16]

The SearchContainingLeaf step processes each query in parallel by assigning one thread per query and sequentializing the inner work. Each thread first walks the tree from root to leaf to find the leaf its query naturally belongs to. The queries are then sorted by their natural leaf to improve locality of reference. Finally, an initial KNN set is computed for each query by bruteforce search within that leaf.

The ExactKNNFstRow procedure is implemented with a while loop. Each query keeps track of its last visited leaf, initially its natural leaf, and a stack that records, for each node on the path, whether the child on the other side of the median has already been visited. Each iteration computes the next leaf to visit and updates the stack through a bottom up traversal, it walks up the current path until it finds a parent whose bit is not set and whose other child, still unvisited, may contain better candidates. It then descends that path to the leaf. Whenever a new leaf is reached, a bruteforce search is performed on it to refine the current candidates. The traversal terminates once it reaches the root and both children of every node on the path have been visited.

The propagation step is carried out by the ProcessRows procedure, which improves the current candidates by searching the leaves inferred from the nearest neighbours of the patch immediately above the current one on the previous row. An entire row is processed in parallel using two kernels. The first determines, for each patch, the set of new leaf indices to be searched, assigning one thread per patch. The second performs the bruteforce search over these leaves, processing one query per CUDA block, with the block size set equal to the leaf size. The second kernel starts by copying the query and its current KNN into shared memory, it processes one new leaf at a time. For each leaf, the threads of the block first compute the distance from the query to every patch in the leaf. A refinement loop then repeatedly extracts the smallest of these distances and, if it improves on the worst of the currently known neighbours, inserts the candidate into the KNN set and marks that distance as used so the next iteration returns the next best candidate. The loop stops once the best remaining leaf candidate is further away than the worst neighbour, since no more improvements are possible. Once all new leaves are done, the updated KNN set is copied back to global memory. Finally, the SelectBest procedure selects, in parallel and for each query patch, the best of its k nearest neighbours with respect to the original image patches.

3.4 Randomized Approximate Nearest Neighbours

Randomized Approximate Nearest Neighbours (RANN) is a general-purpose approximate nearest neighbour algorithm based on random rotations and KD-trees. Unlike propagation-assisted KD-trees, which exploit the spatial coherence of image patches, RANN is not specific to images. It instead attempts to make nearest neighbour search easier by repeatedly transforming the input space, building a KD-tree in the transformed space, and searching only a small subset of leaves for each query. [8]

Let

$$A = \{y_0, y_1, \ldots, y_{m-1}\} \subset \mathbb{R}^d$$

be the reference set and let

$$B = \{x_0, x_1, \ldots, x_{n-1}\} \subset \mathbb{R}^d$$

be the query set. The goal is to find the k nearest neighbours in A for every query point $x_i \in B$. RANN constructs, for each query x_i , a much smaller candidate set $V_i \subset A$, and only searches this subset. The assumption is that if V_i is selected so that it is likely to contain good nearest neighbour candidates, then an exact search within V_i can give a good approximation at much lower cost.

The algorithm begins by shifting the points so that the reference set is centered around the origin. This is done by subtracting the center of mass of the reference set A from both the reference points and the query points. Afterwards, a pseudorandom orthogonal transformation Θ is applied to all points. The transformation is constructed from a combination of coordinate permutations, pairwise coordinate rotations, and a Fourier-transformation operation. Since the transformation is orthogonal, it preserves Euclidean distances, but it changes the orientation of the point cluster. The purpose is to not distort the nearest neighbour problem, and just change the layout of the points so that a KD-tree split may separate the data in a more useful way.

After the random transformation, a KD-tree is built over the transformed reference set $\Theta(A)$. The tree is constructed level by level. At each level, the current nodes are split by the median value along a selected coordinate, and the coordinate used for splitting changes with the tree level. Searching the tree for a transformed query point $\Theta(x_i)$ first gives the query's natural leaf. It searches neighbouring leaves obtained by changing one bit in the path from the root to the natural leaf. If the tree has height h, this gives the natural leaf plus h related leaves. The union of the points in these leaves forms the candidate set V_i. The algorithm then performs an exact bruteforce search inside V_i and updates the current k best candidates for x_i.

The parameter T controls how many independent randomized trees are used. Each iteration applies a new random orthogonal transformation, builds a new KD-tree, computes new candidate sets V_i, and updates the nearest neighbour candidates. Thus, increasing T increases the number of different transformed views of the data. A larger T generally improves accuracy, because a true nearest neighbour that is missed in one tree may be placed in a useful leaf in another tree. The cost is that the main loop is repeated T times: T random transformations are performed, T KD-trees are built, and T searches are carried out. At a high level, the computational complexity of RANN is therefore described as $n \cdot \log(m) \cdot T$, where n is the number of query points and m is the number of reference points.

RANN may also use an additional refinement step called *supercharging*. If a point $y \in A$ is already a good candidate for a query x_i, then the neighbours of y in the reference set may also be good candidates for x_i. Supercharging therefore searches through the nearest neighbours of the current nearest neighbour candidates. If each query has k candidates and each of those candidates also has k neighbours, then supercharging examines up to k^2 additional candidates per query. This can improve accuracy, but it also increases both runtime and memory use [10]. It additionally requires nearest neighbour information for the reference points themselves, since the algorithm needs to know the neighbours of the candidates in A. For this reason, supercharging is most fitting when the query set and reference set are the same, or when nearest neighbour candidates for the reference set have already been computed.

In the context of this thesis, RANN is useful as a comparison point because it represents a different kind of approximate nearest neighbour strategy. Propagation-assisted KD-trees are image-specific: they use the spatial structure of images and propagate good matches between neighbouring patches. RANN is context-independent. Its candidate sets are produced by randomized transformations and KD-tree structure alone. Comparing RANN with propagation-assisted methods therefore helps to show the difference between generic approximate nearest neighbour search and image-specific propagation.

4 Futhark Language Preliminaries

Most of the implementations presented in this thesis are written in the Futhark language. Futhark is a small purely functional array language designed to be compiled to efficient parallel code. It is statically typed, data-parallel and has a heavily optimising compiler. The compiled code can

be either multi-threaded CPU code or GPU code via either the CUDA or OpenCL backends [3].

This section provides a brief overview of the Futhark Second Order Array Combinators (SOACs) and features used in this thesis.

4.1 Array constructors and combinators

4.1.1 Iota

The `iota` function constructs a 1D array containing consecutive integers from zero up to, but not including, the given size:

$$\text{iota} : (n : i64) \to [n]i64$$

$$\text{iota } n = [0, 1, ..., n-1]$$

For example, `iota 5` returns `[0, 1, 2, 3, 4]`.

4.1.2 Map

Map has the following types and semantics

$$\text{map} : (\alpha \to \beta) \to IIn.[n]\alpha \to [n]\beta$$

$$\text{map } f[a_1, ..., a_n] = [fa_1, ..., fa_n] \text{ [6]}$$

Map takes an array and a function and applies the function to each element in the array argument and thus creates a new array. For instance, given $A = [1, 2, 3, 4]$ and $f\ x = x * 2$ then *map* on A with the function f will return $[2, 4, 6, 8]$.

4.1.3 Reduce

Reduce has the following types and semantics

$$\text{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to IIn.[n]\alpha \to \alpha$$

$$\text{reduce } \oplus 0_\oplus[a_1, ..., a_n] = 0_\oplus \oplus a_1 \oplus ... \oplus a_n \text{ [6]}$$

Reduce takes an array, a neutral element which can also be described as the start value for the accumulator and an associative binary operator $\oplus$. It applies the operator between each element of the input array and the neutral element and stores the tentative result in the accumulator. At the end it returns the finished accumulated result. The type of the neutral element and the elements in the input array must match.
For instance, given $A = [1, 2, 3, 4]$, 0 and + then *reduce* on A, 0 and + will return 10. *reduce* on A, 1 and $*$ will return 24.

4.1.4 Scan

Scan has the following types and semantics

$$\text{scan} : (\alpha \to \alpha \to \alpha) \to \alpha \to IIn.[n]\alpha \to [n]\alpha$$

$$\text{scan} \oplus 0_{\oplus}[a_1, ..., a_n] = [a_1, ..., a_1 \oplus ... \oplus a_n] \text{ [6]}$$

Scan takes the same arguments as reduce. However, where reduce only returns a single value scan returns an array essentially showing all the prefix sums or 'steps' of the reduce. Futhark's scan is inclusive. This just refers to whether the result array begins with the neutral element or the result of the first calculation.
For instance, given $A = [1, 2, 3, 4]$, 0 and + then inclusive *scan* on A, 0 and + will return $[1, 3, 6, 10]$.

4.1.5 Scatter

The `scatter` function constructs a new array by writing a collection of values into specified positions of an existing destination array:

$$\text{scatter} : [k]\alpha \to [n]i64 \to [n]\alpha \to [k]\alpha.$$

Given a destination array `dest`, an array of indices `inds`, and an array of values `vals`, the expression `scatter dest inds vals` returns a copy of `dest` where each value `vals[i]` has been written to position `inds[i]`. For instance, `scatter [0,0,0,0] [1,3] [7,9] = [0,7,0,9]`.

4.1.6 Segmented Scan

Futhark supports regular nested arrays, but irregular nested arrays cannot be represented directly as arrays whose inner dimensions vary. A common strategy is therefore to flatten the irregular structure into a single array and use flags, offsets or segments to retain information about the irregular segmentation. Segmented scans are useful because they allow scan computations to be performed independently inside each logical segment while still operating on a flat array.
In this thesis the following implementation of a segmented Scan is used.

```
24 def segmented_scan [n] 't
25                    (op: t -> t -> t)
26                    (ne: t)
27                    (flags: [n]bool)
28                    (as: [n]t) : *[n]t =
29   (unzip (scan (\(x_flag, x) (y_flag, y) ->
30                   ( x_flag || y_flag
31                   , if y_flag then y else x `op` y
32                   ))
33                 (false, ne)
34                 (zip flags as))).1
```

It takes one more argument than regular scan which is a flag array. Given an array

```
fltarr = [1, 3, 5, 7, 9, 2, 4, 6, 8]
```

a flag array would be an array of the same size that indicates the start of new segments. For instance the flag array

```
flags = [True, False, False, True, True, False, True, False,
                                                    False]
```

for `fltarr` would indicate the segments

```
[[1, 3, 5], [7], [9, 2], [4, 6, 8]].
```

4.2 Time Complexity Analysis

Time complexity analysis for parallel programming is quite different from sequential programming. In sequential programming, runtime is usually described in terms of the total number of operations performed. In parallel programming, this is not sufficient, since many independent operations may be executed simultaneously. It is therefore necessary to distinguish between the work and span of a program.
The work of a program is the total number of operations it performs, while the span is the time required assuming an unlimited number of processors i.e. the length of the longest chain of dependencies. A table of work and span for the presented functions is shown below on table 1.

Function	**Work**	**Span**
iota	$O(n)$	$O(1)$
map	$O(n * W(f))$	$O(S(f))$
reduce	$O(n * W(f))$	$O(\log(n) * W(f))$
scan	$O(n * W(f))$	$O(\log(n) * W(f))$
scatter	$O(n)$	$O(1)$
segmented scan	$O(n * W(f))$	$O(\log(n) * W(f))$

Table 1

$W(f)$ and $S(f)$ denote the work and span of the function parameter, respectively.

4.3 Incremental Flattening

Futhark programs can contain nested parallelism, where a SOAC contains another SOAC, such as a map containing another map. There is no universally optimal compilation scheme for nested parallelism, since the best choice depends on the input data, which is not known at compile time. One can therefore use incremental flattening, which generates multiple code versions for each instance of nested parallelism, guarding each version with a predicate that compares the amount of parallelism of that version against a chosen threshold. [12, 7] These different versions will be arranged in a tuning tree as seen in Figure 2.

