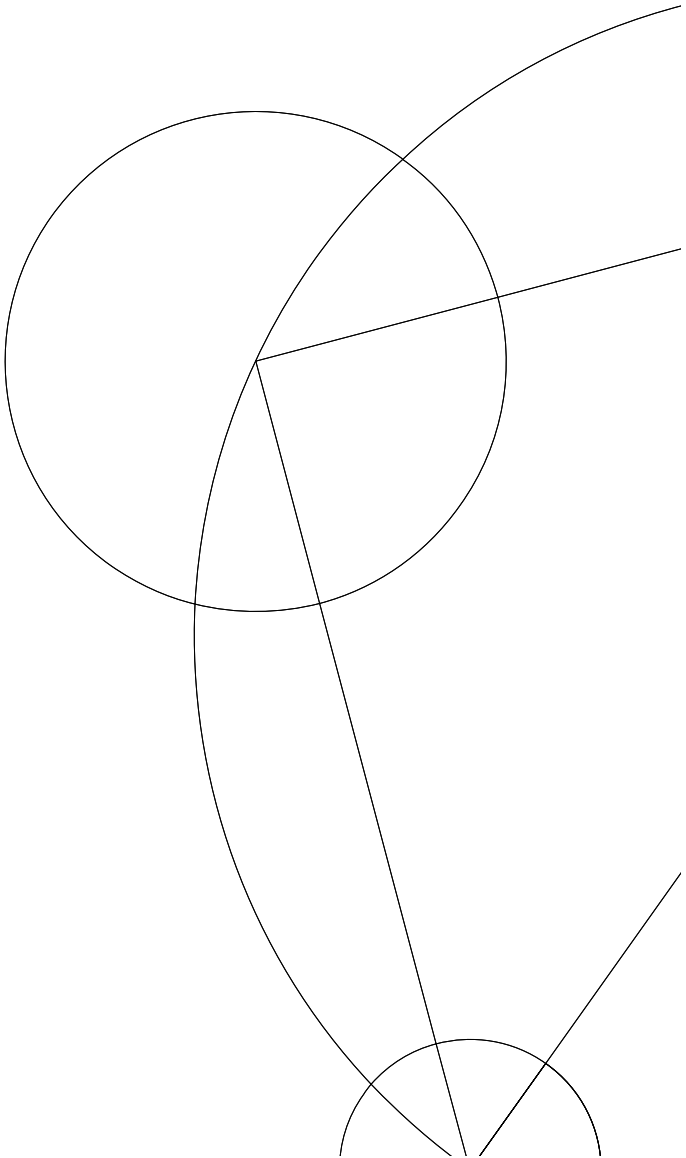
UNIVERSITY OF COPENHAGEN

Data-Parallel Coherency Sensitive Hashing for Approximate Nearest Neighbour Fields

Department of Computer Science
University of Copenhagen

May 31, 2021

Kristian B. Andreasen <ncx523@alumni.ku.dk>
Supervisor: Cosmin Eugen Oancea



Contents

Abstract

Finding similar patches in two images is an often used technique in computer vision. Methods that achieve this purpose, known as Nearest Neighbour Fields, are slow and expensive and modern demands have only served to increase the resolution of the images these techniques are used on. To alleviate this issue approximate solutions are computed, which trade exactness for speed. Recent research has shown that highly parallel implementations that use GPUs for computing can be used to achieve large speedups with no impact on accuracy. In this thesis I propose a data-parallel implementation of Coherency Sensitive Hashing(CSH). CSH, when using CPUs, has been observed to achieve good performance by exploiting the coherency of images and similarity between patches. The data-parallel implementation shows a significant increase in speed at no cost to accuracy and is quick enough to be run in real time on high quality images. The performance of data-parallel CSH has been verified using an image set of 133 image pairs.

1 Introduction

Similarities between pictures is an often visited concept in computer vision. Finding similarities has been used for a variety of purposes,such as **retargeting** [2], which allows for dynamically sizing images to new aspect ratios while keeping the image looking good. Another purpose is **image completion** [11], which allows the user to remove objects and the area removed is then painted in to look natural.

One way of describing such similarities is through Nearest Neighbour Fields(NNFs). Images can be divided into overlapping fields with each field or patch being comprised of a number of pixels. For a patch a in image A an NNF is a patch b in an image B such that no other patch in image B has a lower distance to patch a. The brute force approach to computing the set of NNFs requires finding the distance from all fields in image A to all fields in image B. This is clearly a challenge that suffers from the curse of dimensionality, which is only becoming worse as the resolution of our images continue to increase. To prevent NNFs from becoming a large computational bottleneck for computer vision an approximate solution is instead used.

Approximate Nearest Neighbour Fields(ANNFs) finds a decent match for each patch rather than the exact match. This reduces the computational load at the cost of accuracy. There are a variety of methods used for this purpose, which makes use of properties relating to images or data-structures to achieve good accuracy without a corresponding large run-time.

KD trees [1] is a data structure based algorithm that uses a k-dimensional tree to guide the search for the nearest neighbour field. It is possible to use KD trees to obtain an exact NNF [5], but the run-time cost of this is typically far too high. Various methods are instead used to prune which branches or leaves are visited, such that only a limited number of candidates are generated instead of the worst case scenario, which is every single field. A KD tree can be viewed as a $\mathbf{R}^d$ ordering of the fields.

This is quite similar to the idea behind Locality Sensitive Hashing [4], which instead use a $\mathbf{R}^d$ hash function on the fields and generates candidates based on collisions coming from those hashes. The number of candidates generated from this is typically low and multiple such hash functions are instead used to generate a rich field of candidates. Both of these approaches are examples of locality sensitive based algorithms, which exploit the content of the fields to find fields with similar content.

These performance of these approaches were beaten by that of PatchMatch [2], which uses coherency to propagate good ANNFs to nearby patches. PatchMatch uses the fact that images are coherent, which means that neigbouring patches are likely to have neighbouring ANNFs. In other words, if a pair of patch/match covers the left part of an apple we would expect the right neighbour to the patch to have a match that covers the middle of an apple(or slightly less

left part of an apple), which would correspond to the right neighbour of the match. PatchMatch makes use of random starting conditions and so several iterations of propagation are required before a good ANNF has been computed.

Locality Sensitive Hashing and PatchMatch inspired the creation of Coherency Sensitive Hashing [12], which makes use of both locality sensitivity and coherency. Similar to LSH it makes use of hashing to obtain collisions and similar to PatchMatch good matches are propagated to neighbouring patches, unlike both of them candidates are also generated from a combination of the two. Like PatchMatch multiple iterations are required for a good result, however the use of both coherency and locality means the algorithm performs better than those it were inspired from.

This in turn was improved on by Propagation Assisted KD trees [7], which instead of solely relying on KD trees, computes the NNF for a row of the image and propagates this to the next row. The next row then uses this information to lookup which leaves to use as candidates. This combination allows the creation of a rich field of candidates at a low run-time cost, while pruning unlikely candidates, which allows for the cost of ranking the candidates to remain low.

The methods mentioned above typically have run-times far lower than that of exact methods, however even then the computational load remains high. Modern demands means that size of images grow and they are now comprised of millions of pixels. As such the computation of ANNFs continue to exist as a computationally heavy problem. One solution to this is to use the old solution of throwing more computational power at the problem. Today that solution is achieved by implementing highly-parallel algorithms that can make use of parallel processing power, such as that delivered by graphical processing units (GPUs). Propagation Assisted KD trees has been implemented using highly parallel approach [13] and allows for the use of ANNFs in real time.

Propagation Assisted KD trees have been implemented in the language Futhark. Futhark has previously been used to parallelize other applications, where runtime is of paramount importance, such as using satellite data to detect changes in the landscape [6].

Earlier methods, such as CSH, have been implemented using a CPU-based approach and not a highly parallel approach. It is relevant to investigate the increase in performance such methods may achieve from changing the approach. Implementing algorithms such that it can be run using highly parallel devices are not without issues and the goal of this thesis is to implement CSH in a parallel fashion, such that these challenges are handled properly. CSH is an interesting problem for this as the original implementation contains several sequential parts that are not trivial to handle when implementing a parallel version.

I have implemented and investigated the performance of a data-parallel version of CSH in Futhark and as part of this process I have identified and solved two critical challenges. The method used for reducing dimensionality has a linear recurrence that is not trivially parallelized with a scan. This has been solved using linear function composition and a scan. Ranking candidates is a sequential operation in the original CSH. A naive parallelization of this results in poor performance due to a high degree of thread divergence. I have solved this through a brute force approach that utilizes intra-group parallelism.

