FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Accelerating Nearest-Neighbours via KD-Trees on GPUs

Author:

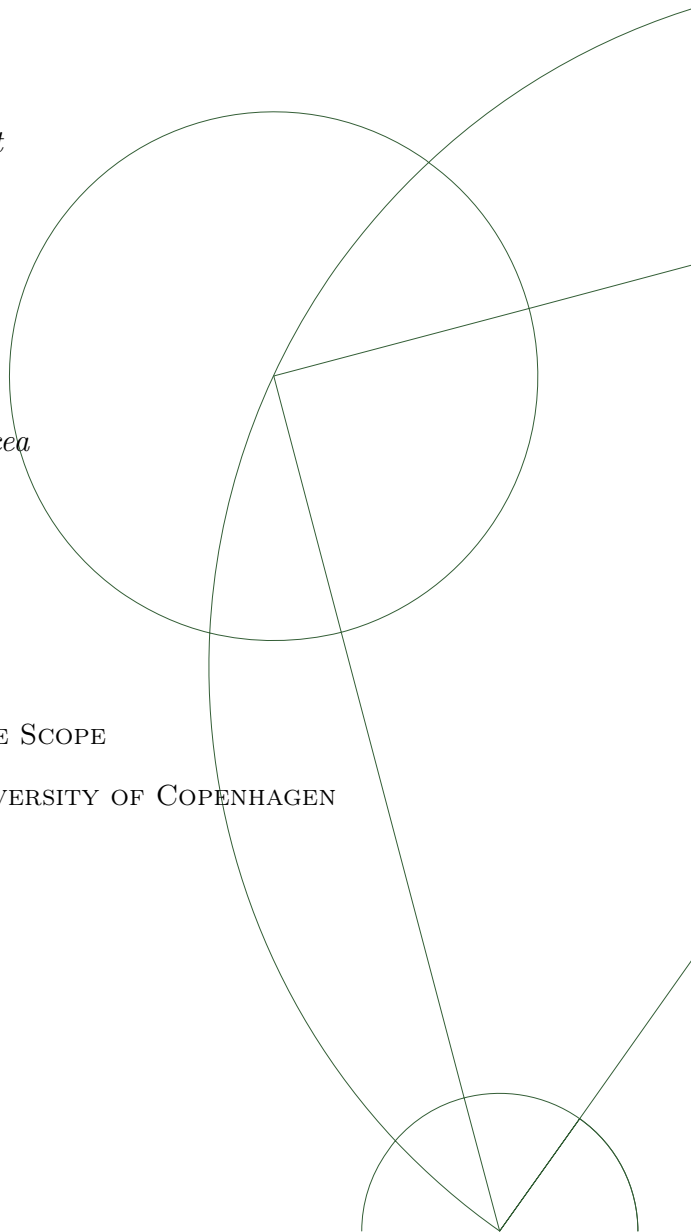
Sarah Maria Hyatt

zfj900@alumni.ku.dk

Supervisor:

Cosmin Eugen Oancea

cosmin.oancea@di.ku.dk



Project Outside Course Scope

Department of Computer Science, University of Copenhagen

April 21, 2020

Contents

1 Introduction

1. short introduction, in which you start by saying that of common agreement with your advisor have decided to narrow the scope of the original project to just targeting the exact k-nn problem solved with the kd tree, and that it was also agreed that this should not negatively influence your grade.

Then describe the main accomplishments of your work. (so perhaps you write the introduction last)

K-Nearest Neighbours (KNN) is a well-known algorithm highly used in areas such as computer vision and computer graphics. One way of applying it is in image recognition where it can be used to find similarities between two images.

(skriv om pixler der bliver til patches med PCA).

The naive approach of implementing KNN is a brute force method which compares each patch of image A with each patch of image B.

Many approaches (He & Sun, 2012) The argument for this approach is that by computing the exact KNN of the first query patch it is then possible to propagate .... and since the assumption is that the images are similar, it is likely that the former query patch will have nearby results as the current and the next.

More in section 2. Background.

Instead this project utilises the benefits of storing data of one image in a multidimensional binary search trees, also known as KD-trees, where k is the dimensionality of the search space. It has shown to be quite efficient in its storage requirements in addition to

1.1 Acknowledgement

6. I would appreciate if you write a short acknowledgment section in which you do not need to thank me, but to acknowledge my intelectual property of some of the key ideas presented in this work, including the tree construction by padding that allows regular nested parallelism, optimized tree traversal where the stack is one integer and involves all medians rather then only the current one, etc. This will not negatively affect your grade in any way.