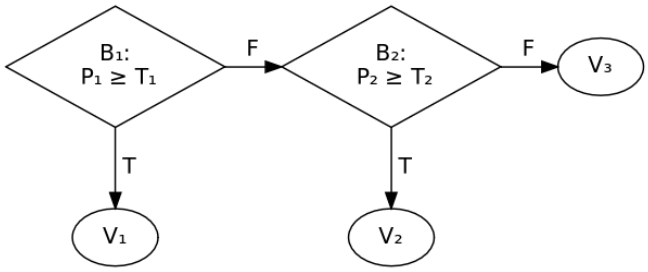


Figure 2: The tuning tree of a nested parallel depth of 2, where P is the degree of parallelism utilized by the code versions, T is the threshold, V is the code version and B is the predicate [12].

For a nested parallel depth of 2, incremental flattening produces three code versions:

- **V1:** The first code version utilizes the parallelism of the outer parallel operation at the current hardware level and then sequentializes its body, which makes this version only utilize global memory.

- **V2:** The second code version also utilizes the parallelism of the outer parallel operation at the current hardware level, and tries to parallelize the body by going a level deeper in the hardware hierarchy. This means the outer parallel operation is mapped to the Cuda grid and the inner parallel work is mapped to the Cuda block level, where the inner parallel work is utilizing shared memory.
- **V3:** The third code version recursively continues the flattening procedure at the current hardware level, maintaining the arrays in global memory, and if the nested parallel depth is greater than two, this produces additional code versions.

At runtime, Futhark checks the predicates in order and selects the first version whose parallelism is at least the threshold. In our implementation we force V2 (intra group) parallelism using the #[incremental_flattening(only_intra)] annotation in 3 kernels FindNaturalLeaves, ExactKNN and Propagation. (see section 5.3.3, 5.3.2, 5.3.4)

5 Implementation of Data-Parallel Irregular KD-Trees

5.1 Overview

This chapter presents the new data-parallel implementation of propagation-assisted KD-trees with irregular leaf sizes for ANNF computation using flat parallelism. Moving forward we will refer to the data-parallel implementation by Oancea et al. as `CUDAold` and our implementation as `CUDAnew`.
In `CUDAold` splitting at the middle index of a sorted sequence introduces a subtle bug when a node contains many patches whose value along the splitting dimension equals the median. The middle element split ends up putting some of these equal patches in the left child and the rest in the right child. This breaks the KD-tree's invariant, namely that every point in the left subtree is strictly less than the median and every point in the right is greater than or equal to it. Since `findNaturalLeaf` compares a query against the median and descends accordingly, a query sitting right at the median can end up in a leaf that does not hold the points it is actually closest to. The sequential version avoids this because its median is a value and not an index number. We therefore propose a data-parallel version that acts like the sequential version. This change is simple enough at the algorithmic level but substantial at the implementation level.

The full procedure still follows the same high level algorithm as `CUDAold` shown in Algorithm 1. Our modifications are concentrated in the parts that handle irregular leaf sizes, the KD-tree construction (5.2) and the irregular

leaf access within the search functions (5.3). In what follows, we describe the full algorithm except the parts which required no editing at all, which are the first three steps of the algorithm: creating the patches, creating the PCA model and reducing the dimensionality.

5.2 Building the KD-Tree

The function `mkKDtree` constructs an irregular KD-tree from the reference patches in a data-parallel fashion. Its input is the tree height, the number of internal tree nodes `q`, and the reference patches array `input` of size $m \times d$. The size of the input array is not padded like in `CUDAold`. Its output consists of the reordered reference patches, the final shape array of the leaves, the indirect index array, and three arrays describing the internal split nodes: the split dimension, the median split value, and the closest ancestor that split the same dimension.

```
let mkKDtree [m] [d] (height: i32) (q: i64)
                     (input: [m][d]f32) :
           (*[m][d]f32, *[q+1]i64, *[m]i32,
            *[q]i32, *[q]f32, *[q]i32) =
```

Before the construction loop starts, the implementation computes the initial lower and upper bounds of the full set of reference patches. These upper and lower bounds are updated at each iteration of the construction loop, and are used to calculate the spread for each node's patches in every dimension. These spreads are later used to identify the split dimension for each node.
The remaining arrays are initialized with neutral values. The indirect array `indir` initially stores the original permutation of the reference patches. The shape array `shp` is initialized with all m patches in the root and zero patches in every other position and with a full length of the final number of leaves `q+1`.

```
let indir       = map (\x -> i32.i64 x) (iota m)
let shp         = scatter (replicate (q+1) 0i64) [0] [m]
let median_vals = replicate q 0.0f32
let median_dims = replicate q (-1i32)
let clanc_eqdim = replicate q (-1i32)
```

The central new attribute of the KD-tree is the shape array, `shp`. Each iteration in the construction loop slices a `2x` bigger chunk of the shape array as the active prefix. At any given level (`lev`) the active prefix of `shp` stores

the number of patches contained in each node at the current level. The length of the sliced `shp` array is always equal to the number of nodes at the current level and the sum of values in the array is always equal to the total number of reference patches.

```
for lev < (height+1) do
    let nodes_this_lvl = 1i64 << i64.i32 lev
    let cur_shp = shp[0:nodes_this_lvl]
```

For instance, if `cur_shp` = [3, 5], then the current level contains two nodes. The first one has 3 patches and the second 5 patches. In `CUDAold` you would always know that the split is [4, 4], but now in `CUDAnew` this information has to be represented explicitly.

After updating the bounds and calculating spreads and split dimensions, the next step is to isolate the split dimension values per node in order to perform rank-k search on them. We need a way to determine which patches belong to which node. We build the helper arrays needed in the flat segmented operations with `mkBFlagsII`. This function is an implementation of the procedure described in the section *Jagged (Irregular Multi-Dim) Array Representation* in [13].

```
let mkBFlagsII (size: i64) (nodes_this_lvl: i64) (shp: []i64)
    : ([]u32, []i64, []i64) =
  let (B1, F1) = mkIrFlagArray (map u32.i64 shp) 0 (map (\x
  ↪  -> x+1) (iota nodes_this_lvl))
  let F1fix = F1 :> [size]i64
  let Farr = map bool.i64 F1
  let II1 = segmented_scan (+) 0 Farr F1 :> [size]i64
  in (B1, F1fix, II1)
```

Here, `B1` contains the beginning offset of each non-empty node segment. `F1fix` is a flag array where the first element of segment i is marked with $i + 1$. The array `II1` is produced by a segmented scan over these flags, so every patch receives the one-indexed id of the node segment it belongs to. The helper function `mkIrFlagArray` is omitted for brevity.

```
let (offsetInds, flags, II1) = mkBFlagsII m nodes_this_lvl
↪  cur_shp
let seg_ids = map (\x -> x - 1) II1
```

The subtraction in `seg_ids` gives zero-indexed segment ids. Consider the irregular shape array `[2, 3, 0, 1, 2, 0, 0, 2]`, `mkBFlagsII`'s returns and `seg_ids` would be:

```
B1      = [0, 2, 5, 5, 6, 8, 8, 8]
flags   = [1, 0, 2, 0, 0, 4, 5, 0, 8, 0]
II1     = [1, 1, 2, 2, 2, 4, 5, 5, 8, 8]
seg_ids = [0, 0, 1, 1, 1, 3, 4, 4, 7, 7]
```

It is essential that the segment ids numbering skips the numbers corresponding to the empty nodes. Otherwise `CUDAnew` would not be able to correctly index into the shape array.
Once every patch has a segment id, `CUDAnew` extracts the dimension value that will be used for splitting.

```
let chosen_columns =
    map2 (\ind seg ->
            input[ind, med_dims[seg]]
         ) indir seg_ids
```

Here `indir` maps the current position in `input` back to the original reference patch, while `seg_ids` selects the correct split dimension from that patch's node. If `cur_shp = [3,5]` and `med_dims = [2,0]`, then the first three patches read coordinate 2, while the next five read coordinate 0.

5.2.1 Splitting Nodes with Rank-k Search and Partition 2L

Rank-k finds for each node the value at rank $\lfloor n/2 \rfloor$, where n is the node size. The rank-k implementation used is from Lynghus's RANN implementation [10]. These searches are performed over the flat `chosen_columns` array using the shape, offset, flag, and segment-id arrays.

```
let med_vals =
    computeMedianWithRankK cur_shp chosen_columns (map
    ↪  i64.u32 offsetInds) flags II1
```

Rank-k search repeatedly chooses candidate pivots and counts, per segment, how many values are smaller than, equal to, or greater than the pivot. These counts determine whether the wanted rank lies below the pivot, at the pivot, or above it. Finished segments keep their result, while unfinished

segments continue with the remaining candidate range. The result is an array `med_vals` containing the median value for each node.

All patches are then classified by the predicates `<` and `>=` according to the median value of their segment, as can be seen below.

```
let bools = map2 (\x seg -> x < med_vals[seg])
         chosen_columns seg_ids
```

A value of `true` means that the patch should be placed in the left child, while a value of `false` means that it should be placed in the right child. The actual reordering is performed by `partition2L`. This is a lifted version of a normal two-way partition, which takes one flat array and places all elements satisfying the predicate before the remaining elements. The example in *Part II: Flattening Nested and Irregular Parallelism* from [13] shows `partition2` returning the split index and reordered patches from a bool array and a patches array.

```
conds = [F,T,F,T,F,F,T]
xss = [1,2,3,4,5,6,7]
partition2 conds 0 xss => (3, [2,4,7,1,3,5,6])
```

The lifted version applies the same operation independently to every segment in an irregular flat array with segmented scans, and can thus be run in parallel.

```
let indir = map i64.i32 indir
let (splitInds, (_, indir')) = partition2L bools 0i64
↪  (cur_shp, indir)
let indir' = map i32.i64 indir'
```

The returned `splitInds` array stores, for each node, how many patches satisfied the predicate, and `indir'` contains the reordered patches partitioned within their segment.

Finally, the shape array is updated for the next level. If a node of length `len` has `ind` patches going left, then its left child has size `ind`, and its right child has size `len - ind`. Empty nodes remain empty.

```
let cur_shp' = map2 (\len ind ->
                       if len == 0
                       then [len] ++ [len]
                       else [ind] ++ [len - ind])
                       cur_shp splitInds64
               |> flatten

let shp' = scatter shp (iota (nodes_this_lvl * 2)) cur_shp'
```

For instance, if

```
cur_shp   = [3, 5]
```

and the partition returns

```
splitInds = [1, 4]
```

then the next level shape becomes

```
cur_shp' = [1, 2, 4, 1]
```

The first parent node contributes children of size 1 and 2, while the second parent node contributes children of size 4 and 1. The children are not forced to be equally sized anymore. Their sizes are determined by the actual outcome of the partitioning step.

5.2.2 The Rank-K Edge Case for Homogeneous leaves

The irregular KD-tree implementation `CUDAnew` introduces an interesting edge case. The construction of the KD-tree relies on repeatedly choosing a split dimension and a splitting value, which should be the median in that dimension. Ideally, this divides the current set of patches into two subsets approximately equal in size. However, this notion can break when many patches in a node have the same value in the split dimension.
In the irregular implementation, if many patches are identical or nearly identical along the chosen splitting dimension, then a split such as $x < median$ may send very few or no patches to the left child, while $x >= median$ sends almost all or all patches to the right child. This also happens if we change the split predicates to be $<=$ and $>$. In both cases, one child may become empty or nearly empty.
The worst case is when more than 50% of the patches in a node is equal to the minimum value in the node. In this case the median value will become equal to the minimum value, and every single point will therefore go to the

right child.