In this thesis I will present a highly parallel implementation of CSH[1] using the language Futhark. Section 2 presents related algorithms that compute ANNFs using a different approach than CSH. Section 3 presents Futhark, which is the language I have implemented data-parallel CSH in. Section 4 presents the original CSH. Section 5 presents the implementation details of and reasoning behind data-parallel CSH. Section 6 presents various trade-offs that may improve the performance of CSH. Section 7 presents the experimental results of data-parallel CSH in comparison to the original CSH, propagation-assisted KD trees and the variations presented in section 6. Section 8 presents various issues and improvements that future work may address. Section 9 is the conclusion.

[1]Github Repository: github.com/mavion/annf

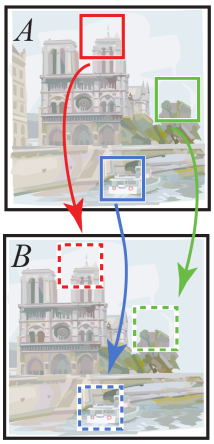
2 Related Work

2.1 Locality Sensitive Hashing

Locality Sensitive Hashing(LSH) is an approach that uses information about the local space to generate candidates. This is done through a hashing scheme followed by exploring the candidate set generated.

The algorithm is comprised of two stages: Indexing and searching. In the indexing stage LSH functions are used to create hash tables in which similar points index into the same hash bins with high probability. A number of such hash functions are concatenated together to create a code for the purpose of increasing the difference in probability of similar points and dissimilar points hashing to the same bin. These codes are used to create a single hash table, which is used in the search stage. In the search stage points are hashed into the table in order to find nearest neighbour candidates. For the purpose of increasing the probability of finding good candidates multiple random codes are generated, which are searched sequentially in the search stage. The scheme used by Datar et al. [4] uses the family of LSH functions of the shape:

$$h_{a,b}(v) = \frac{a \cdot v + b}{r} \tag{1}$$



Each field in the source image A has a corresponding nearest neighbour field in the target image B. Only a very small minority of fields are shown.

where r is a predefined integer defining the width of a bucket, b is a random value in the interval *[0,r)* and a is a d-dimensional vector with entries chosen randomly from a Gaussian distribution. v is the vector representing the patch. In short the vector v is projected onto a line a, offset by b and then divided by r with the final result being the bin it hashes into. The maximum number of bins depends on the magnitude of v and scales inversely with r.

Hashing patches into bins is a way of compacting information into a more usable format. This way of quantizing information has some issues, such as binning similar patches with similar hashes into separate bins solely because the bin edge happened to be between the two. To address this issue a random offset b is used, which increases the probability that similar patches hash into the same bucket. In short, b is used to neutralize the quantization effects of fixed binning.

A number of such hash functions with independently chosen random variables are used to create a hash code:

$$hc(v) = h_{a_1,b_1}(v) +\!+ \ldots +\!+ h_{a_m,b_m}(v) \tag{2}$$

which is concatenation of m different hash functions. This is then repeated to create a number of hash tables where each patch has a hash code for each hash table. To summarize: the LSH algorithm is called with two images. All patches in both images are hashed using a random family of hash functions to generate a hash for each patch. Collisions between patches in separate images are considered candidates for nearest neighbour field. Multiple such hashes are generated, so as to generate a richer field of candidates. To keep it tractable the number of candidates generated per hash is upper bounded.

The method for LSH to achieve it's accuracy is through the assumption that nearest neighbour fields are similar to each other and that similar fields hash similarly. Improving the accuracy of LSH is done by generating more candidates either through hashing more times or by increasing the number of collisions per hash, which can be done by increasing bin width or using more representatives. This comes with a natural increase in run-time, which can be mitigated by using a GPU based solution, such as that presented by Pan and Manocha [14].

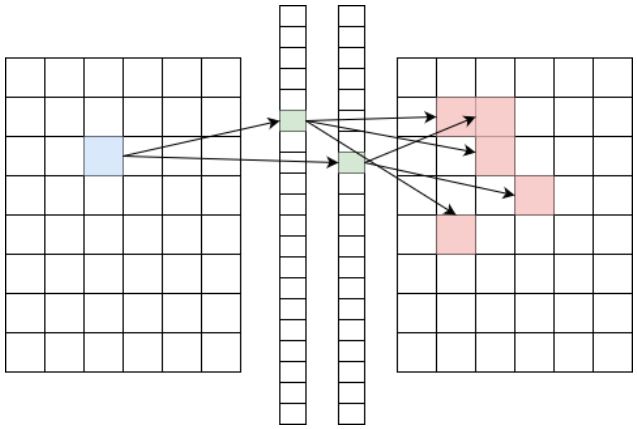


Figure 1: The hashing and collision scheme used by LSH. The patch in the left image has two different hash values corresponding to the two hash tables in the middle, these hash into different bins and the collisions from these bins generate candidates in the right image.

2.2 PatchMatch

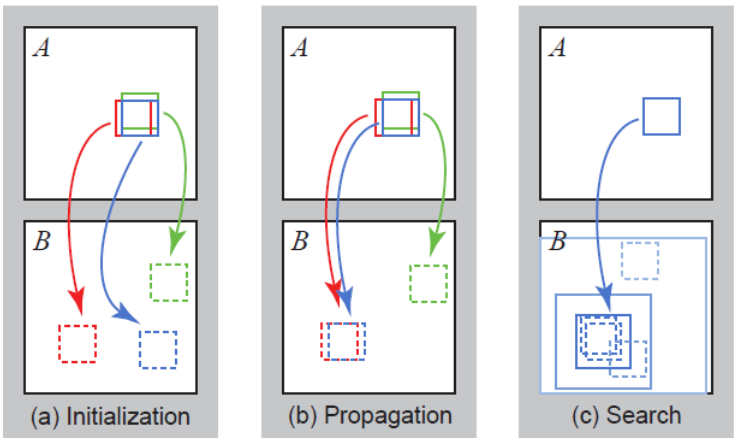


Figure 2: The 3 phases used by the PatchMatch algorithm. (a) initially matches are randomly distributed. (b) red propagates it's good match to the neighbouring blue. (c) A random search is performed for blue. b and c is repeated several times until a good result is found or a set number of iterations have been performed.

PatchMatch [2] is an approach that makes use of the coherency of images. This is done by generating random candidates and propagating good candidates to neighbouring fields.

PatchMatch is comprised of two stages: initialization and iteration. Initialization consists of setting the initial matches for each patch, which can be done either by using random starts or by using prior information. One approach suggested by Barnes et al. [2] is generating prior information by running the algorithm on a downscaled version of the images.

Iterating is done in two steps(figure 2): Propagation and random search. Propagation is done using coherency, which means that patches within an image and between two images are correlated. In other words if an object appears in both images then the patches that correspond to that object are related. If a patch finds a match on that object then the neighbours of that

patch likely also have a good match on that object, which is likely next to the original match. In other words if a patch/match are close then the patches to the right of those two are likely close as well. As such good matches can be propagated to neighbours, which can then propagate them further.

Propagation to neighbours however run into the issue of getting trapped in local minima. To mitigate this issue random candidates are generated. The approach used by Barnes et al. [2] generates random candidates based on the currently found match. Candidates are more likely to be generated the closer they are to the current match.

2.3 Propagation-Assisted K-D Trees

Propagation-Assisted K-D Trees is a variant of K-D trees that use the coherency of the image to propagate good matches to neighbouring patches.

K-D trees are a data structure for a point set $P \subset \mathbb{R}^d$, where the root of the tree corresponds to all points in P. Each level of the tree divides the tree along one of the dimensions, such that the set of points is evenly divided along the two branches. The branches are then further divided evenly until such a stopping criteria is reached, which typically corresponds to each leaf containing less than a set number of points. In other words K-D trees are a K dimensional analogue to binary trees.

A K-D tree can be used to find the exact nearest neighbour for a point q. This is done by traversing the tree from top to bottom to find the leaf that q would fit into. The points in that leaf are then stored as the k nearest neighbours. The tree is then fully traversed from bottom to top while only checking those branches that are necessary. It is only necessary to check a branch if the distance from q to the box surrounding that branch is less than the distance from q to any of the k nearest neighbours. This set of k nearest neighbours may be updated whenever a branch is checked.

For a low number of dimensions this substantially improves the running time compared to brute forcing the nearest neighbour fields, but for a large number of dimensions branches will often be checked and so there's little to no benefit for the running time. This can be mitigated by reducing the dimensionality of P, which can be done through methods such as Principal Component Analysis or similar.