2 Background

Computing KNN is widely applied due to its simplicity and excellent empirical performance, it has no training phase, it can handle binary as well as multiclass data, and while often applied for comparing two images, it faces the challenge of slow performance because both images are of image size.

When comparing two images; image A and image B, the goal of KNN is to find the patches of image B that are most similar to each query patch of image A according to a measured distance. The naive approach is using brute force, in which each patch of image B is compared with each patch of image A, if each image has m patches the complexity of the search will be $O(m^2)$. See Brute Force. Although brute force is a simple solution, there exists several alternative techniques for computing KNN. One commonly applied method is using

a generalisation of the binary tree; the k-d tree.

The k-d tree consists of a root, nodes and leaves; the root represents all patches, the nodes represent a partitioned segment of the patches, and the leaves collectively contain all patches spread out on 2^{h+1} leaves where h is the height of the tree excluding leaves.

Using a k-d tree has advances w.r.t search performance, thanks to the binary tree structure it yields a complexity of $O(m \ \lg \ m)$ (Friedman, Bentley, & Finkel, 1977).

Algorithms 1, 2 and 3 below represent a recursive implementation of the k-d tree construction and the tree traversal as high-level pseudo-code.

Cosmin: should I mention that the pseudo-code is based on the Python code and the authors?

Algorithm 1 is the `main` function in which dimensionality of images A and B are reduced using PCA[1], a k-d tree is created of image B, and all patches of image A are iterated through, of which a call to `TRAVERSE` yields the best neighbours for each patch in A.

Algorithm 1 Main

```
1: procedure MAIN(k)
2:     imA ← PCA(imageA)
3:     imB ← PCA(imageB)
4:     tree ← BUILDTREE(imB, imB.1, 0, 0, NONE)
5:     neighbours ← NONE
6: foreach query in imA do:
7:     neighbours ← TRAVERSE(tree, query, 0, neighbours, k)
8: return neighbours
```

Algorithm 2 represents the construction of the k-d tree. The tree construction is divided into two statements; working with the nodes (lines 4-9) or working with the leaves (lines 20-21). This condition is measured by computing the height of the tree and checking the current level against the height.

The patches are partitioned by finding the dimension with the widest spread and choosing

[1]Given a collection of points in two or higher dimensional space a principal component analysis (PCA) can be created by first choosing the line that minimises the average squared distance from a point to the line, resulting in an Eigenvector, second choosing the line perpendicular to the first, resulting in a new Eigenvector, and repeating the process will result in orthogonal basis vectors. These vectors are called principal components, and several related procedures principal component analysis (PCA).

the median value of that dimension. The nodes, therefore, contain a dimension and a median value. Line 5 uses the function GetSplitDimension to attain the dimension, lines 6-7 sort the indices and patches w.r.t the chosen dimension and lines 9-10 pick out the median index and value. Once done, the median and the dimension are ready to be stored, which essentially creates that particular node of the tree. Lines 13-19 compute the left and right indices for the next iteration and recursively call the BuildTree function again, in order to complete the creation of nodes in the tree.

Last, line 21 sorts the leaves w.r.t the indices that were sorted and partitioned throughout the recursive node creation.

Algorithm 2 Building the Tree

```
1:  procedure BuildTree(patches, indices, depth, index, tree)
2:      maxDepth ← ComputeMaxDepth(patches)
3:
4:      if depth < maxDepth-1 then
5:          dim ← GetSplitDimension(patches)
6:          indices ← SortByDim(indices, dim)
7:          points ← patches[indices]
8:
9:          medianIdx ← GetMedianIdx(indices)
10:         median ← GetMedian(points)
11:
12:         tree ← (median, dim)
13:         depth ← depth + 1
14:
15:         leftIdx ← AddToLeft(index)
16:         rightIdx ← AddToRight(index)
17:
18:         BuildTree(patches[: medianIdx], indices, depth, leftIdx, tree)
19:         BuildTree(patches[medianIdx :], indices, depth, rightIdx, tree)
20:     else
21:         leaves[index] ← SortLeavesByIndices(patches, indices)
```

The traversal is presented in Algorithm 3 below. It deals with three different stages; reaching a leaf, which is the first and second node to visit and a check to visit the second leaf. The first stage on lines 2-4, reaching a leaf, is straightforward. We reach a leaf; therefore, we perform brute force with the query patch and the patches of image B in that current leaf.