This behaviour matters because the irregular tree no longer uses the middle index as the median. As a result, homogeneous data can lead to very unbalanced subtrees, large leaves and empty leaves through repeated ineffective splits. The effect of this is shown in 7.9.

5.3 Searching the KD-Tree

This section describes the search phase of `CUDAnew`. After the irregular KD-tree has been constructed, the search phase must account for the new representation of the tree. We first describe the parallel brute-force search used to refine a candidate list within a single irregular leaf, since this routine is shared by all later search stages. We then explain how queries are assigned initial candidates from their natural leaves, how exact KD-tree traversal is used for the first row, and how propagation improves the candidate list.

5.3.1 BruteForce

The brute force search of a single leaf is implemented by the function bruteForceParIrreg (given to us by our supervisor). We describe it in detail here because it is critical to the performance of `CUDAnew` and is used at several stages of the algorithm (see FindingNaturalLeaves, ExactKNN and Propagate.)

```
let bruteForceParIrreg [m][d][k]
      (query: [d]f32)
      (knn0: [k](i32,f32))
      (beg: i64)
      (len: i64)
      (avg_leavesize: i64)
      (refs: [m][d]f32)
    : [k](i32,f32) = #[unsafe]
```

Bruteforce updates the k nearest neighbours of a single query by searching one leaf. It takes the query point, its current list of candidates (`knn0`) where each candidate is a tuple holding the index of the candidate point in the reference array together with its squared distance to the query, the index of the first point of the leaf within the reference array (`beg`), the number of points in the leaf (`len`), the average leaf size (`avg_leavesize`) and the reference array (`refs`). The candidate list is sorted in ascending order of distance, and the function returns the updated candidate list.

The two constants Q and B, in the code snippet below, control how the work of one leaf is divided between threads.

```
9  let Q = 4i64
10 let B = (avg_leavesize + Q - 1) / Q
```

A Q value of 4 allows an average leaf size of up to 4096, since B must at most be 1024, the maximum threads in a CUDA block. Every time Bruteforce is called, it is run with intra group parallelism (See 5.3.3, 5.3.4, 5.3.2) meaning each call is mapped to a single CUDA block and the work inside is carried out by the threads of that block. `Q` is the sequentialisation factor which determines the number of points each thread is responsible for. Here it is fixed at 4. `B` is the CUDA block size, which determines the number of threads needed to cover one chunk of `avg_leavesize` points when each thread handles `Q` of them. `Q` contributes with 2 benefits: Firstly it reduces the inter-thread communication for reduce and scan operations. Secondly it allows scaling to a larger leaf size, in which the effects are shown in 7.7.

```
11 loop knn = copy knn0
12   for qq < (len + avg_leavesize - 1) / avg_leavesize do
13     let offset = qq * avg_leavesize
14     let fdist q =
15           let ind = offset + q + beg
16           in  if ind < len + beg
17               then sumSqrsSeq query refs[ind]
18               else f32.highest
19     let dists = map fdist (0 ... B-1) ++
20                 map fdist (B ... 2*B-1) ++
21                 map fdist (2*B ... 3*B-1) ++
22                 map fdist (3*B ... 4*B-1)
```

Since the leaves can be irregular in this implementation, `len` may exceed `avg_leavesize`. The outer loop therefore partitions the leaf into chunks and processes them one after another, only a leaf greater than the `avg_leavesize` requires more than one iteration. The candidate list is copied once and carried through the loop, so each iteration sees the updates made by the previous one.

The function `fdist` computes the distance from the query to a single reference point. It first computes a index from `q`, where `beg` accounts for the position of the leaf within the reference array. If the index lies within the leaf, it returns the sum of squared differences between the two points. If the index is out of range for the leaf, it returns f32.highest so an out of range position never can be chosen as a candidate.

The array `dists` holds the distances for all points of the chunk. It is computed over four separate maps, over the ranges `(0 ... B-1)`, `(B ... 2*B-1)`, `(2*B ... 3*B-1)` and `(3*B ... 4*B-1)`. The four maps are concatenated into one distance array, this is optimized by the Futhark compiler utilizing a short-circuiting memory optimization [11]. Instead of allocating 4 intermediate arrays for each map and copying the result to a destination array, each map is computed directly in a slice of the destination array, which reduces the amount of shared memory used.

```
23      let cycle = true
24      let j = 0i32
25      let (_, knn, _, _) =
26        loop (dists, knn, j, cycle)
27          while cycle && (j < (i32.i64 k)) do
28            let fmap li =
29                loop (midx, mval) = (i32.highest,
                ↪  f32.highest)
30                  for i < Q do
31                    let pt_idx = i * B + li
32                    let dis = dists[pt_idx]
33                    in if dis < mval then (i32.i64 (offset +
                      ↪  pt_idx), dis)
34                    else (midx, mval)
```

Within the outer loop an inner `while` loop extracts the best candidate from the chunk and inserts it into the list of candidates. The loop carries four values, the distance array (`dists`), the candidate list (`knn`), a counter (`j`) of how many candidates have been inserted and a Boolean (`cycle`) that allows for an early termination of the loop. The loop only continues while `cycle` is true and fewer than `k` candidates have been updated.
Finding the smallest distance in the chunk is a two-step process. The first step is sequential and is performed by the function `fmap`, which computes the `Q` points belonging to thread `li`. It performs `Q` iterations, reading one point per iteration and keeping the smallest distance out of the `Q` points together with that points index. After `Q` iterations it returns the smallest distance among that thread's points.

```
35            let (min_ind, min_val) =
36              map fmap (iota B) |>
37              reduce_comm (\ (i1,v1) (i2,v2) ->
38                             if v1 < v2 then (i1, v1) else
39                             if v1 > v2 then (i2, v2) else
40                             (if i1 <= i2 then i1 else i2, v1)
41                           ) (i32.highest, f32.highest)
42            in  if min_val < (knn[k-1-(i64.i32 j)].1)
43                then  let dists[min_ind-(i32.i64 offset)] =
                      ↪  f32.highest
44                      let knn[k-1-(i64.i32 j)] = (min_ind +
                        ↪  i32.i64 beg, min_val)
45                      in  (dists, knn, j+1, true)
46                else  (dists, knn, j, false)
47      let knn_sort = sortPartSortedSeqs knn
48      in  knn_sort
```

The second step is done in parallel, by mapping `fmap` over `iota B`, which makes one thread compute `Q` points. Reduce_comm then combine these B values into the single smallest distance in the chunk. The final step is the insertion logic, each chosen minimum point is overwritten in the `dists` array with f32.highest, so the following iterations of the while loop finds the second smallest distance point in the chunk. We check the candidate list from the worst end, on iteration `j` the chunks `j`-th smallest distance is compared against the `knn[k-1-j]` value. If the new distance is smaller, two updates take place. First, `dists[min_ind - offset]` is set to `f32.highest`, as explained above. Second, the candidate is written into `knn[k-1-j]`. The loop then continues with `j+1` and `cycle` remaining true. If instead the new distance is not smaller than `knn[k-1-j]`, the loop stops by setting `cycle` to false. This early termination happens because the distances arrive in ascending order while the candidate compared against grow smaller. Once a chunk distance fails to beat `knn[k-1-j]`, every later distance is at least as large and every still remaining original candidate is at least as small, so no further update will achieve anything. After the loop terminates, the candidate list is sorted and the result becomes `knn` for the next chunk, and after the final chunk it is the updated candidate list returned by `bruteForceParIrreg`. Please note that the last sorting step, implemented by `sortPartSortedSeqs`, is challenging to parallelize because it uses induction variables that are conditionally incremented on some of the control-flow paths [15], therefore it is executed sequentially.

5.3.2 Find Natural Leaf

After the KD-tree is constructed `findNaturalLeaves` computes, for each query, an initial set of k nearest neighbour candidates from the leaf the query naturally belongs to. It does so using the array of reference points, the shape array, the median split dimensions, the median split values and the array of queries.

```
49  let leaf_offsets = exScan (+) 0i64 shp
50  let avg_leavesize = (reduce (+) 0i64 shp) / length(shp)
51  let query_leaves0 = map (findLeaf median_dims median_vals
    ↪  h) queries
52  let knns0 = imap2intra (\leaf_ind query ->
53                      let beg = leaf_offsets[leaf_ind]
54                      let len = shp[leaf_ind]
55                      in  bruteForceParIrreg query
56                          (replicate k (-1i32, f32.highest))
57                          beg len avg_leavesize ref_pts
58                    ) query_leaves0 queries
```

After computing the leaf offset and average leaf size, the function determines each query's natural leaf using the `findLeaf` helper function, which walks the tree from root to leaf, comparing the query's coordinate in the split dimension against the median value at each internal node and branching left or right accordingly. In `CUDAold` a memory locality optimization was implemented by reordering the queries by their natural leaf using radix sort. Sorted queries that share a leaf are processed consecutively, allowing the leaf data to be reused from the L1 cache [16]. This optimization has not been preserved in this implementation due to time constraints. Each query computes an initial list of k candidates by bruteforce searching its natural leaf, using `bruteForceParIrreg` (see 5.3.1). To utilize parallelism even when the leaves differ in size, the function `imap2intra` is used, defined as:

```
1  def imap2intra f as bs =
   ↪  #[incremental_flattening(only_intra)] map2 f as bs
```

It instructs the compiler to only use intra-group mapping for the map function (4.3), meaning the outer map is assigned to the CUDA grid, with one block per query, and the bruteforce search runs across the threads of that block. The remaining lines unpack the knn tuples and return the index array, distance array, and the array of natural leaves.

5.3.3 Exact KNN

After each query has found its natural leaf, and an initial set of k candidates, `exactKnn` finds the exact k nearest neighbours for the first row of queries

through a full tree traversal.

```
let exactKnn [m][q][d][n][k][p]
             (ref_pts: [m][d]f32)
             (shp: [p]i64)
             (kd_tree: [q](i32,f32,i32))
             (queries: [n][d]f32)
             (nat_leaves: *[n]i32)
             (knns:   *[n][k](i32,f32)) : (*[n][k](i32,f32),
             ↪  i32) =
```

The function takes the reference points, a triple called kd_trees which consists of the array of median split dimensions, median split values, and the closest previous ancestors with equal split dimensions. It also takes the array of queries, the natural leaves as traversal starting points, and the candidate lists from Find Natural Leaf, and returns the updated candidate list.

```
  let (ord_knns', _, _, _, loop_count, _) =
      loop  (knns, last_leaves, stacks, dists, i, n')
        while n' > 0 do
          let wnns = map (\arr -> arr[k-1].1) knns
          let (new_leaves, new_stacks, new_dists) = unzip3 <|
            map2 (traverseOnce h kd_tree) (zip queries wnns)
             ↪  (zip3 last_leaves stacks dists)
          let n'' = map (\leaf_ind -> if leaf_ind < i32.i64
           ↪  num_leaves then 1i32 else 0i32) new_leaves
                |> reduce_comm (+) 0i32
                |> opaque
```

ExactKNN's outer loop iterates until every query for the first row has finished its traversal. Each iteration performs three steps:

1. For each query we extract the worst k-neighbour distance. Any subtree whose minimum distance to the query exceeds this value can be skipped.

2. We call TraverseOnce which performs one step of kd-tree traversal. Starting from the previously visited leaf (initialized to the natural leaf), it backtracks up the tree until it finds a sibling subtree worth visiting, then descends down to a new leaf which it returns. If no further subtree is worth visiting, it returns an out of range value indicating that the query's traversal is complete.