Propagation-Assisted K-D trees is a novel approach that makes use of coherency to remove the need for traversing the tree when finding the nearest neighbour fields. This approach was first shown by He and Sun [7] and later improved by Oancea et al [13].

The approach is comprised of 5 steps:

1. Reducing the dimensionality. Less dimensions means that traversing the KD tree is less costly and it reduces the cost of computing the distance from a point to another point. This comes at the cost of accuracy.
2. Building the KD tree.
3. Find the Exact KNN for the first row. The method that has been described already is used on the first row and only the first row.
4. Propagate using the results from the first row. The results from previous row is propagated as seen in figure 3.
5. Pick from KNN using full dimensionality. The KNN found in step 3 and 4 were found using reduced dimensionality and so the current ranking may not be accurate.

To summarize: Propagation-assisted KD trees use KD trees to initialize a row with good matches. These matches are then propagated to neighbouring patches, which makes use of the coherency of the image. The leaves of the KD tree these candidates belong are used to enrich set of candidates, which is an example of locality sensitivity.

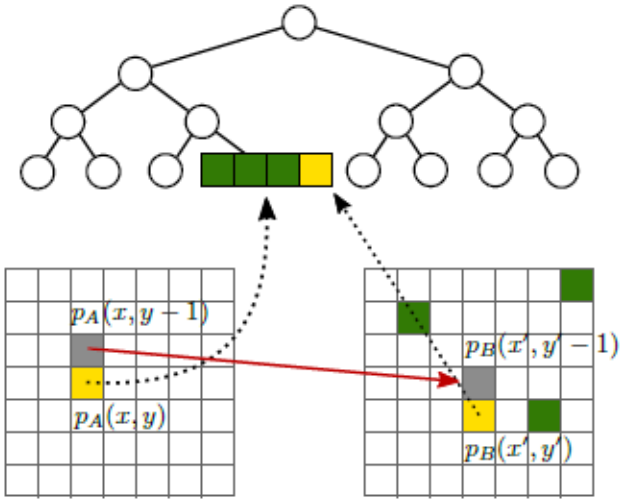


Figure 3: The propagation performed by KD trees. The two neighbours of patch/match pair are a patch/-candidate pair. For each such candidate the leaf in the KD-tree that contains the candidate are also candidates for said patch.

3 Futhark: The programming language

The implementation of the algorithms are done in Futhark [8] [9]. Futhark is a programming language designed to generate efficient parallel code. It is statically typed, data-parallel, and purely functional. The Futhark compiler can generate both multi-threaded CPU code and optimized parallel GPU code.

Futhark provides a number of functions for writing parallel programs. Readers that are familiar with other parallel languages may recognize these. The most relevant are shown here.

Name	Function type
iota	(n: i64) → [n]i64
map	(a → x) → [n]a → [n]x
reduce	(a → a → a) → a → [n]a → a
scan	(a → a → a) → a → [n]a → [n]a
scatter	*[n]d → [m]i64 → [m]d → [n]d

Table 1: Parallel functions for writing parallel code in Futhark. → indicates that the left hand side is used to produce the right hand side. A sequence of arrows can be read as multiple inputs to the same function. Arrows inside parentheses(like in map) indicates that the input is a function. A unique input is denoted by * and its semantics is that it is consumed by the corresponding operation, such as scatter, i.e., it is illegal to access it on any execution path following the consuming operation.

3.1 Functions

3.1.1 Iota

Iota has the function type: (n: i64) → [n]i64, where `[n]i64` denotes the type of an array of `n` 64-bit integral elements. It has one input and one output. The input is a 64-bit integer while the output is a list of integers of the same length as the value of the input. The output is a list of the integers from 0 to n-1 ordered in ascending order.

```
let a = iota 5
```

This snippet in Futhark is the equivalent to the following snippet in C-like code:

```
int *a;
for (int i=0; i<5;i++){
    a[i] = i;
}
```

Implicit indexing is a common technique, but the index may be needed. In sequential programming the index is often produced through an iterator as part of a loop or similar, but in parallel programming this is less than ideal. A way is needed to produce the index cheaply, which iota is suited for.

3.1.2 Map

Map has the function type: (a → x) → [n]a → [n]x. It takes two inputs; a function, and an array, while it outputs an array of the same size as the input array. Map is used to apply a function to each element of an array.

```
let a = map (+2) (iota 5)
```

This snippet in Futhark is the equivalent to:

```
int *a;
for (int i=0; i<5;i++){
    a[i] = i+2;
}
```

3.1.3 Reduce

Reduce has the function type (a → a → a) → a → [n]a → a, where a is an anonymous type that does not change between the elements. There are 3 inputs; an associative operator $\odot$, the neutral element of the operator and an array. The semantics are as follows:

$$\text{reduce} \odot \text{ne} [a_1, ..., a_n] = a_1 \odot a_2 ... \odot a_n$$

The following is an example of summation implemented in Futhark, which can be done easily using reduce.

```
let sum as = reduce (+) 0 as
let a = sum (iota 5)
```

3.1.4 Scan

Scan has the function type (a → a → a) → a → [n]a → a, where a is an anonymous type. There are 3 inputs; an associative operator $\odot$, the neutral element of the operator and an array. Scan outputs an array of the same size as the input array, where each element has the value that is the equivalent of having performed a reduce up to that element. In other words:

$$\text{scan} \odot \text{ne} [a_1, ..., a_n] = [\text{reduce} \odot \text{ne} [a_1], \text{reduce} \odot \text{ne} [a_1, a_2], ..., \text{reduce} \odot \text{ne} [a_1, ..., a_n]]$$

The following declarations of b have the same values:

```
let b = scan (+) 0 (iota 5)
let sum bs = reduce (+) 0 bs
let b = [sum (iota 1), sum (iota 2), sum (iota 3),
         sum (iota 4), sum (iota 5)]
let b = [0,1,3,6,10]
```

In C-like code scan can be done as follows:

```
int *a;
a[0] = 0;
for (int i=1; i<5;i++){
    a[i] = i;
}
for (int i=1; i<5;i++){
    a[i] = A[i-1]+a[i];
}
```

3.1.5 Scatter

Scatter takes a destination array, an array of array indexes, and an array of values. The output is the first array with values updated in the positions pointed too by the second array, while the values are supplied by the third.

```
let a = iota 5 -- [0,1,2,3,4]
let inds = [3,2,1,0]
let a = scatter a inds (iota 4) -- [3,2,1,0,4]
```

Scatter is used to perform a large amount of in place memory updates in parallel. The behaviour is undefined if the same index is updated with multiple values unless those values are identical. Scatter can be implemented sequentially as follows.

```
void scatter(int* dest, int* inds, int* as, size){
    for(int i; i< size, i++){
        dest[inds[i]] = as[i];
    }
}
```

3.2 Cost analysis

For sequential programming one way of denoting cost is the number of operations needed as this typically correlates linearly with run-time, which is what we're typically trying to optimize for. However this does not accurately the behaviour of parallel programming.

Parallel programming utilizes a number of threads to run operations in parallel. The same number of operations is executed in less time if more of them can be run in parallel. To describe this I will use two concepts: Work and span. Work is the number of operations performed, while span is the number of operations performed by a single thread if an infinite number of threads were available.

The costs for the functions described in this section can be seen in table 2. The costs for the sequential equivalents have not been included, but in all cases have the same asymptotic work.

Name	Work	Span
iota n	O(n)	O(1)
map f (as: [n]t)	O(n*W(f))	O(W(f))
reduce f ne (as: [n]t)	O(n*W(f))	O(log(n)*W(f))
scan f ne (as: [n]t)	O(n*W(f))	O(log(n)*W(f))
scatter d (i: [n]i64) v	O(n)	O(1)

Table 2: Costs for parallel functions as seen in Futhark. Types have been included where relevant for the cost.

3.3 Other considerations

Futhark has a highly efficient compiler, which handles a lot of the low implementations details, which allows the programmer to focus on the larger issues with the implementation. When compiled to GPU code the Futhark programmer does not need to take care of threads, their ids or similar, but they still exist and can have an impact on the performance of the code.

An important detail is the existence of warps. Threads are grouped together in a warp and on GPU, threads in a warp execute in SIMD fashion.

3.3.1 Coalesced access to memory

Reading data from global memory is a slow process. As such reducing the number of global memory accesses is important for programs where memory is the bottleneck. Warps are implemented such that threads in the same warp requesting access to global memory can be done using a lower number of accesses if the requested memory is located consecutively. Access is coalesced when the threads in a warp access consecutive memory locations.