Algorithm 3 The Tree Traversal

```
1:  procedure TRAVERSE(tree, query, nodeIndex, neighbours, k)
2:      if IsLeaf(nodeIndex) then
3:          neighbours ← BRUTEFORCE(tree, query, nodeIndex, neighbours, k)
4:          return neighbours
5:
6:      dimens ← tree[nodeIndex].1
7:      median ← tree[nodeIndex].0
8:      queryV ← query[dimens]
9:
10:     if queryV <= median then
11:         first ← GOLEFT(nodeIndex)
12:         second ← GORIGHT(nodeIndex)
13:     else
14:         first ← GORIGHT(nodeIndex)
15:         second ← GOLEFT(nodeIndex)
16:
17:     neighbours ← TRAVERSE(tree, query, first, neighbours, k)
18:     worstNeighbour ← LAST OF neighbours
19:
20:     if (median - queryV) < worstNeighbour then
21:         neighbours ← TRAVERSE(tree, query, second, neighbours, k)
22:  return neighbours
```

The third stage, lines 18-21, determine whether we should visit the second node by checking if the difference between the median and the query is less than the worst nearest neighbour. If this is the case; the second node will be visited in a recursive call of Traverse on line 21. The check is an estimate to assure that it makes sense to continue the traversal, because

if the worst nearest neighbours is a better result in XXXX.

Argument the logic behind this.

The second stage on lines 6-17 determines which following nodes are the first and second, respectively. The first is always the node that is on the same side of the current node's median as the query. We might think of this as going left first on lines 10-12 and right first on lines 13-15. The dimension and median of the current node is extracted from the tree, lines 6-7, and the query value based on the current dimension of that node, line 8. Line 17 executes a recursive call of Traverse in which the first node is the next node to be visited.

- approximate, PCA
- implement in Futhark, why
-

2.1 Related Work

2.1.1 PatchMatch

PatchMatch uses a randomised algorithm for quickly finding approximate nearest neighbour fields (ANNF) between image patches.

While previous solutions use KD-Trees with dimensionality reduction, PatchMatch instead searches through a 2D space of possible patch offset. The initial step is choosing random patches followed by 4-5 iterations containing two processes: (1) propagation applying a statistic that can be used to examine the relation between two signals or data sets, known as coherence, (2) random search in the concentric neighbourhood to seek better matches by multiple scales of random offsets. The argument is that one random choice when assigning a patch is non-optimal, however applying a large field of random assignments is likely a good choice because it populates a larger domain.

The gains are particularly performance enabling interactive image editing and sparse memory usage, resulting in runtime O(mM log M) and memory usage O(M), where m is the number of pixels and M the number of patches. The downside is that PatchMatch depends on similar images because it searches in nearby regions in few iterations, which in turn also affects its accuracy.

2.1.2 Coherency Sensitive Hashing

Coherency Sensitive Hashing (CSH) is inspired by the techniques of Locality Sensitive Hashing (LSH) and PatchMatch. CSH uses a likewise hashing scheme of LSH and is built on two overall stages, indexing and searching.

The indexing stage applies 2D Walsh-Hadamard kernels in which the projections of each patch in image A and B are initially computed onto the WH kernels. Subsequently the hash tables are created by the following. First, it takes a random line defined by a patch and divides it into bins of a constant width while shifting the division by a random offset. Second, the patch is projected onto the most significant 2D Walsh-Hadamard kernels. Last, a hash value is assigned, being the index of the bin it falls into.

Applying hash tables has the benefit that similar patches are likely to be hashed into the same entry.

The searching stage starts by initialising an arbitrary candidate map of ANNF. This is followed by iterations through each patch in image A where: (1) the nearest neighbour candidates of image B are found using the current ANNF and the current hash table, (2) the current ANNF mapping is updated with approximate distances.

Selecting candidates follows the appearance-based techniques of LSH and as well as the coherence-based techniques of PatchMatch. Choosing among the candidates is done in an approximate manner where the WH kernels rejection scheme for pattern matching is applied beneficially.

CSH has ‘46%‘ better performance than PatchMatch and a higher level of accuracy with error rates that are ‘22%‘ lower in comparison

3 Implementation

3.1 The Brute Force Version

```
entry nnk [m] [n] (imA : [m][n]real)
                  (imB : [m][n]real) : [m][k](int,real) =
    map (\a_patch ->
        if a_patch[0] == real_inf
        then replicate k (-2i32, real_inf)
        else
        let  nn = replicate k (-1i32, real_inf)
        in loop nn for q < m do
            let b_patch = imB[q]
            let dist = euclidean a_patch b_patch
            let b_idx = q in
            let (_, _, nn') =
                loop (dist, b_idx, nn) for i < k do
                    let cur_nn = nn[i].1  in
                    if dist <= cur_nn then
                        let tmp_ind = nn[i].0
                        let nn[i] = (b_idx, dist)
                        let b_idx = tmp_ind
                        let dist  = cur_nn
                        in  (dist, b_idx, nn)
                    else    (dist, b_idx, nn)
            in  nn'
    ) imA
```

Listing 1: Futhark implementation of the Brute Force.