3. For queries with a valid new leaf, we refine the knn via bruteforce, for queries whose traversal has completed, their knn is returned unchanged.

```
let knns' =
imap3intra (\query knn leaf_ind ->
               let is_valid = leaf_ind < i32.i64 num_leaves
               let beg = if (!is_valid) then 0 else
               ↪  leaf_offsets[i64.i32 leaf_ind]
               let len = if (!is_valid) then 0 else
               ↪  shp[i64.i32 leaf_ind]
               let result = bruteForceParIrreg query knn beg
               ↪  len avg_leavesize ref_pts
               in if (is_valid) then result else knn
        ) queries knns new_leaves
```

Our contribution is the following: for each query, `is_valid` checks whether the new leaf index is in range. A valid query has its candidate list updated by using `bruteForceParIrreg` on the newly found leaf, an invalid query returns its list unchanged. The map is launched with `imap3intra`, the three argument version of `imap2intra` from Find Natural Leaf, which forces intra group flattening. The final lines of `exactKnn` return the exact k nearest neighbours for the first row of the image.

5.3.4 Propagation

The last part of the algorithm is propagation. This is where each k nearest neighbours found in the row above is mapped back to its original image coordinate. Then each coordinate is moved down one row and a search is performed in the leaf containing that projected point.

After the first row of queries has been computed exactly, the remaining rows initial KNN candidates from `findNaturalLeaves` are improved using propagation. The purpose of propagation is to exploit the spatial coherence of images.

```
let estimateIndex [m] (n_rows: i32) (n_cols: i32)
                  (indir: [m]i32) (orig2leaf: [m]i32)
                  (par_ind: i32) : i32 =
  let orig_par_ind = indir[par_ind]
  let orig_par_y = orig_par_ind / n_cols
  let orig_par_x = orig_par_ind - orig_par_y * n_cols
  let cand_x = orig_par_x
  let cand_y = if orig_par_y < (n_rows-1) then orig_par_y+1
   ↪  else n_rows-1
  let orig_cand_ind = cand_y * n_cols + cand_x
  let cand_leaf_ind = orig2leaf[orig_cand_ind]
  in  cand_leaf_ind
```

The helper function `estimateIndex` performs the actual propagation step. It takes a candidate index from the previous row's KNN list named `par_ind`. This index refers to the reordered reference array used by the KD-tree, so the first step is to translate it back into the original image using `indir`. From this original index, the function reconstructs the 2D image coordinates by computing the x and y-coordinates. The candidate is then propagated one row down by keeping the same x and incrementing the y, unless the point is already in the last row. Finally, the new original image index is translated into a leaf index using `orig2leaf`.

```
let propagate [m][d][p][nr][nc][k]
              (ref_pts:   [m][d]f32)
              (indir:     [m]i32)
              (shp:       [p]i64)
              (orig2leaf: [m]i32)
              (queries:   [nr][nc][d]f32)
              (nat_leaves:[nr][nc]i32)
              (knns:      *[nr][nc][k](i32,f32))
            : *[nr][nc][k](i32,f32) =
  let leaf_offsets = exScan (+) 0i64 shp
  let avg_leavesize = (reduce (+) 0i64 shp) / length shp
```

The `propagate` function takes the reference points, the indirect array, the irregular leaf shape array, the `orig2leaf` mapping, the image-shaped query array, the natural leaf of each query, and the current KNN candidates. The shape array shp is used to compute `leaf_offsets`, which gives the starting position of each leaf in the reordered reference array. As in the other search procedures, `avg_leavesize` is computed from the shape array and passed to `bruteForceParIrreg`.

```
23  let knns' =
24    loop (knns) for im1 < nr-1 do
25      let i = im1 + 1
26      let upw_knn_inds = map (map (.0)) knns[im1]
27      let cur_nodes = opaque <| copy knns[i]
```

The outer loop in `Propagate` processes the image row by row, starting from the second row. The first row has already been handled by `ExactKNN`, and each later row is improved using the completed KNN results from the row immediately above it. For a row i, `upw_knn_inds` extracts the KNN indices from row $i-1$. These are the parent candidates that will be projected downwards. The current row's KNNs are copied into `cur_nodes`. These already contain the candidates found by searching the natural leaf of each query, and propagation will only improve them further.

```
28      let (n_leavess, to_search_leavess) =
29        opaque <| unzip <|
30        map2 (\(par_inds0: [k]i32) (nat_leaf: i32) : (i32,
          ↪  [k]i32) ->
31                let par_inds = opaque (copy par_inds0)
32                let u_leaves = replicate k (-1i32)
33                let n_inds = 0i32
34
35                let (n_inds, u_leaves) =
36                  loop (n_inds, u_leaves)
37                    for q < k do
38                      let par_ind = estimateIndex (i32.i64
                        ↪  nr) (i32.i64 nc) indir orig2leaf
                        ↪  (par_inds[q])
39                      let not_found = par_ind != nat_leaf
```

For each query in the current row, the function computes which leaves should be searched. Each of the k candidates from the query above is passed through `estimateIndex`, producing a propagated leaf index. The natural leaf is ignored, since that leaf has already been searched in `findNaturalLeaves`.

```
40                      let j = 0i32
41                      let (_,not_found) =
42                        loop (j,not_found)
43                          while not_found && j < n_inds do
44                            if u_leaves[j] == par_ind
45                            then (j,   false)
46                            else (j+1, true)
47                      in  if not_found
48                          then  let u_leaves[n_inds] =
                            ↪  par_ind
49                                in  (n_inds+1, u_leaves)
50                          else  (n_inds, u_leaves)
51
52                  in (n_inds, u_leaves)
53              ) upw_knn_inds nat_leaves[i]
```

The propagated leaves are also deduplicated. Since several candidates from the row above may project into the same leaf, searching all of them would be redundant work. The array `u_leaves` stores the unique leaves that should be searched for a query, while `n_inds` records how many valid leaves were found. The size of `u_leaves` is k because at most one leaf can be generated from each of the k propagated candidates.

```
54      let new_knn_row =
55        opaque <|
56        imap4intra (\(n_inds: i32) (u_leaves: [k]i32) (knn0:
          ↪  [k](i32,f32)) (query: [d]f32) ->
57              let knn = opaque <| copy knn0
58              in loop (knn) for q < n_inds do
59                   let leaf_ind = u_leaves[q]
60                   let beg = leaf_offsets[i64.i32 leaf_ind]
61                   let len = shp[i64.i32 leaf_ind]
62                   in bruteForceParIrreg query knn beg len
                     ↪  avg_leavesize ref_pts
63            ) n_leavess to_search_leavess cur_nodes
              ↪  queries[i]
```

Once the set of propagated leaves has been found, the current row is improved in parallel. The function `imap4intra` maps over the queries of the row, their current KNNs, the number of propagated leaves, and the list of leaves to search. For each query, it loops over the propagated leaves and calls `bruteForceParIrreg` on each of them. The beginning of the leaf is looked up in `leaf_offsets`, and the length of the leaf is looked up in `shp`.

Each call to `bruteForceParIrreg` updates the current KNN list if better candidates are found in the propagated leaf. Since `bruteForceParIrreg` returns a sorted KNN list, the result of one propagated leaf can be passed directly in the next propagated leaf search.

```
64        let knns[i] = new_knn_row
65        in knns
66
67    in knns'
```

Finally, the updated row is written back into the full KNN array. Because the outer loop carries the KNNs as its loop variable, the result of propagating to row i is available when processing row $i + 1$. Propagation therefore proceeds down through the image, where each row benefits of the improved candidates of the row above. This makes propagation cheaper than a full exact traversal for every query, while still allowing good candidates to spread through the image.

6 Framework Integration

Within the existing framework from Oancea et al. [16], we have extended it with our irregular implementation as well as with a data-parallel implementation of Randomized Approximate Nearest Neighbours from Lynghus [10]. The existing framework, already had three algorithms implemented in the framework, namely openmp, a CPU Multi Core implementation of the proposed algorithm by He and Sun [5] implemented by Oancea et al., `CUDAold` and a naive algorithm for calculating the exact nearest neighbour as a baseline. This section will go over the structure of the framework, and the changes needed to allow for the extension of the framework.

6.1 Framework Structure

The general structure of the parent directory of the framework is as seen below. Note that openclirreg corresponds to `CUDAnew` and openclopt corresponds to `CUDAold`:

```
annfmp/
├── annfmp/
│   ├── naive/
│   ├── openclirreg/
│   ├── openclopt/
│   ├── openmp/
│   ├── rann/
│   ├── sklearn/
│   ├── __init__.py
│   ├── base.py
│   └── setup.py
├── examples
├── data
├── docs/
├── examples
├── scripts
└── setup.py
```

The framework is implemented with three different programming languages, Python, C and Futhark for the data parallel implementations. The interface used to establish communication between C and Python is SWIG, which allows functions defined and implemented in C to be called from other programming languages. Each algorithm using Futhark has the following directory structure:

```
openclirreg/
├── src/
│   ├── futhark/
│   │   └── (Futhark parallel code)
│   ├── include/
│   │   └── base.h
│   └── base.c
├── SWIG/
│   ├── float.i
│   └── numpy.i
├── __init__.py
├── base.py
└── setup.py
```

To enable this communication between Futhark, C and Python, a few files are needed. First, the Futhark file containing the required entry points is compiled with the `--library` flag, which generates a C library containing a `.c` and `.h` file pair that provides access to those entry points. Next, the `base.c` file in the algorithm directory wraps the exposed entry points into a smaller API, defined in `src/include/base.h`. This is the API that `SWIG` uses to pass parameters and results between Python and the Futhark

code. SWIG's pipeline works as follows. First, one writes an interface file for SWIG that defines the module name, specifies how Numpy arrays from Python are expanded into C arguments, and exposes the API declared in base.h. This process is required for every algorithm that contains Futhark parallel code, in this case CUDAold, CUDAnew and rann. When the setup code in the parent directory is run, SWIG processes the interface file in each algorithm folder and generates a C wrapper and Python wrapper. The C compiler then produces a shared library (.so file) by linking these wrappers with the corresponding base.c file. At runtime, base.py in each algorithm directory imports that shared library, enabling calls to the exposed API and creating a class for that specific algorithm. Finally, the base.py file in the parent directory combines all of these algorithm specific classes into a single class called ANNField, which allows selection between the algorithms and instantiates them.

6.2 Integration of Randomized Approximate Nearest Neighbours

In addition to CUDAnew, the framework was extended with a data-parallel implementation of Randomized Approximate Nearest Neighbours. The purpose of this integration was to make RANN executable from the same Python interface as the other ANNF strategies. To make its output comparable with the other strategies and be able to adjust and tune the algorithm, we had to extend the original Futhark driver by exposing the supercharge variation as well as making the KD-tree height an explicit parameter and computing the final L2 accuracy score.

6.2.1 Entry Points

We added separate entry points for the ordinary and supercharged variants. This made it possible to compile and call both versions through the same wrapper mechanism used by the other backends.

```
entry main [m] [n] [d]
           (Tval: i32) (k: i64) (h: i32)
           (test_set: [m][d]f32) (queries: [n][d]f32) =
  RANN Tval k h test_set queries

entry mainSuper [m] [n] [d]
                (Tval: i32) (k: i64) (h: i32)
                (test_set: [m][d]f32) (queries: [n][d]f32) =
  superRANN Tval k h test_set queries
```

As can be seen in the entry points we also turned the height of the KD-tree into a parameter. Before our change the way the height of the tree was computed was by always taking the $\log_2$ of the number of reference data points divided by 256. This way the height was always set according to a target points-per-leaf of 256.

```
let height = ( log2Int (m / 256))
```

This is a reasonable default, but it is an important experimental parameter in this study, which motivated the change.