3.3.2 Thread Divergence

Threads in the same warp work in lockstep, which means that they execute the same operation simultaneously, but with different data. This is what allows the previously mentioned coalesced memory accesses, but it has a negative impact on performance for branching statements. Suppose you have a function similar to the following:

```
1 map (\x -> if x % 16 then x
2             else reduce (+) 0 (iota 100)) (iota 1000)
```

If the content of the map is executed sequentially then 1 out of 16 threads have much more work to accomplish. With a warp of size 16 this means that all threads will either need to wait for or perform the reduce. The example is of course a bit sought as a good compiler should be able to optimize that away.

4 Coherency Sensitive Hashing

Coherency Sensitive hashing is an approach that makes use of Locality Sensitive Hashing, which was first introduced by Indyk and Motwani [10]. LSH functions has the feature that points that are similar have a higher probability of hashing to the same bucket than dissimilar points have. In this section I will outline the specific version used for CSH.

4.1 Walsh-Hadamard kernels

CSH generates a number of hash tables in the indexing stage, making use of the same general framework as LSH. However Korman and Avidan [12] found that the use of Walsh-Hadamard kernels(WH kernels) as the family of functions used for hashing was crucial for performance as they allow for the efficient computation of hash codes and lower bounding of the L_2 distance. The L_2 distance is used in the search stage for comparing similarity of patches.

WH kernels are a subset of Gray code kernels. Gray code kernels have two properties that are relevant for ANNF: they represent a way of reducing dimensionality and they can be computed effficiently [3]. A grey code kernel can be computed in only two additions per pixel given the existence of a related kernel. Related kernels are kernels that differ only in a single step when looking at the recurrence they correspond to.

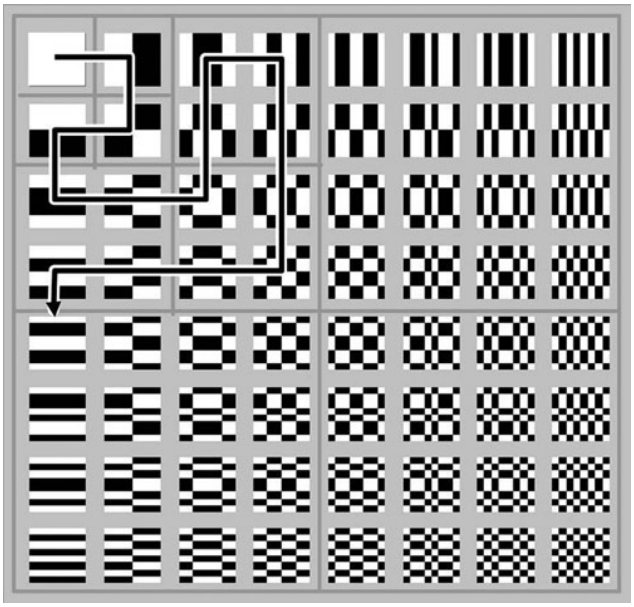


Figure 4: The 8x8 set of Walsh Hadamard kernels. A valid gray code sequence is shown. This is not the only valid sequence.

1D WH kernels are generated through the following recurrence:

$$WH_0 = 1 \tag{3}$$

$$WH_k = \{w_{k-1} +\!\!+ \alpha_k * w_{k-1}\} \text{ s.t. } w_{k-1} \in WH_{k-1}, \alpha_k \in \{+1, -1\} \tag{4}$$

A graphical example of this is shown in figure 5. The operator $+\!\!+$ indicates concatenation. For each kernel w_{k-1} in a set WH_{k-1}, two kernels of the next level w_k are produced by concatenating the kernel with itself or the additive inverse of itself. This recurrence creates a set of size 2^k 1D kernels. 2D kernels are generated through the following formula:

$$WH_k^2 = \{w_1 \otimes w_2\} \text{ s.t. } w_1, w_2 \in WH_k^1 \tag{5}$$

WH_k^D is the set of Walsh Hadamard kernels of dimensionality D and size k, where k is the size along all dimensions. WH_3^2 is the full set of WH kernels of size 8x8. WH_3^2 is computed by taking all kernels from WH_3^1, pairing each kernel up with all kernels from WH_3^1 and applying the outer product to all such pairs. WH3^2 is shown in figure 4, which forms the complete basis for 8x8 patches. Only a part of the set is used for CSH.

The standard approach is to use a function that projects the patch onto a 1-dimensional subset(a line). Ideally the dispersion of the projections of the image onto the line is such that similar patches end up close to each other, while patches with large differences are easily discriminated between. One way of accomplishing this is by taking the eigenvalues of the covariance matrix of the entire set of image patches. For natural images the eigenvalues turn out to form a sinusoidal basis, ordered in increasing frequency. WH kernels, when ordered by increasing frequency(fig 5), produce a similar set of projection lines.

These projections have been shown to be effective for pattern matching in images by [3]. The full set of WH kernels form a complete and ortogonal basis, which means that it is possible to compute the squared L_2 distance by summing the difference between the projections of two

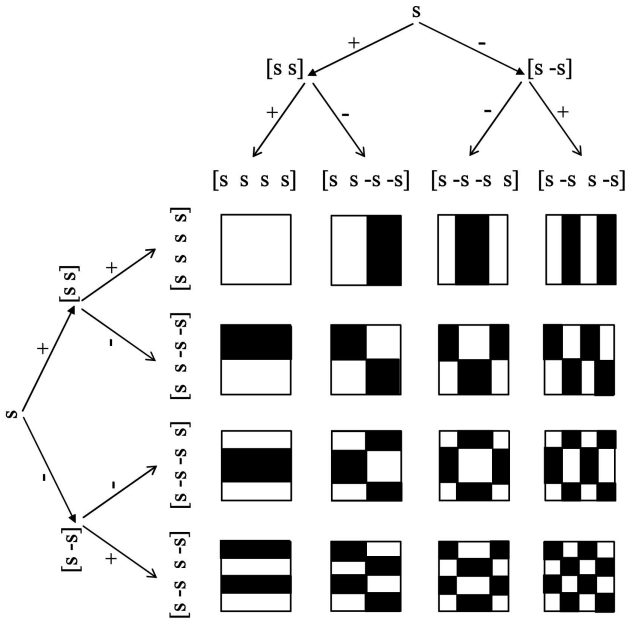


Figure 5: The 4x4 set of Walsh Hadamard kernels ordered by increasing frequency. Lowest frequency is top left. The two binary trees are the recurrences that generate the 1D WH kernels. The matrix is the 2D WH kernels generated through applying the outer product to the 1D WH kernels.

patches onto the basis elements:

$$||p_1 - p_2||^2 = \sum_{j=1}^{n} (w_j \cdot p_1 - w_j \cdot p_2)^2, w_j \in WH_k^D \tag{6}$$

where p_i is a patch in vector representation, w_j is a WH kernel in vector representation, n is the number of elements in the vectors and WH_k^D is the full set of WH kernels of that size.

Patches without any transformation has the information spread equally amongst it's features, which means that computing the L_2 distance the total accumulates at a constant pace. WH kernels, when ordered by frequency, has more of the information contained in earlier features. This means that a good lower bound on L_2 can be computed using less than the full amount of kernels.

Furthermore, assuming that the projections have been computed already, it has been shown(Ben-Artzi et al. [3]) that the summation can be done by using only two additions per patch per kernel if the summation of the kernels is done in a specific order. This ordering forms a Gray-Code sequence of the WH kernels. In CSH the projection of patches onto Walsh Hadamard kernels serve as a form of reduced dimensionality. The projections are computed once and used multiple times.

A WH kernel can only be computed efficiently from a different WH kernel if the two are closely related. Two WH kernels are closely related if the recurrence(equation 4) that created them differs only in a single sign. As an example(figure 5) the kernel [s s s s] is closely related to [s s -s -s] and [s -s s -s], but not [s -s -s s]. The efficient computation is done pixelwise through the following formula

$$WH_+(x) = WH_-(x) + WH_+(x-k) + WH_-(x-k) \tag{7}$$

where the sign is based on what sign they have, x is the pixel, and k is a constant that depends

on where in the recurrence the sign differed. A similar equation exists for $WH_{-}(x)$, which is trivial to isolate.

4.2 Indexing

Indexing in the CSH algorithm works similar to that of LSH: to obtain the indexes several hash functions are applied and the results are concatenated together. The hash functions used for CSH differ from those of LSH, which has some implications for the results.

