FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Accelerating Nearest-Neighbours via KD-Trees on GPUs

Author:

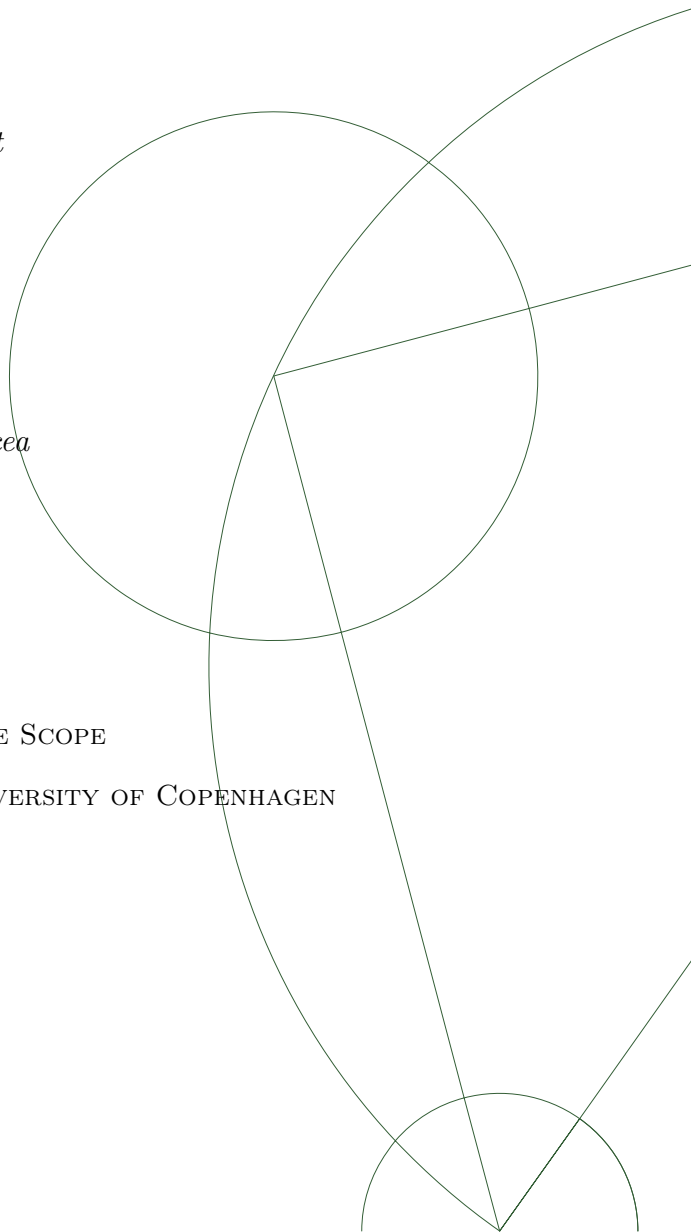
Sarah Maria Hyatt

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 15, 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.

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 K-Nearest Neighbours (KNN) is widely used due to its simplicity and excellent empirical performance. It has no training phase and it can handle binary as well as multiclass data. When comparing two images the challenge of KNN is finding the patches of say image B that are most similar to the query patch of say image A according to some measured distance. While there are many ways to compute KNN, one commonly applied method is using the *k-d* tree. The *k-d* tree is a binary tree and in this case we define each node to represent a sub-image of the patches in the image representation and a partitioning of that sub-image, meaning the root will represent the whole image.

Once the tree is created, all leaves will together contain the full partitioned image and each node will represent $\frac{m}{2^{level}}$ of the image patches, where $level$ is the current level of the node in the tree in which $level \leq height + 1$, $height$ is the height of the tree excluding leaves and m is the total number of patches in the image.

The benefit of using a *k-d* tree is the advances w.r.t. search performance. Thanks to the binary tree structure a KNN search for patches will have a complexity of $O(n \ \lg \ n)$. The naive approach of computing KNN is the brute force method, comparing each patch of image

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
```

B with each patch of image A, resulting in a complexity of $O(n^2)$ (Friedman, Bentley, & Finkel, 1977).

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.

While this report focuses on computing the exact KNN, it will still be an approximate KNN, due to the use of PCA to reduce dimensionality of the pixels in each image.

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

Algorithm 2 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
```

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.

Algorithm 3 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(patches, 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)
```

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

3. section(s) in which you describe your work: each step – brute force version – k-d tree construction – tree traversal – how you plug everything together

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).

3.2 KD-Tree Construction

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).

3.3 Tree Traversal

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).

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

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*.