6.2.2 L2 accuracy metric

The original RANN implementation returns the approximate nearest neighbour indices, but the framework evaluates all algorithms using the original image patches and their true L2 distance. Therefore, this integration also adds a separate `selectBestNN` entry point. Like in the other algorithms this function isolates the candidate with the smallest squared L2 distance. This means RANN can be evaluated using the same accuracy metric as `CUDAold`, `CUDAnew`, and `openmp`.

7 Experiments

7.1 Methodology

This section presents a comparative analysis of the four different ANNF implementations: `OpenMP`, `CUDAold`, `CUDAnew` and `RANN` in which we demonstrate the strengths and weaknesses along L2 score and runtime.

7.1.1 Hardware and Datasets

All the experiments were run on NVIDIA A100 GPUs available through UCPH's Futhark servers [14].

There are two datasets used in the experiments: the 133 image pairs from movie trailers in 1920 resolution `Vidpairs` [4] and another dataset of 155 image pairs from movie trailers `Vidpairs4k` [17] in the resolutions `500p, 720p, 1080p, 1440p and 2160p`.
The algorithms used are compiled with Futhark 0.25.24 and the Cuda backend except the CPU variant.

7.1.2 Experimental Design

We mainly tweak four parameters: the level of PCA reduction, the patch size, the points per leaf, and the image resolution. PCA affects the number

of dimensions, patch size affects both the dimensions and the input size, and resolution affects only the input size.
`OpenMP` and `CUDAold` take a leaf size, whereas `CUDAnew` and RANN take a height. To account for this, the experiment scripts convert the target points per leaf (PPL) to the average PPL for that image. Throughout the experiments we list the target PPL, but the script takes that target, calculates the height that gives an average PPL closest to the target. It then passes the average PPL to `OpenMP` and `CUDAold` while passing the corresponding height to RANN and `CUDAnew`, so that all algorithms run with the approximately same average PPL. This is calculated for every image pair, since images in the Vidpairs[4] and Vidpairs4k[17] datasets can have different patch counts because the image height can differ whilst the image width is consistent.

7.2 Impact of Points Per Leaf

In this experiment, RANN operates on the full dimensionality of the image patches, which is determined by the patch size. With a patch size of 4x4, this gives 48 dimensions, following the formula $d = p^2 \cdot c$, where p is the flattened patch size and c is the number of colour channels (here 3, for RGB). Accordingly, the KD-Tree algorithms were given a PCA rotation of the data and remained at 48 dimensions, ensuring that both families of algorithms operate on the same number of dimensions. We then sweep the leaf size to determine its effect on both the L2 score and the run time.

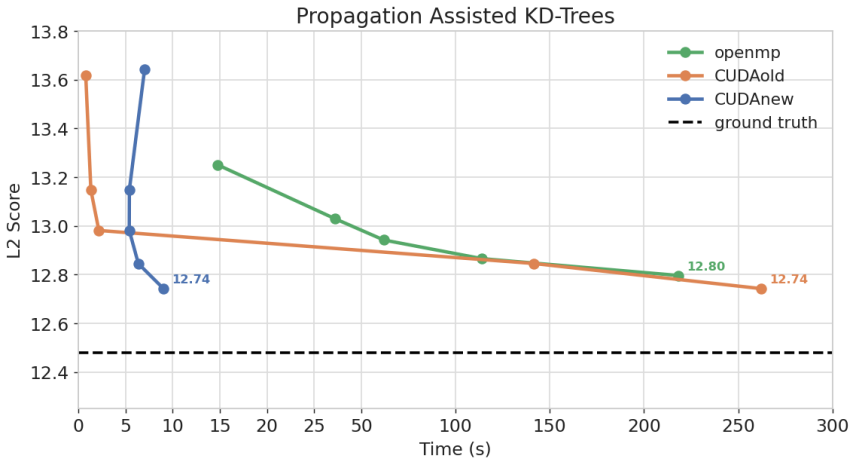


Figure 3: Comparison between the Propagation Assisted KD-Tree algorithms using the VidPairs dataset[4] with all 133 image pairs. The patch size was fixed to 4x4 and PCA 48. All methods had to find k = 8 nearest neighbours. Markers on the algorithms curve represent different target Points per leaf (128, 512, 1024, 2048, 4096).

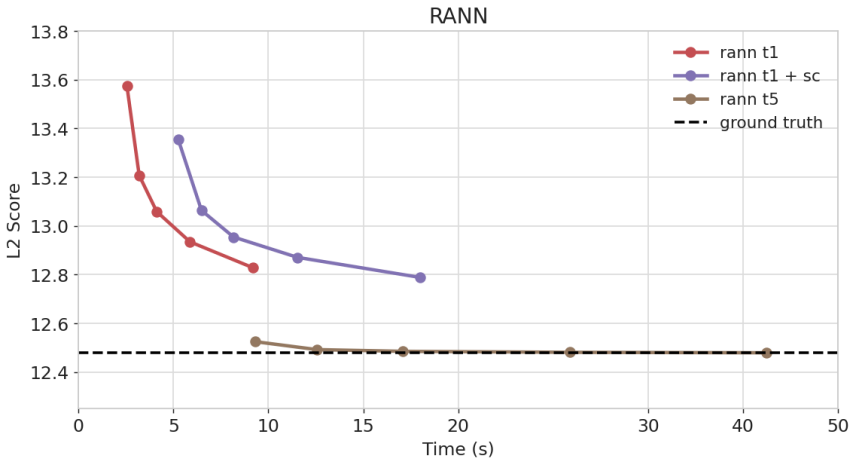


Figure 4: Comparison between the RANN algorithm with different settings using the VidPairs dataset[4] with all 133 image pairs. The patch size was fixed to 4x4. All methods had to find k = 8 nearest neighbours. Markers on the algorithms curve represent different target Points per leaf (128, 512, 1024, 2048, 4096).

As shown in Figure 3, all of the KD-Tree algorithms achieve a better L2 score as the leaf size increases. From the smallest target PPL of 128 to the largest at 4096. Runtime, however, degrades with larger leaf sizes for all methods, and `openmp` and `CUDAold` scale particularly poorly once the leaf size exceeds 1024. This is because their bruteforce step is not optimised to handle such large leaf sizes, unlike `CUDAnew`. By combining incremental flattening and the use of a sequentialization factor `Q` in the bruteforce stage, `CUDAnew` scales well, with only a minimal increase in run time across leaf sizes. At a leaf size of 4096, `CUDAnew` is around 25 times faster than both `CUDAold` and `openmp`.
Turning to RANN in Figure 4, the parameter T is the dominant factor in the L2 difference: $T = 5$ comes close to the ground truth even at the smallest leaf size and approaches it further as the leaf size grows. It is, however, also the largest contributor to run time. By comparison, $T = 1$ is roughly 3 times faster than $T = 5$ at the smallest leaf size and, although less accurate, its L2 score remains within 1.5 points of the ground truth at that leaf size. Enabling supercharge with $T = 1$ gives a slightly better L2 score at smaller leaf sizes than $T = 1$ without it, but at larger leaf sizes the improvement becomes negligible while the run time is roughly 1.5 times worse than without supercharge.
A common trend across all algorithms is that increasing the leaf size improves the L2 score, but at a substantial run time cost, unless the algorithm is specifically optimised to handle large leaf sizes, as `CUDAnew` is.

7.3 PCA on RANN

RANN, as described, does not rely on reducing dimensionality with PCA to avoid the curse of dimensionality, since it does not perform a full tree traversal. Figure 5 sweeps RANN over patch sizes without PCA, plotting runtime against L2 for each patch configuration, as the patch size grows the runtime increases steeply from 48 dimensions (4x4 patches) to 192 dimensions (8x8) to 768 dimensions (16x16).

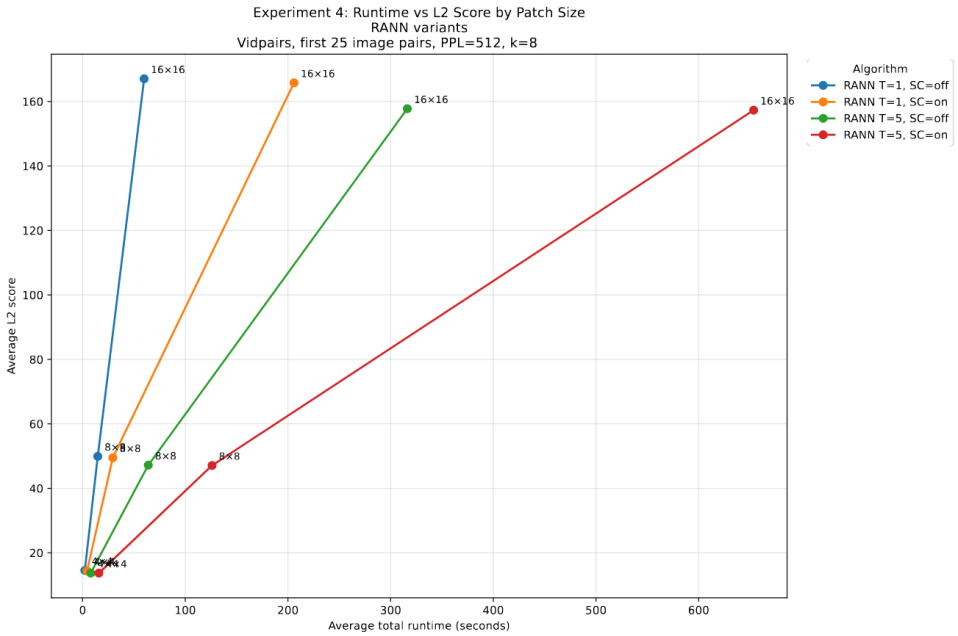


Figure 5: Comparison between RANN with patchsize (4x4,8x8,16x16) on full dimensionality. With T value 1 and 5. The comparison is run on the first 25 image pairs of the Vidpairs dataset[4].

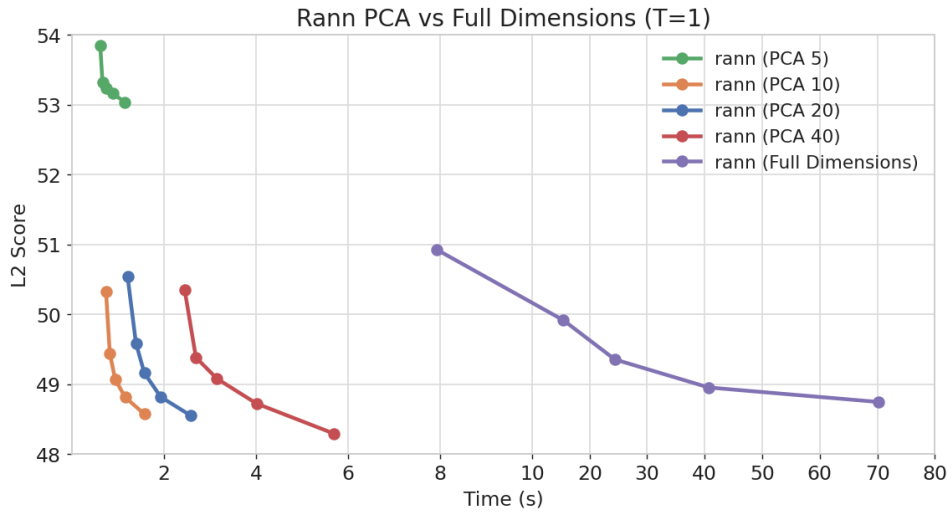


Figure 6: Comparison between RANN with PCA (5,10,20,40) vs RANN with no PCA i.e Full Dimensionality. T = 1, PatchSize 8x8, Sweeping over leaf size (128,512,1024,2048,4096). The comparison is run on the first 25 image pairs of the Vidpairs dataset[4].