- 2D Walsh-Hadamard kernels are used instead of the family of hashing functions used in Locality Sensitive Hashing. Each patch is transformed using a predefined set of lowest frequency WH kernels instead of a random set of lines.
- LSH uses multiple random functions of the form $h(v) = \frac{a \cdot v + b}{r}$ to compute the hash codes, where r is the width of the bins, b is a random value in the interval [0,r), v is a vector representing a patch, and a is a vector of the same size as v with random entries in the interval [0,1). The values of v are upper bounded as they are the intensities of an image, which means that the number of bins depends inversely on r.

  In CSH a fixed number of bins is assigned to each projection. This is done to reflect some projections containing more information than others. It should be noted that the number of bins has an impact on the number of bits in the final hash code.
- LSH has fixed width spacing. This means that all bins has intervals of the same size, while the number of elements in each bin may be and likely is different. CSH has variable width spacing. This means that the edges of the bins are chosen such that the amount of elements in each bin is the same. The size of the intervals of the bins may be and likely is different.

The set of kernels used and the number of bins for CSH was determined by Korman and Avidan offline for their implementation and I have used the same values. For the sake of convenience these are shown in table and are fixed for all images. These numbers were chosen for their performance and reflects their observations regarding WH kernels and the impact of each channel.

patch dim.	$Y_{0,0}$	$Cb_{0,0}$	$Cr_{0,0}$	$Y_{1,0}$	$Y_{0,1}$	$Y_{1,1}$	$Y_{2,1}$	$Y_{1,2}$
16 or 8	32 (5)	4 (2)	4 (2)	8 (3)	8 (3)	2 (1)	2 (1)	2 (1)
4	32 (5)	4 (2)	4 (2)	8 (3)	8 (3)	-	2 (1)	2 (1)
2	32 (5)	4 (2)	4 (2)	8 (3)	8 (3)	-	-	-

Table 3: Bit/kernel allocations. The table shows the number of bins allocated per kernel with the corresponding number of bits in parenthesis. Patch dim corresponds to the patch size, i.e. 16 corresponds to patches of 16x16. Y, Cb, and Cr are the channels used for each bit. Subscript indicates what projection is used with 0,0 indicating the upper left-most kernel as seen in figure 4. These numbers are calculated offline and has been copied from Korman and Avidan [12].

The even amount of patches per bin means that the bin edges needs to be computed before binning is possible. This is done by taking a large sample of patch projections and using those to place the edges at the relevant percentiles. Like in LSH a random offset is applied to the bin edges so as to avoid the effect of quantization on bin limits. The expectation is that all bins have the same number of elements with the exception of the first and last bin.

The result of binning is a number of bits per patch per kernel, which are concatenated to create a hash code per patch for a specific set of offsets. The offset is the only source of variability for CSH and so the construction of the L hash tables depend on this.

4.3 Searching

The search stage makes use of the L hash tables constructed in the indexing stage. The hash tables exhibit local sensitivity, i.e. that similar patches hash to the same bin. The LSH approach is to exploit this by searching each of the L bins that a patch hashes for candidates and then compare these candidates.

This does not make use of spatial arrangement nor does it propagate information to neighbouring patches.

4.3.1 Candidate creation

CSH expands on the LSH algorithm by considering candidates based on more than just the hash.

For each iteration of the algorithm, which each makes use of one of the L hash tables, the purpose is to generate candidate patches for each patch in the source image. Candidate patches are found in the target image and are candidates for being the nearest neighbour patch of a specific patch.

Korman and Avidan [12] make use of four observations for determining good candidates. Let s_i denote a patch in the source image and t_i denote a patch in the target image

1. If s and t hash to the same entry then t is a good candidate for s. This is the kind of candidates that LSH uses exclusively.
2. If t is a candidate for s_1 then t is also a candidate for any patch s_2 that hashes to the same entry as s_1.
3. If t_1 is a candidate for s then any patch t_2 that hashes to the same entry as t_1 is a candidate for s.
4. If t_1's left neighbour t_2 has a candidate s_2 then s_2's right neighbour is a candidate for t_1. This goes for all four cardinal directions.

Observations 1-3 follows from the local sensitivity property of the hash function used to create the hash tables. Observation 4 comes the assumption that patches in images have some sort of coherency to them. These combine into 3 types of candidates, which are shown in figure 6.

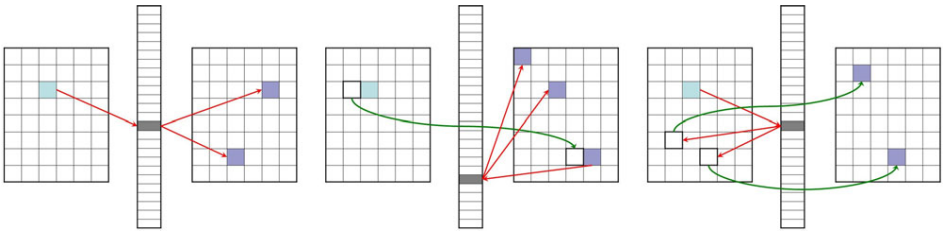


Figure 6: The various types of candidates created in CSH. For each type we have the source image on the left, the target image on the right and the hash table in the middle. (left) These are type 1 patches, which are patches that hash into the same index. (middle) These are type 2 patches, where patches propagate good matches to neighbouring patches. (right) These are type 3 patches, where patches propagates matches to patches that share the same hash.

This creates a number of candidates. The number depends on the width k of hash table, which is the number of patches per entry that is stored. The original paper(and us) uses a width of 2. Type 1 generates k candidates, type 2 generates k+1 candidates per cardinal direction, and type 3 generates k candidates. These candidates may overlap. The total number of candidates is 6k+4 or 16 with a width of 2.

4.3.2 Candidate Ranking

Given a set of candidates it is necessary to rank them. Computing the distance of each candidate is the main time consumer of the function. As such an approximation of the L_2 distance is used. This approximation is based on Walsh-Hadamard kernels as mentioned earlier.

Each patch is randomly assigned a current match for nearest neighbour at the start of the search stage. The lower bound for the match is computed using the full set of L_2 kernels. It should be noted that this set is still just a subset(23 out of 192) of the actual full set of kernels. As mentioned previously the earlier kernels capture more of the information regarding the distance between two patches. This not only allows the lower bounding to be used without suffering significantly from lack of precision, but is also means that candidates can be rejected quickly, as bad candidates are likely to grow over the estimate quickly.

4.3.3 High-level CSH: the algorithm

1 function Project (image, a);
- **Input :** An image of shape (x,y,3), where x and y are the dimensions of the image and 3 is the channels. A positive integer a, which corresponds to the size of patches.
- **Output :** An array of shape (x+1-a,y+1-a,t)=(r,s), where r and s are the dimensions of the patches, while t is the number of kernels.
- **Purpose:** Apply the Walsh Hadamard transformation on the patches of an image to achieve reduced dimensionality.

2 function Create_hashes ($proj_s$, $proj_t$, l);
- **Input :** The projections for the source and target image with shapes (x,y,t) and (n,m,t) respectively, where x,y,n,m are the dimensions and t is the number of projections. A positive integer l, which is the number of iterations in the propagation stage.
- **Output :** Two arrays of shape (x,y,l) and (n,m,l), which corresponds to the hash codes of the source and target images respectively.
- **Purpose:** Use the input to create b hash functions. Apply each hash function to each pixel. Return l hash codes per patch.

3 function Create_tables (hashes,l,k);
- **Input :** The hashes for an image of shape (x,y,l), such as those created by "Create_hashes", where x,y are the dimensions of the image. Two positive integers l and k, which is the number of iterations and the width of the hash table respectively.
- **Output :** A hash table of shape (l,n,k) where n is a constant and l,k is copied from the input.
- **Purpose:** Takes a set of values, for each unique value finds the indexes containing that value. Returns a hash table that contains k representatives for each unique value.

4 function Initialize_matches ($proj_s$, $proj_t$);
- **Input :** The set of projections for two images, such as those generated by "Create_hashes". These have shape (x,y,t) and (n,m,t) respectively.
- **Output :** Two arrays of shape (x,y).
- **Purpose:** Takes the Walsh Hadamard projections of two images and returns an array of size (x,y) of tuples of integers with values in the range (0:n, 0:m). This array is the set of matches for each patch of the first image. Also returns the distance to each match.