3.2 KD-Tree Construction

```

```

Listing 2: Futhark implementation of the tree creation.

3.3 Tree Traversal

```
entry firstTraverse [d][q] (height:   i32)  (median_dims: [q]i32)
                           (query: [d]f32)  (median_vals: [q]f32) =

    let new_leaf = loop node_index = 0
        while !(isLeaf height node_index) do
          if query[median_dims[node_index]] <= median_vals[node_index]
          then (node_index+1)*2-1
          else (node_index+1)*2

    in new_leaf
```

Listing 3: Futhark implementation of the first tree traversal.

```
entry traverse [d][n][l] (height:             i32)  (median_dims:     [n]i32)
                         (median_vals:     [n]f32)  (wknn:               f32)
                         (query:           [d]f32)  (stack:              i32)
                         (last_leaf:          i32)  (lower_bounds: [l][d]f32)
                         (upper_bounds: [l][d]f32)  : (i32, i32) =

  let setVisited (stk: i32) (c: i32) : i32 =
      stk | (1 << c)
  let resetVisit (stk: i32) (c: i32) : i32 =
      stk & !(1 << c)
  let isVisited (stk: i32) (c: i32) : bool =
      (stk & (1 << c)) > 0i32

  let (parent_rec, stack, count, rec_node) =
      loop (node_index, stack, count, rec_node) =
           (last_leaf, stack, height, -1)
            while (node_index != 0) && (rec_node < 0) do
                let parent = getParent node_index
                let second = node_index + addToSecond node_index in

                if isVisited stack count
                then (parent, stack, count-1, -1)
                else
                  let ack =
                    loop ack = 0.0f32
                      for i < d do
                          let cur_q = query[i]
                          let lower = lower_bounds[second,i]
                          let upper = upper_bounds[second,i] in

                          if cur_q <= lower then
                              let res = (cur_q-lower)*(cur_q-lower)
                              in (ack + res)
                          else if cur_q >= upper then
                              let res = (cur_q-upper)*(cur_q-upper)
                              in (ack + res)
                          else (ack + 0.0)

                  let to_visit = (f32.sqrt ack) < wknn in
                  let to_visit = f32.abs(median_vals[parent] - query[median_dims[parent]]) < wknn in
                  if !to_visit
                  then (parent, stack, count-1, -1)
                  else
                    let second = node_index + addToSecond node_index
                    let stack  = setVisited stack count in
                    (parent, stack, count, second)

  let (new_leaf, stack, _) =
      if parent_rec == 0 && rec_node == -1
```

3.4 The Full Implementation

At each step you reason about tradeoffs/alternative design choices and justify why you did it in the way you did it (advantages/shortcommings, and why your approach is reasonable). Of course mostly related to performance.

Whenever possible, support your reasoning with (as in point to) experimental evaluation results (next).

4 Experimental Evaluation

Validate the claims that you have done in 3. and introduction.

Here you think what you need to demonstrate and how to best demonstrate it (what kind of graphs, tables, etc are needed). You can do the experimental evaluation (i.e., running/benchmarking the futhark programs and gathering performance data) at this point, once you put some though into what this should be.

- The full brute force - Implementation with slow brute force - Implementation with fast brute force - Implementation with fast brute force and merge sort - Implementation with fast brute force and radix sort - Implementation with fast brute force, radix sort and partition - Implementation with fast brute force, radix sort, partition and fixed traversal - Implementation with fast brute force, radix sort and leaf sort - Implementation with fast brute force, radix sort, leaf sort and fixed traversal

Get a smallish starting point like m=100000 and then increase 400000 , 800000, 1.2mil, 1.6 mil, 2mil, 4mil, ... until it still takes reasonable time to compute.

With high dimensionality, for large k you will wait for ages because it's going to visit all leaves, so maybe 1, 3, 5, 7, 17 (the last one will take a while)

Maybe d=1, 4, 6, 8, 11, 16

100000

400000

800000

1200000

1600000

2000000

4000000

k 1 3 5 7 17

d 1 4 6 8 11 16

5 Conclusion

References

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*.

He, K., & Sun, J. (2012). Computing nearest-neighbor fields via propagation-assisted kd-trees. *Microsoft Research Asia*.