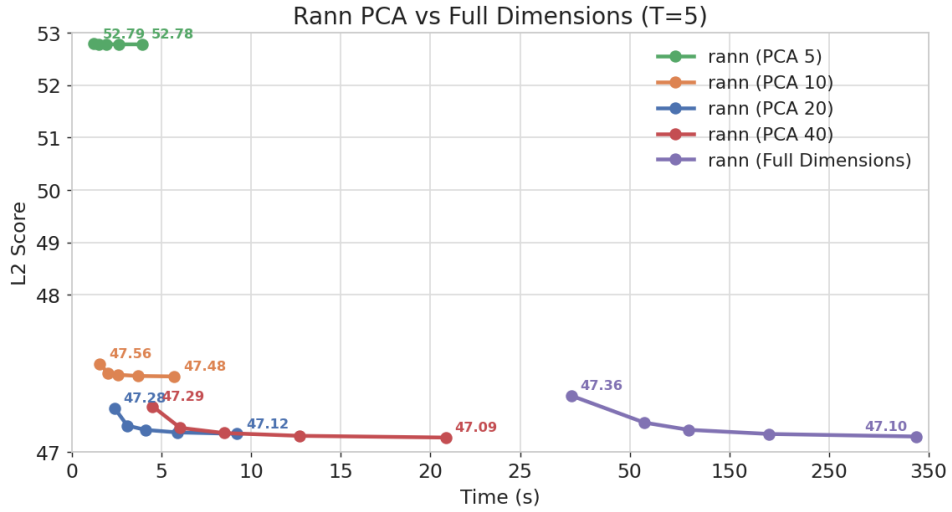
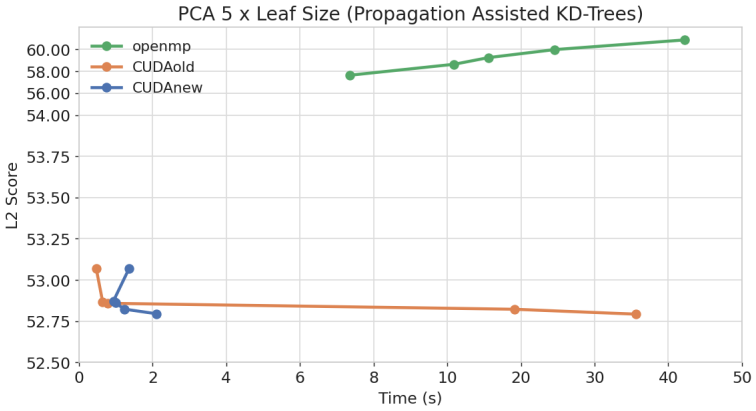


Figure 7: Comparison between RANN With PCA (5,10,20,40) vs RANN with no PCA i.e Full Dimensionality. T = 5, PatchSize 8x8, Sweeping over leaf size (128,512,1024,2048,4096). The comparison is run on the first 25 image pairs of the Vidpairs dataset[4].

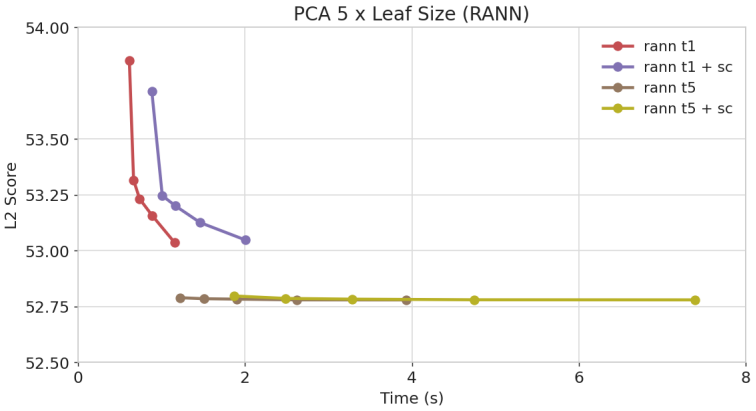
Figures 6 and 7 fix the patch size at 8×8 and compare RANN at full dimensionality against PCA reductions (5, 10, 20, and 40), sweeping leaf sizes (128, 512, 1024, 2048, and 4096) with T = 1 in Figure 6 and T =

5 in Figure 7. Reducing the dimensionality lowers runtime relative to full dimensionality for both T = 1 and T = 5. PCA (10, 20, 40) with T = 1 and PCA (40) with T = 5 achieve a better L2 score across all leaf sizes. This improvement indicates that high dimensional points carry substantial noise. Once the low variance dimensions are discarded, RANN can find nearest neighbours more reliably. Very aggressive reduction (PCA 5) discards useful dimensions, which in turn hurts accuracy, whereas moderate reductions (PCA 10, 20, 40) trim mainly noise. All PCA reductions give a runtime improvement of between 1.5x and 9x over full dimensionality, depending on leaf size and T value.

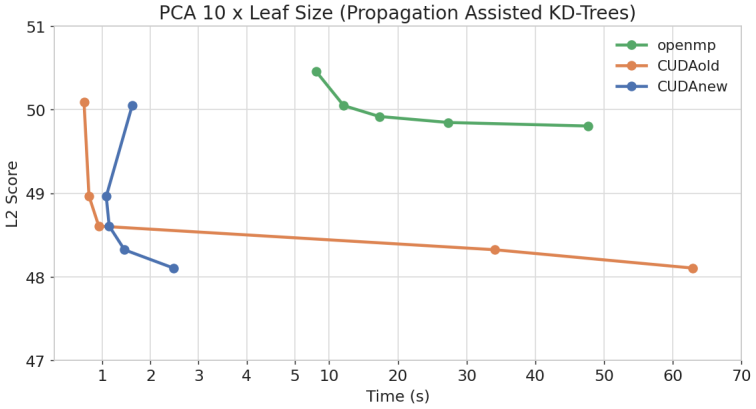
7.4 PCA and Points Per Leaf



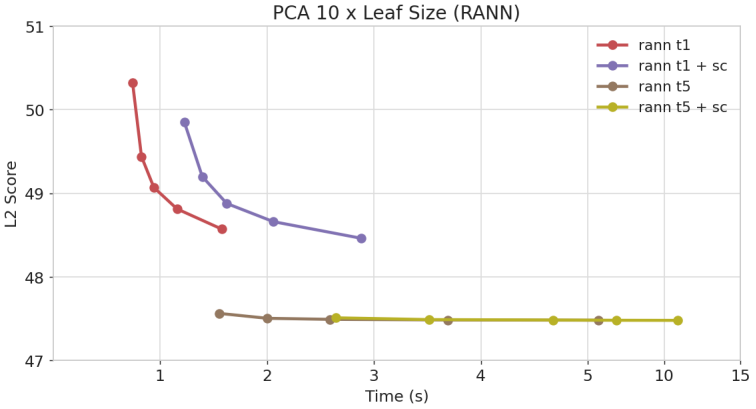
(a) PCA 5 Leaf Size Sweep (KD-Tree)



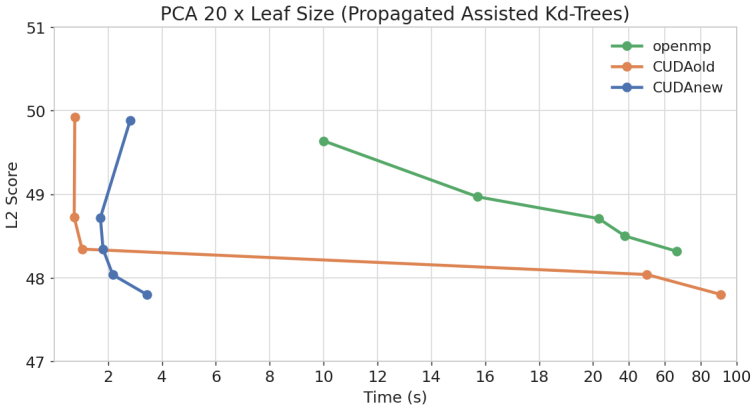
(b) PCA 5 Leaf Size Sweep (RANN)



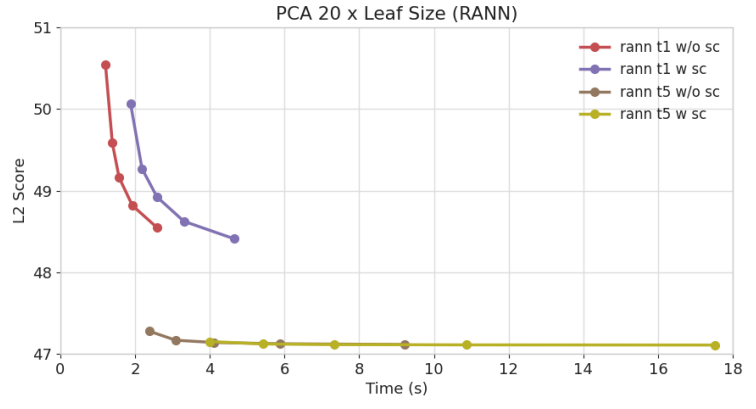
(c) PCA 10 Leaf Size Sweep (KD-Tree)



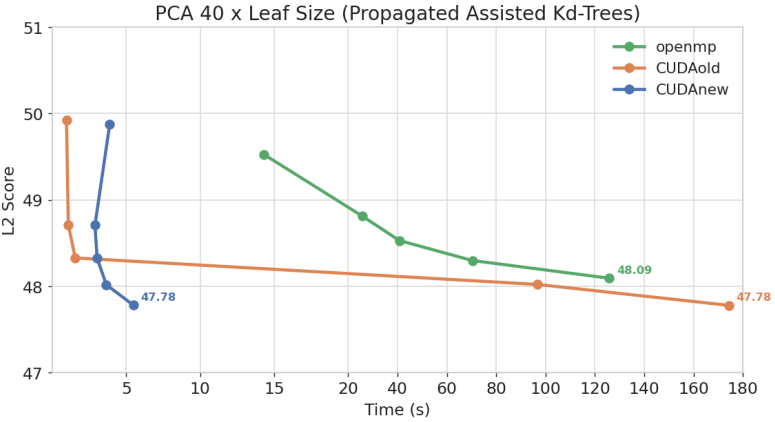
(d) PCA 10 Leaf Size Sweep (RANN)



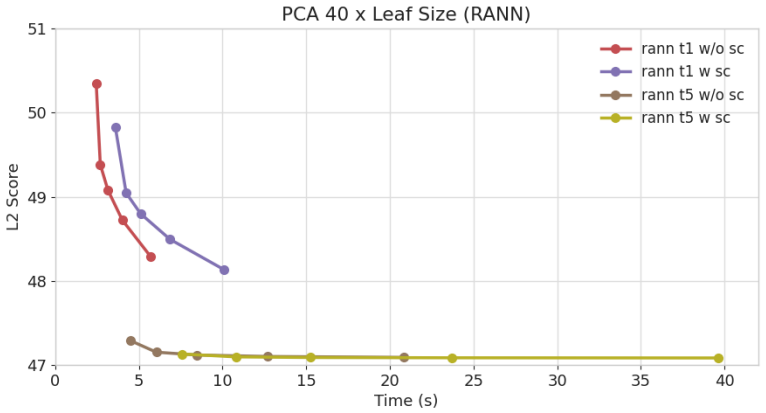
(e) PCA 20 Leaf Size Sweep (KD-Tree)



(f) PCA 20 Leaf Size Sweep (RANN)



(g) PCA 40 Leaf Size Sweep (KD-Tree)



(h) PCA 40 Leaf Size Sweep (RANN)

Figure 8: Comparison between competing algorithms using the first 25 image pairs from the VidPairs dataset[4]. The patch size was fixed to 8x8, each graph pair is using a different PCA reduction (5,10,20,40). All methods had to find k = 8 nearest neighbours. Markers on the curves represent the average L2 and runtime for the PPL (128,512,1024,2048,4096).