5 function Find ($hash_s$, $hash_t$, $table_s$, $table_t$, matches);
Input : An array of hashes of shape (x,y). An array of hashes of shape (n,m). A hash table of shape (x,y,k). A hash table of shape (n,m,k). An array of matches of shape (x,y).
Output : An array of candidates of shape (x,y,c), where c is a constant.
Purpose: Finds the type 1, 2, and 3 candidates for each patch and returns them. There are c candidates for each patch.
6 function Distance ($proj_s$, $proj_t$, candidates);
Input : An array of projections of shape (x,y,t). An array of projections of shape (n,m,t). An array of candidates of shape (x,y,c), where c is the number of candidates. The projections were created by "Project" and the candidates by "Find"
Output : An array of distances of shape (x,y,c).
Purpose: Given a set of candidates returns the distance to each candidate. The projections are the coordinates of the points representing each candidate.
7 function Pick (matches, match_dist, candidates, cand_dist);
Input : Two arrays for the matches containing their ID and distance respectively, both are of shape (x,y). Two arrays for the candidates containing their ID and distance respectively, both are of shape (x,y,c), where c is the number of candidates per patch.
Output : Two arrays of shape (x,y).
Purpose: Takes the candidates and compares them to the matches patch-wise. For each patch returns the match or candidate with the lowest distance along with the distance.
8 function CSH ($image_s$, $image_t$, a,b);
Input : Two images of shape (x,y,3) and (n,m,3) respectively. Two positive integers a and b, which is the size of the patches and the number of iterations in the propagation stage respectively.
Output : An array of size (x+1-a, y+1-a).
9 $proj_s$ = Project($image_s$, a);
10 $proj_t$ = Project($image_t$, a);
11 $hash_s$,$hash_t$ = Create_hashes($proj_s$, $proj_t$, b);
12 $table_s$ = Create_tables($hash_s$, b, 2);
13 $table_t$ = Create_tables($hash_t$, b, 2);
14 matches, dist = Initialize_matches($proj_s$, $proj_t$);
15 **for** $i = 0$ *to* b **do**
16 cands = Find($hash_s$, $hash_t$, $table_s$, $table_t$, matches);
17 cand_dist = Distance($proj_s$, $proj_t$, cands);
18 matches, dist = Pick(matches, dist, cands, cand_dist);
19 **end**
20 return matches;

The high level algorithm is shown in terms of purpose, input and output of each step. The final algorithm shows the complete algorithm and how each part contributes to the complete algorithm. KNN and the width of the hash table are input parameters in the final version, but have been left out for simplicity.

4.3.4 K nearest neighbours

The use of reduced dimensionality may mean that the found nearest neighbour is not the optimal one and that a better candidate was checked, but that the lower bound used for ranking candidates ended up ranking them inversely.

A K nearest neighbour approach can be used to mitigate this. Saving several matches per patch allows one to rank them using the full and original set of features for a more accurate ranking. This changes the algorithm in two places.

1. Finding candidates now have a greater number of matches to work with. This allows the creation of a richer set of candidates per iteration. The tradeoff is an increased cost per iteration, which means that less iterations can be done in the same amount of time.

2. Picking the best candidate now has to consider multiple matches instead of one.

Finding the K best candidates can no longer be done merely by finding the minimum value in a list. The simple approach is to brute force it, which would result in something similar to the following when done sequentially:

```
# Input consists of arrays corresponding to a single patch.
# For each candidate, check if it is better than the worst match
# If it is better sort it into the list of matches and evict the worst match
# Performs inplace updates of matches and match_dist
def brute_force_pick(matches, match_dist, cands, cand_dist):
    KNN = len(matches)
    for i in range(len(cands)):
        if cand_dist[i]>=match_dist[knn-1]: continue
        else:
            cur_dist = cand_dist[i]
            cur_cand = cands[i]
            for j in range(KNN):
                if cur_dist < match_dist[j]:
                    cur_dist, match_dist[j] = match_dist[j], cur_dist
                    cur_cand, match[j] = match[j], cur_cand


# Input is the full arrays, i.e matchess has size (patch_count, KNN)
# Is the functional equivalent of a map over all patches
# Performs inplace updates of matchess and match_dists
def brute_force_pick_all(matchess, match_dists, candss, cand_dists):
    for i in range(len(matchess[:,0]):
        brute_force_pick(matches[i], match_dists[i], candss[i], cand_dists[i])
```

The alternative to bruteforcing it is to sort the array, however the above algorithm is O(kn) and so sorting provides no asymptotic benefit.

5 Data Parallel CSH

In this section I will describe the implementation of data parallel CSH. The high-level description is the same as sequential CSH, but there are important differences in implementation. Critical sections are projecting the Walsh Hadamard Kernels and picking the best candidates as these would be bottlenecked by sequential operations without changes.

The algorithm is assumed to be limited by memory and not by operations performed. It is of interest to establish an estimate of how many memory operations each part requires as this provides a way to determine how efficient the implementation is at utilizing the hardware.

5.1 Projecting the patches

This section details the work required to transform an image represented as fields to one with lower dimensionality where the features have been computed by Walsh Hadamard transformations. Efficiently computing the projection of the patches on Walsh-Hadamard kernels requires a previous Walsh-Hadamard kernel to have been computed. As such each channel requires

a starting projection to be computed using a slower approach. All 3 channels use the same projection as their first projection, which is done by summing the features.

```
let p0 (image: [n][m]i32) =
    map (\x ->
        map(\y ->
            reduce (+) 0 (map (reduce (+) 0) (image[x-7:x+1, y-7:y+1])
        ) (iota m)
    ) (iota n)
```

In the preceding snippet we see the code required for computing the initial Walsh Hadamard projection. For the sake of simplicity this code does not handle boundary issues, which in practice is handled by treating out of bound accesses as zeroes.

Computing the initial projection is a slow process as each pixel requires summing the entire field, which for an 8x8 is 64 values. The efficient computation of a Walsh Hadamard projection requires the existence of a related projection to precede it. No such kernel can exist for the first projection, which is why the slow approach is necessary. The kernels used for CSH are related to each and can be computed using the Gray code steps shown in figure 7.

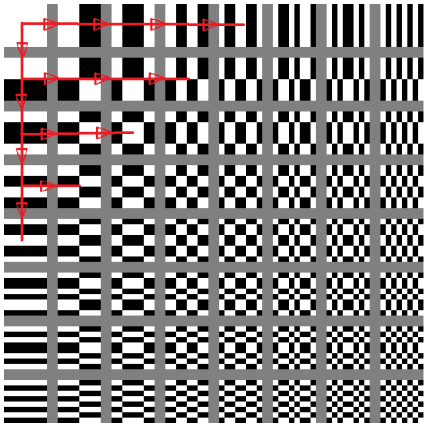


Figure 7: Sequence for computing the Walsh Hadamard projections for a patch size of 8x8. 15 kernels are computed for the shown channel. Steps along the rows are done on the preceding kernel, while steps along the column is done on the transposition of the preceding kernel and produces the transposed kernel. This is done so that data can be accessed in coalesced fashion.

5.1.1 Translating Sequential Recurrences into Parallel Scans

The sequential approach to computing the projections of a 1D image efficiently uses one of the two following formulas depending on the relation between the two projections. As seen with equation 7 the relation between kernels depends on what sign they differ in in the recurrences that generated them. The first projection of a channel $p_0(j)$ is computed using the slower approach. The rest are computed using the efficient approach, which is given by:

$$p_i(j) = p_i(j-k) + p_{i-1}(j) + p_{i-1}(j-k) \tag{8}$$

$$p_i(j) = -p_i(j-k) + p_{i-1}(j) - p_{i-1}(j-k) \tag{9}$$

where i denotes the projection number, j the patch number and k depends on how the two projections are related. As written this is an inherently sequential operation, but it can be rewritten and computed in parallel using a scan and linear function composition. This increases the cost of the scan by a constant scalar.

To parallelize the recurrence we want the recurrences to be in a format that scan has a known solution for. I start with the simple case of k=1, which can later be generalized. To start with I group the constant terms into a single term and introduce s, which is either + or -, which allows the two equations to be grouped as one.

$$p_i(j) = s * p_i(j-1) + a(j), a(j) = p_{i-1}(j) + s * p_{i-1}(j-1) \tag{10}$$

The term $a(j)$ can be computed before the scan. The above recurrence, if s was +, corresponds to the typical inclusive scan. The case where s is - is not so simple however and the recurrence 10 can instead be solved using linear function composition. The generic form for solving linear recurrences using function composition is as follows:

$$\begin{aligned} f_0(x) &= a_0 + b_0 * x \\ f_1(x) &= a_1 + b_1 * x \\ f_1 \circ f_0(x) &= a_1 + b_1 * (a_0 + b_0 * x) \\ f_1 \circ f_0(x) &= (a_1 + b_1 * a_0) + b_1 * b_0 * x \end{aligned}$$

This allows us to extract

$$f_1 \circ f_0(x) = f'(x) = a' + b' * x \tag{11}$$
$$a' = (a_1 + b_1 * a_0) \tag{12}$$
$$b' = b_1 * b_0 \tag{13}$$

By putting in values where relevant it can be done using a scan such as the following.

```
let scanvals =
    scan (\(a0,b0) (a1,b1) -> (a1 + b1*a0, b0*b1)
        ) (0,1) vals
in map (\(a,b) -> a+b*x) scanvals
```

where vals is a list of tuples (a,b) where a is given by 10 and b is the sign. It is important to note two things.