The results show that the effect of the leaf size differs greatly between the propagation-assisted implementations and RANN. For the propagation-assisted implementations, increasing the leaf size generally increases runtime, especially for `CUDAold` and `openmp`. This is expected, since larger leaves require more candidate patches to be evaluated during traversal. The CPU-based `openmp` implementation is consistently slower than the GPU-based implementations and also produces worse L2 scores for everything but the lowest leaf sizes. `CUDAold`, on the other hand, benefits from larger leaves in terms of accuracy: as the leaf size increases, its L2 score improves steadily, but at a significant runtime cost when points per leaf reaches more than 1024, where the work cannot be done with a single CUDA block anymore. The accuracy-runtime tradeoff, is most visible for PCA 20 and PCA 40, where `CUDAold` reaches some of the best propagation-assisted L2 scores but only at much higher runtimes.

The irregular KD-tree implementation, `CUDAnew`, behaves very differently. Across PCA 10, 20, and 40, it stays in a very small runtime range while still improving in L2 score as the leaf size changes. In particular, it consistently reaches L2 scores equivalent to `CUDAold`, while remaining much faster on higher leaf sizes. This shows the strength of the `Q` optimization that allows `CUDAnew` to scale much better with leaf sizes. The exception is PCA 5, where all propagation-assisted GPU methods achieve similar L2 values around 53, while `openmp` performs much worse.

The RANN results show a strong separation between the effect of $T = 1$ and $T = 5$. We learned in the first experiment that $T = 5$ can match the baseline for most leaf sizes, which is why increasing the leaf size cannot give a lower L2 score. Therefore the effect of leaf sizes and supercharging can only really be observed in full when looking at $T = 1$. Our findings are consistent with what Lynghus found: that T value is the most impactful parameter [10].

Overall, the experiment shows that `CUDAnew` with high leaf size and `RANN T5` with low leaf size compete for the strongest compromise between runtime and accuracy, with `RANN T5` being a bit slower with the best accuracy and `CUDAnew` being a bit faster with an inferior accuracy.

7.5 Resolution Scaling

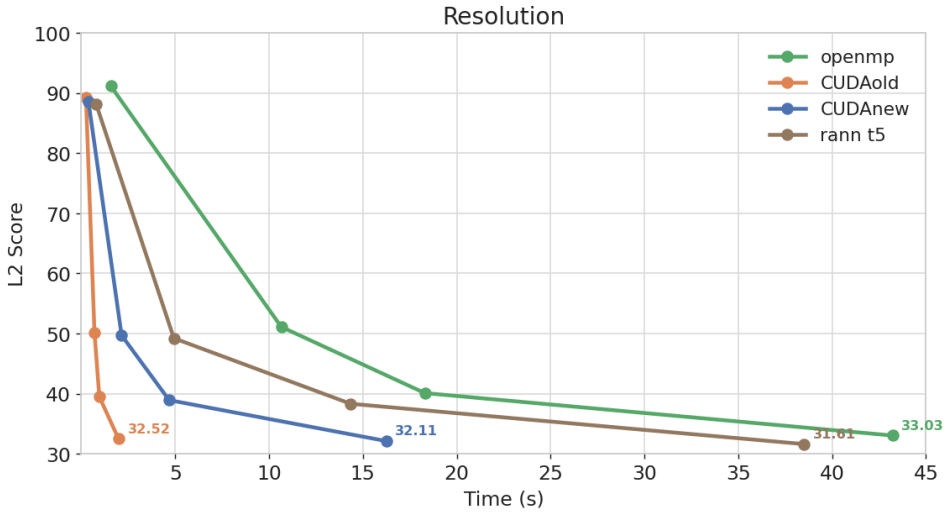


Figure 9: Comparison between competing algorithms using the first 25 image pairs from the VidPairs4K dataset[17] in the following resolutions: (500p, 720p, 1080p, 1440p). The patch size was fixed to 8x8, PCA reduction 40. All methods had to find k = 8 nearest neighbours. Markers on the curve represent the average L2 and runtime for the 4 different resolutions. The PPL for each algorithm was chosen based on the best L2 and runtime trade off found in the earlier experiments.

Figure 9 shows each algorithm as a curve whose four markers correspond to increasing image resolution (500p,720p, 1080p and 1440p). The downward trend in L2 as resolutions rises is primarily a dataset effect. All four algorithms have nearly the same L2 at each resolution. The real distinction lies in the runtime, where the spread is substantial. `CUDAold` scales best, remaining under 2 seconds even at 1440p. `Openmp` scales the worst at around

43 seconds, whilst `rann` and `CUDAnew` lie in between. Each algorithm was tuned to its strongest setting in the compromise between runtime and accuracy with `Openmp` at 128 points per leaf, `CUDAold` at 512, `CUDAnew` at 2048 and `rann` at 128.

7.6 Impact of T value

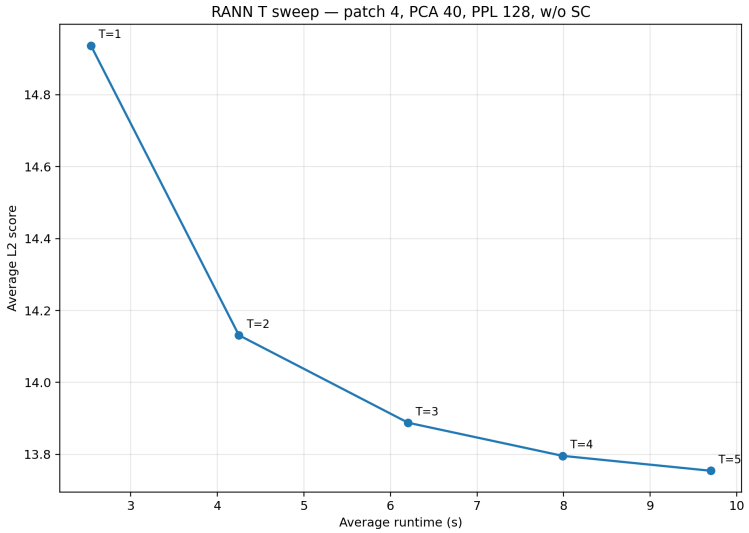


Figure 10: Sweep over T-values for RANN without Supercharge. Comparison between RANN With T values (1,2,3,4,5). PCA 40, PatchSize 4x4, and leaf size 128). The comparison is run on the first 25 image pairs of the Vidpairs dataset[4].

The T sweep shows that increasing the number of randomized trees improves the average L2 score, but with clear diminishing returns. The largest improvement occurs when increasing from $T = 1$ to $T = 2$, where the L2 score drops substantially. Further increases to $T = 3$, $T = 4$, and $T = 5$ continue to improve the result, but the gains become progressively smaller.

At the same time, runtime increases linearly with T a little less than two seconds per T, since each additional tree introduces more traversal and candidate generation work. The curve therefore exposes the central tradeoff in RANN: higher T improves accuracy by increasing candidate diversity, but this comes at a runtime cost. In this configuration, $T = 5$ gives the best L2 score, but $T = 3$ or $T = 4$ may represent a more attractive compromise if runtime is prioritized, since they capture most of the accuracy improvement while avoiding the full cost of $T = 5$.

7.7 Impact of Q

As introduced in the bruteforce section, `Q` is the sequentialization factor, which determines the amount of sequential work each thread executes in the bruteforce function 5.3.1.

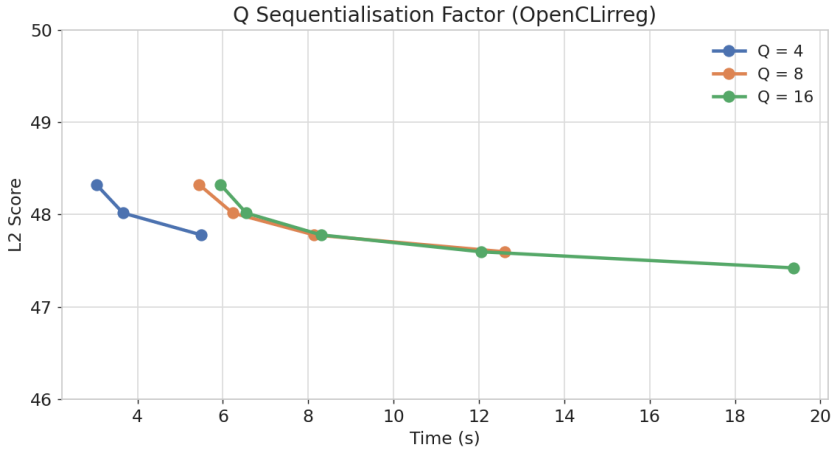


Figure 11: Comparison between `CUDAnew`, where each curve is a Q factor size (4,8,16). The patch size was fixed to 8x8 and PCA reduction 40. Markers on the curve represent leaf sizes (1024, 2048, 4096, 8192, 16384). The comparison is run on the first 25 image pairs of Vidpairs[4]

As can be seen in Figure 11, the Q factor makes it possible to run the algorithm with higher leaf sizes. When Q is 4, the maximum possible leaf size is 4096, since Q directly affects the CUDA block size (see 5.3.1), when the leaf size exceeds 4096 for Q = 4, the CUDA block size exceeds 1024, which is not permissible. As demonstrated in previous experiments, a higher leaf size gives better L2 scores, however, as Figure 11 shows, running with a leaf size of 16384 when Q = 16 produces only a negligible improvement in L2 on this dataset compared to a leaf size of 4096 when Q = 4, at the cost of a roughly 3 time increase in runtime. The figure also shows that Q = 4 at leaf size 1024 is substantially faster than leaf size 1024 with Q = 8 or Q = 16. This effect is most pronounced when comparing Q = 4 against Q = 8 and Q = 16. When comparing Q = 8 against Q = 16, however, Q = 16 is only slightly slower than Q = 8 at leaf sizes 1024 and 2048, and is slightly faster at leaf size 8192. Q should therefore be tuned to match the maximum leaf size one expects to use, using a smaller Q when possible.

7.8 Accuracy on Image Pairs where `OpenMP` outperforms `CUDAold`

This is an experiment on 20 selected image pairs from Vidpairs4k, where `CUDAold` has a > 0.5 worse accuracy score than `OpenMP`. We investigate how

CUDAnew performs in these special cases. The image pairs were found on 128 points per leaf, PCA 40, patch size 8 and searching through the 500p and 720p pictures.

Implementation	Average L2 score
OpenMP	70.686
CUDAold	71.364
CUDAnew	71.379

Table 2: Average L2 scores on the 20 selected image pairs from Vidpairs4k, where CUDAold has more than 0.5 worse accuracy than OpenMP.

As can be seen on Table 2 CUDAnew does not on average perform better than CUDAold when OpenMP is significantly better. It is worth noting that on the individual pictures CUDAnew beat CUDAold 10 out of 20 times, so the advantage of CUDAold seems very situational.
This experiment tests whether the semantic incongruence in CUDAold is responsible for its worse L2 score on these 20 image pairs. CUDAnew eliminates the incongruence but achieves no better L2 score, which does not support this hypothesis.