- The solution has been shown for scans with recurrences of stride 1(i.e. f(x) = f(x-1)), but we need something that can handle other strides. We observe that for a stride of 2 there are two non-overlapping sequences: $f_0(x), f_2(x), f_4(x), ..., f_{2m}[x]$ and another offset by 1: $f_1(x), f_3(x), f_5(x), ..., f_{2m+1}(x)$, where each element depends on the prior element. A single sequence can be solved using the same scan that was the solution to stride 1. Similar can be observed for recurrences with a stride above 2.

  In summary: the solution to recurrences with a stride greater than 1 is to segment the recurrence and scan each segment separately.

- What is x? Patches are computed using a patch to the left of them. If we go far enough to the left we exit the image, where we can set pixels to any value. If we set these pixels to 0 then the Walsh Hadamard projection based on them is also 0, which is what x is. This is the reason why the image is padded, as x would depend on the leftmost pixels otherwise.

5.1.2 Implementing Efficient Walsh Hadamard transformation in Futhark

The parallel version has been shown using scans, however scans work based on stride 1, which as a recurrence has the form $p_i(j) = p_i(j-1) + p_i(j)$, but the mentioned formula has stride k. The recurrence does not overlap, which means one can simply partition each recurrence into it's own segment and scan them separately.

The scan for a row is done as follows:

```
-- Input is a row of (sign, val)
-- output is the computed projection for those values
let gray_code_step_segment vals =
    let scanvals =
        scan (\(a0,b0) (a1,b1) -> (a1 + b1*a0, b0*b1)
            ) (0,1) vals
    in map (\(left,_) -> left) scanvals

-- Input is the sign and stride (both determined by the relation of the two
    projections)
-- Prior is the previous projection
-- output is the projection that is computed
-- Sign is either -1 or +1 for Walsh Hadamard transformations
-- Stride is 2^m where m depends on the relation
let gray_code_step [n] sign stride (prior: [n]i32) =
    let afuns =
        map (\i ->
            if stride - 1 < i
            then (prior[i] + (sign * prior[i-stride]), sign)
            else (prior[i], sign)) (iota n))
    let res =
        loop res = afuns for j < stride do
            let res[j:m:stride] = gray_code_step_segment
                                    (map (\x -> (sign, x)) res[j:m:stride]
            in res
    in res
```

The above code is 1 dimensional and the scans are computed using a sequential loop for the sake of simplicity. The scans are independent and can be computed in parallel. The dimensionality does not matter when it comes to computing the projections. The inputs to the function are the sign and the prior. The prior is the projection that can be used to compute the next projection, while the sign is either 1 or -1 depending on how the projections relate to each other. More information on this was shown in section 4.1.

A naive approach to calculating the number of global memory accesses assumes values are not kept in local memory for long. This is a reasonable assumption as each projection depends on the prior projection and values are typically not reused for each index, with the sole exception to this being the scan. The cost of computing the projections can be divided into two parts: the initial projection p_0 for each channel and the remaining 20 projections.

Computing the initial projection requires accessing all features of the patch and writing the final sum, which is 64 reads and 1 write for an 8x8 patch. This needs to be done for all patches and all 3 channels. The remaining 20 projections can be done efficiently through a gray code step. The code is primarily composed of a map with two reads and two writes, a scan with one read and one write, and finally a map with one read and one write. More than this may be necessary depending on how available parallelism is best utilized, but it would not change the lower bound.

The use of padding means more patches need to be handled and for this stage only the number of patches is equal to the number of pixels. The amount of global memory accesses for computing the projections for one image is 65*3+8*20=355 accesses per pixel.

5.2 Creating hash values

Creating the hash values is done using the Walsh-Hadamard projections. As mentioned previously each projection contributes a number of bits of the final hash code. The process for a single projection is as follows:

```
let create_hashes [n] [m] [r]
            (projs_s: [r][n]i32)
            (projs_t: [r][m]i32)
```

```
4             (l: i32)
5             =
6     let arg_sort_both =
7         map2 (\x y -> radix_sort_int_by_key (\(_,i) -> i)
8                 i32.num_bits i32.get_bit
9                 (zip (iota (n+m)) (x ++ y))
10             ) projs_s projs_t
11    let both_pos =
12        map (\x ->
13            let (y,_) = unzip x
14                in scatter (replicate count_sum 0) y  (iota count_sum)
15            ) arg_sort_both
16    let projs_sort_s = both_pos[:,0:n]
17    let projs_sort_t = both_pos[:,n:]
18    -- Initialize various constants and randomization
19    -- f g and h are used as stand in functions to simplify the code
20    let rand_offsets = ... :> [l][r]i32
21    let bin_count = ... :> [r]i32
22    let conc_offsets = ... :> [r]i32
23    let hash_s =
24        map (\x ->
25            map (\y ->
26                let vals = map (\i ->
27                    let hash_i = (projs_sort_s[i,y]*bin_count[i] + rand_offset[i])/
                          count_sum)
28                    in hash_i<<conc_offsets[i]) (iota r)
29                in reduce (+) 0 vals
30            ) (iota n)
31        ) (iota l)
32    let hash_t = ... -- same to hash_s
33    in (hash_s, hash_t
```

The above code show the implementation for computing the hashes. Some parts have been left out for brevity. The function takes the projections of both images as input along with an integer l, which is the number of iterations in the propagation stage. Not all projections are used for creating the hash values. For a patch size of 8x8 the count is 8 out of 23.

Creating the hash code for a patch is composed of two parts: For each projection find the bin the current patch would hash into. These bin numbers are then concatenated to create the hash code. In my implementation these steps are handled as follows:

- **Finding the bins:** The input projections have type [r][n] and [r][m], which means the total amount of patches is n+m, while the number of projections is r. Each of these projections is sorted individually using radix sort for the purpose of finding the index each patch would sort into for that specific projection.

  This index is then translated into a bin number using the formula (projs_sort_s[i,y] * bin_count[i] + rand_offset[i])/count_sum), which has the same shape as $\frac{a*v+b}{r}$. I previously claimed(section 4.2) that such a function led to a variable amount of patches in each bin. This is not the case here due to using a sorted list, which is a list of values that are evenly spaced. Applying the hash function to a list of evenly spaced values yields an equal amount of patches in each bin.

  $\frac{\text{projs\_sort\_s[i,y]}}{\text{count\_sum}}$ is a number in the range [0,1), while bin_count is an integer equal to the number of bins for the i'th projection. The number of bins for a given projection depends on the importance of the projection. These values are determined offline.

  The array rand_offsets is the equivalent of b and is an offset to the placement of the bin edges. It's purpose is to neutralize the effect fixed bin edges has on the probability of two similar values binning similarly.

- The produced hash codes are integers, which were produced by the concatenation of integers with a smaller bit-count, fx. two 2-bit integers produces a 4-bit integer: $11 +\!\!+ 10 = 1110$. This can also be done through addition if an appropriate offset is applied to the integers, which is what the combination of reduce (+) and left shifting is used to achieve.

Sorting and applying the hash function is the main sources of memory accesses for creating the hash values. Radix sort is done in several passes. Each pass consists of a scan and scatter. The scan requires 1 read and 4 writes while the scatter requires 4 reads and 1 write. The number of passes depends on the size of the keys used for sorting. 2 bits per key is handled each pass, which means 32 bit integers requires 16 passes. 8(the used number of projections) calls to Radix sort is issued, which makes the total cost 16*10*8=1280 accesses per patch.

Applying the hash function is done once per patch per iteration per Walsh-Hadamard kernel used. For both patch size 8 and 16 the number of used kernels is 8, so this comes out to 8 accesses per iteration per patch.

5.3 Creating hash tables

The purpose of the hash tables is to enable figuring out what patches have the same hash value, i.e. to find collisions. The width of the hash table is fixed, which is done to keep the array regular. An entry in the hash table contains the indexes of patches that would hash into that entry.