7.9 The Rank-K Edge Case for Homogeneous leaves

To illustrate what exactly the edge case 5.2.2 looks like we show the following extract from the shape array for the image pair 55 and 56 in the 1920p resolution from Vidpairs4k [17]. The accuracy score and runtime are pasted at the end.

```
...39, 0, 0, 0, 31351, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 41761, 637, 656, 606, 703, 118, 1173, 675, 686, 335,
482, 671, 1290, 692, 698, 695, 696, 328, 336, 155, 516, 88,
104, 231, 14724, 0, 0, 0, 0, 0, 0, 0, 19406, 1432, 1434, 1410,
1559, 1436, 1482, 1461, 1461, 1291, 1297, 1323, 1811, 1488,
1488, 1488, 1489, 1459, 1459, 1459, 1460, 1459, 1460, 1459,
1460, 1459, 1460, 1459, 1460, 1459, 1460, 1459, 1460, 49, 57,
53, 56, 33, 74, 30, 78, 53, 53, 50, 57, 51, 58, 55, 56, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 48323...

CUDAnew L2 score: 34.68472671508789
CUDAnew runtime: 12.994828224182129 seconds
```

The shape array shows evenly split leaves with $\approx 1400 - 1500$ patches, but also a large amount of empty leaves and huge leaves containing upwards of 48323 patches in a single leaf.

One attempted solution to this imbalance was a rank-k fix, where instead of always splitting exactly at the median, the implementation tries to detect cases where the median does not produce an even partition. If too many points have the same value as the median, the split value can be adjusted by choosing another nearby ranked value instead. Conceptually, this tries to avoid choosing a split value that places almost all points on the same side of the predicate.

A second attempted solution was dimension swapping. When the best splitting dimension is too homogeneous, the implementation can try another dimension with a slightly smaller spread. Maybe a dimension with a smaller spread but a more even split is preferable. In our experiments we allowed the implementation to peek at the 3 other dimensions with the largest spread.

Applying both ideas we got a new shape array that looks like this:

```
...1787, 1367, 1223, 1084, 31298, 31, 2, 1, 14, 5, 10, 8,
14596, 18, 110, 59, 410, 408, 357, 353, 55, 53, 56, 40, 57,
45, 56, 44, 52, 51, 58, 40, 53, 51, 50, 46, 1422, 1420,
1421, 1420, 1422, 1420, 1421, 1419, 1426, 1419, 1419, 1418,
1421, 1420, 1420, 1419, 1523, 1420, 1513, 1233, 1713, 1348,
1452, 1161, 1444, 1398, 1421, 1419, 1421, 1420, 1420, 1419,
34912, 36, 58, 39, 121, 111, 116, 114, 197, 37, 118, 101,
231, 62, 83, 75, 5513, 233, 148, 146, 146, 140, 141, 139,
306, 305, 304, 303, 305, 304, 304, 303, 32678, 263, 588, 579,
643, 635...

CUDAnew L2 score: 34.68598556518555
CUDAnew runtime: 11.91391921043396 seconds
```

This shape array is notably different. There are no longer any empty leaves, although there would be if the height incremented once. There are still huge leaves of upwards to 32678 patches, but this can hardly be mitigated if they are all equal in all dimensions. Overall the shape array is much more balanced. The L2 score here is equivalent, but the runtime in this particular example is more than a second faster. This could be due to the leaves being more even, and big leaves are slow to bruteforce.

These fixes illustrate a trade-off in the irregular representation of the KD-tree. A more aggressive splitting strategy like this can produce a better balanced tree and may improve runtime. However, it may also change the structure of the tree in ways that do not necessarily improve nearest-neighbour quality. Further empirical investigation is needed in this area. None of the proposed fixes was used in the previous experiments.

8 Future Work

Several improvements can be made on the work presented in this thesis. The experimental framework could be expanded further with an implementation of Coherency Sensitive Hashing. This would add another image-specific ANN method to the comparison. Also running some of the experiments on more than just the first 25 image pairs in Vidpairs would strengthen the results substantially, since there is quite a lot of variance between the images in terms of runtime on the different implementations.

Another relevant extension would be to restore the memory optimization used in `CUDAold`'s `findNaturalLeaves` as discussed earlier in 5.3. This would make this step ≈ 3 times faster, but the step only accounts for 9.2% of the total runtime on 1920p images according to [16, TABLE I].

Finally, further investigation into better node splitting strategies could improve `CUDAnew`'s tree quality. The rank-k fix and dimension-swapping strategies presented needs further testing and development. This direction of improvement possibly only improves runtime and leaves the accuracy largely unaffected.

9 Conclusion

This thesis investigated Approximate Nearest Neighbour Field computation for image patches on massively parallel hardware. Starting from the existing data-parallel propagation-assisted KD-tree implementation, `CUDAold`, we implemented `CUDAnew`: an irregular KD-tree variant that avoids enforcing fixed-size leaves and instead preserves the split rule of the He and Sun sequential algorithm more closely [5]. This required changing both the KD-tree construction and the search procedures to support irregular leaf sizes through a shape array, irregular leaf offsets, and a redesigned bruteforce leaf search. We also extended the testing framework with an implementation of Randomized Approximate Nearest Neighbour (RANN) search, making it possible to compare it with `openmp`, `CUDAold` and `CUDAnew` under the same runtime and accuracy metric.

The experiments show that `CUDAnew` is a strong compromise between runtime and accuracy among the propagation-assisted KD-tree methods. It scales well in many respects, be it dimensionality, input size and in particular leaf size. `CUDAold` can improve its L2 score by increasing the leaf size, but this comes at a substantial runtime cost once the leaf size exceeds what can be handled efficiently within a single CUDA block. By contrast `CUDAnew` remains much more stable as the effective leaf size grows, mainly due to the `Q` optimization in the irregular bruteforce search. However, the irregular

representation also exposes a rank-k/median edge case where, when many points share the same split value, the tree can become highly unbalanced and produce very large leaves and tiny leaves. This shows that preserving the algorithmic split rule alone might not be sufficient to ensure the best splits when operating on images, where identical values occur as soon as a uniform colour appear anywhere on an image.

RANN turned out to be highly competitive in the comparative study. We showed that a T value of 5 was able to match the exact baseline in some experiments and achieved the best L2 scores we have seen. This comes with additional runtime costs. Supercharging has a smaller effect than the choice of T, confirming that the number of randomized trees is the dominant accuracy parameter.

All in all we have demonstrated that irregular propagation-assisted KD-trees are viable on the GPU and offers a competitive runtime-accuracy trade-off.

References

[1] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B. Goldman. "PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing". In: *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 1st ed. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9798400708978. URL: https://doi.org/10.1145/3596711.3596777.

[2] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. "Locality-sensitive hashing scheme based on p-stable distributions". In: *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. SCG '04. Brooklyn, New York, USA: Association for Computing Machinery, 2004, pp. 253–262. ISBN: 1581138857. DOI: 10.1145/997817.997857. URL: https://doi.org/10.1145/997817.997857.

[3] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. DIKU, 2018. URL: https://app.readthedocs.org/projects/futhark-book/downloads/pdf/latest/.

[4] Fabian Gieseke. *annmp*. Private GitLab repository. Source of the Vid-pairs dataset with 133 image pairs used in this thesis. Available at https://gitlab.com/fgieseke/annmp. Access restricted. Accessed: 2026-06-09. n.d.

[5] Kaiming He and Jian Sun. "Computing Nearest-Neighbor Fields via Propagation-Assisted KD-Trees". In: *Proceedings / CVPR, IEEE Computer Society Conference on Computer Vision and Pattern Recogni-*

tion. IEEE Computer Society Conference on Computer Vision and Pattern Recognition. June 2012, pp. 111–118. ISBN: 978-1-4673-1226-4. DOI: 10.1109/CVPR.2012.6247665.

[6] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. "Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates". In: *Proceedings of the 2017 ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354.

[7] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. "Incremental Flattening for Nested Data Parallelism". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: ACM, 2019, pp. 53–67. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295707. URL: http://doi.acm.org/10.1145/3293883.3295707.

[8] Peter W. Jones, Andrei Osipov, and Vladimir Rokhlin. "Randomized approximate nearest neighbors algorithm". In: *Proceedings of the National Academy of Sciences* 108.38 (2011), pp. 15679–15686. DOI: 10.1073/pnas.1110094108.

[9] Simon Korman and Shai Avidan. "Coherency Sensitive Hashing". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.6 (2016), pp. 1099–1112. DOI: 10.1109/TPAMI.2015.2477814.

[10] Tjørn Lynghus. "Data-Parallel Implementation of Randomized Approximate Nearest Neighbours". Bachelor's thesis. Department of Computer Science, University of Copenhagen, June 2023.

[11] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea. "Memory Optimizations in an Array Language". In: *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2022, pp. 424–438. URL: https://doi.ieeecomputersociety.org/.

[12] Philip Munksgaard, Svend Lund Breddam, Troels Henriksen, Fabian Cristian Gieseke, and Cosmin Oancea. "Dataset Sensitive Autotuning of Multi-versioned Code Based on Monotonic Properties". In: *Trends in Functional Programming. TFP 2021. Lecture Notes in Computer Science*. Ed. by Viktória Zsók and John Hughes. Vol. 12834. Cham: Springer International Publishing, 2021, pp. 3–23. ISBN: 978-3-030-83978-9. DOI: 10.1007/978-3-030-83978-9_1.

[13] Cosmin E. Oancea. *Flattening Irregular Nested Parallelism*. DPP Lecture Slides. Lecture slides. Dec. 2025. URL: https://github.com/diku-dk/dpp-e2025-pub/blob/main/slides/L3and4-irreg-flattening.pdf.

[14] Cosmin E. Oancea. *Lecture Notes for the Software Track of the PMPH Course*. University of Copenhagen, 2018. URL: https://hjemmesider.diku.dk/~zgh600/Publications/lecture-notes-pmph.pdf.

[15] Cosmin E. Oancea and Lawrence Rauchwerger. "Scalable conditional induction variables (CIV) analysis". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 213–224. DOI: 10.1109/CGO.2015.7054201.

[16] Cosmin Eugen Oancea, Ties Robroek, and Fabian Gieseke. "Approximate Nearest-Neighbour Fields via Massively-Parallel Propagation-Assisted K-D Trees". In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 5172–5181. DOI: 10.1109/BigData50022.2020.9378426.

[17] Sipondo. *vidpairs4k*. https://github.com/Sipondo/vidpairs4k. GitHub repository. Vidpairs 4K dataset with 155 image pairs collected from movie trailers from YouTube. Accessed: 2026-06-09. 2025.

A AI Declaration

Version 1

Deklaration for anvendelse af generative AI-værktøjer (studerende)

☒ **Jeg/vi har benyttet generativ AI som hjælpemiddel/værktøj** *(sæt kryds)*

☐ **Jeg/vi har IKKE benyttet generativ AI som hjælpemiddel/værktøj** *(sæt kryds)*

Hvis brug af generativ AI er tilladt til eksamen, men du ikke har benyttet det i din opgave, skal du blot krydse af, at du ikke har brugt GAI, og behøver ikke at udfylde resten.

Oplist, hvilke GAI-værktøjer der er benyttet, inkl. link til platformen (hvis muligt):

chatgpt.com
claude.ai

Beskriv hvordan generativ AI er anvendt i opgaven:

1) *Formål (hvad har du/I brugt værktøjet til)*
2) *Arbejdsfase (hvornår i arbejdsprocessen har du/I brugt GAI)*
3) *Hvad gjorde du/I med outputtet (herunder også, om du/I har redigeret outputtet og arbejdet videre med det)*

1) Vi har brugt generativ AI til rettelse af grammatik (tegnsætning og stavefejl) samt til forklaring af koncepter og eksisterende kode.
2) Vi har brugt det løbende gennem arbejdsprocessen, men mest i den afsluttende fase til rettelse af grammatiske fejl.

3) Til det grammatiske brug har vi gennemgået de foreslåede rettelser og anvendt dem, vi vurderede var gavnlige for opgaven.

NB. GAI-genereret indhold brugt som kilde i opgaven kræver korrekt brug af citationstegn og kildehenvisning. Læs retningslinjer fra Københavns Universitetsbibliotek på KUnet.