Creating the hash table requires finding the patches that would hash into the same entry. This can be done by sorting the patches by their hash values, which would place patches with the same hash value in consecutive order. The hash table can then be generated from this sorted list merely by picking random elements from each group. For the sake of simplicity the elements that are picked are the first two.

```
1  let create_hash_table [n]
2      (hash: [n]i32)
3      (span: i64)
4      (width: i64)
5      : ([span][width]i32) =
6      let (ind_r, hash_r) = unzip (radix_sort_int_by_key (\(_,i) -> i)
7                                   i32.num_bits i32.get_bit (zip (iota n) hash))
8      -- pick width codes to keep
9      let flags = map2 (\i j -> i != j) (rotate (-1) hash_r) hash_r
10     let offsets = segmented_scan (+) 0 flags (replicate n 1)
11     let indices = map2 (\offset x ->
12                         if offset <= width
13                         then x*width+offset-1
14                         else -1
15                        ) offsets hash_r
16     let spanwidth = span*width
17     let hash_table_flat =  scatter (replicate spanwidth 0) indices ind_r)
18     in map (unflatten span width) hash_table_flat
```

The shown function generates a hash table given an array of hashes and two integers span and width, which define the dimensions of the hash table. A hash is an integer in the interval [0,span). Sorting is used to group patches by their hash. Lines 9-15 is used to pick the elements that are going to make up the hash table. The first map is used for finding the first element of each group. The scan is then used to number each patch by their position in their group. The second map is used to pick the elements that are going to make up the hash table. All patches are either assigned a unique index in the hash table or given an out of bounds memory location(-1). Scatter handles out of bounds writes by ignoring them. The hash table is finally constructed using a scatter.

The cost of this is dominated by the sorting, which requires 160 accesses per iteration per patch.

5.4 Propagation

Propagation is composed of 3 parts:

1. Picking candidates.
2. Calculating the distance of candidates.
3. Picking candidates to keep as matches

Initially a set of random patches is used as the first set of matches. As the set of matches is much smaller than the set of candidates for even a single iteration this part is not performance critical and won't be discussed further.

5.4.1 Finding new candidates

Picking the candidates is based on the set of methods described in section 4.3.1. 3 different types of candidates exist(figure 6), where type 1 candidates exploit the similarity of patches, type 2 propagate good matches to neighbouring patches, and type 3 propagate good matches to similar patches. The implementation of these are similar and in the following code only type 1 candidates are shown.

```
let find_all [n] [o] [w]
    (hash_s: [n]i32)
    (hash_table_t: [o][w]i32)
    : [n][]i32 =
    let find_i i =
        let type1 = hash_table_t[hash_s[i],:]
        in type1
    in map (find_i) (iota n)
```

The function find_all takes as input two arrays. The array hash_s contains the hashes of the patches of the source image. These hashes are used to index into hash_table_t, which contains patches from the target image that would hash into that index. The hash table contains w representatives per index. Only type 1 candidates are shown, but the process of finding type 2 or type 3 candidates are no different and only differ in the specific arrays used. Finding candidates is a process of chain indexing, where the result of indexing into one array is used to index into the next array.

The majority of indexing when finding candidates are random accesses to memory. If we only exploit the outer parallelism of the map then the first set of memory accesses can be done in coalesced fashion. Threads in the same warp will have consecutive values of i, which means the threads perform memory accesses to hash_[i] in consecutive order. The result of these accesses are not ordered however, which means the next step of reading from hash_table_t is not done to consecutive locations in memory. A similar behaviour can be observed for type 2 and type 3 candidates. Most steps for finding candidates does not involve coalesced access to memory(figure 8). In practice finding the candidates has proven to be a minor contributor to the run-time and so further optimizations has not been pursued.

5.4.2 Computing the distance

The second part of propagation is calculating the distance for each candidate. Calculating the distance is expensive and is the primary motivation for using reduced dimensionality. This is done via:

```
let dist2  [k] (xs: [k]f32) (ys: [k]f32) : f32 =
    reduce (+) 0 (map2 (\x y -> (x-y)*(x-y)) xs ys)
let candidates12 =
```

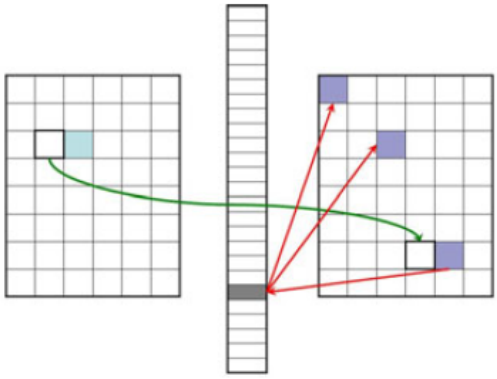


Figure 8: Type 2 candidates. Stepping to the neighbour can be done coalesced, however the match(green arrow) and hash(red arrows) are random access and is unlikely to be coalesced.

```
4     map2 (\x ys ->
5         map (\y -> dist2 wh_src_trs[x,:] wh_trg_trs[y,:]
6             ) ys) (iota patch_count) candidates
```

Finding the distance between two points is a straight forward issue. It is the root of the sum of the difference between each feature. In this case the root has been left out as the distance is used for comparisons and $x^2 > y^2$ returns the same result as $x > y$ for positive x and y, which is the case for distances.

5.4.3 Picking the best candidate

The last part of propagation is picking the nearest neighbours. Given a sorted list of k matches and n candidates the goal is to find the best matches. This could be done by sorting directly, however the list of candidates is much longer than the list of matches and by proxy the cost of sorting too expensive.

A sequential approach to bruteforcing has been shown previously. This approach sorts a candidate into the list of matches, which is repeated until all candidates for the current patch has been checked. This approach would not work well in a parallel regime. The lack of parallelism means that threads in the same warp would be working on different patches. It is possible to order the data such that coalesced access happens, but the method used to sort requires a lot of if-statements, which means the sequential approach would suffer from thread divergence if implemented in a parallel language.

A different approach that makes use of intra-group parallelism can instead be created. As with the sequential approach coalesced access to memory happens, but thread divergence will no longer be an issue. As mentioned previously the divergence happens when sorting candidates into the list of matches, which means a parallel variant needs to avoid or mitigate this.

This is done by iterating over two steps. The first step is taking the current list of candidates and finding the one with the lowest distance. The second step is comparing the candidate to one match to determine whether it's worth keeping and replaces that match if so. This is repeated until all matches have been replaced or the best candidate isn't worth keeping. Candidates that have been found the best are ignored for these repetitions. The result of the loop is a list where any remaining matches are in the start of the list in ascending order and the rest of the list are the kept candidates, which are in descending order. This is then sorted sequentially.

The first step uses a simple reduce pattern, which is an example of intra-group parallelism. The second step does branch, but this does not result in thread divergence. The branching

statement determines whether or not to repeat, however the warp is working on the same patch, which means the result of that branch is the same for all threads in the warp.

```
let bruteForcePar [k] [n]
            (matches: [k]i32)
            (match_dist: [k]f32)
            (candidates: [n]i32)
            (candidate_dist: [n]f32)
            : ([k]i32, [k]f32) =
    let knn = copy (zip matches match_dist)
    let dists = copy candidate_dist
    let cycle = true
    let j = 0i64
    let op = \(i1,v1) (i2,v2) ->
              if v1 < v2 then (i1, v1) else
              if v1 > v2 then (i2, v2) else
              (if i1 <= i2 then i1 else i2, v1)
    let (_, knn, _, _) =
        loop (dists, knn, j, cycle)
            while cycle && (j < k) do
                let (min_ind, min_val) =
                    reduce_comm op
                        (n, f32.highest)
                        (zip (iota n) dists)
            in  if min_val < (knn[k-1-j].1)
                then  let dists[min_ind] = f32.highest
                    let knn[k-1-j] =
                        (candidates[min_ind], min_val)
                    in  (dists, knn, j+1, true)
                else  (dists, knn, j, false)
    let knn_sort = sortPartSortedSeqs knn
    in  unzip knn_sort
```

The number of memory accesses for each iteration is:

- Finding candidates contributes roughly 2 accesses per candidate per patch.
- Computing the distance contributes twice the number of features per candidate per patch. The number of features for something of patch size 8x8 is 23.
- The loop that picks candidates is run at most k times, where k is the number of nearest neighbours found for each patch. Each loop has an amount of accesses equivalent to the number of candidates. Merging two sorted lists can be done with a number of accesses equivalent to the number of elements, which is k. The total cost is k accesses per (candidate+1) per patch.

This version of CSH only finds the nearest neighbours for the first image.

5.5 Cost of data-parallel CSH

CSH is a memory bound algorithm, which is why the number of accesses have been calculated for each part of CSH. Showing that my algorithm can utilize the GPU thus requires showing that it utilizes the bandwidth provided.

As can be seen in table 4 the propagation stage dominates the number of accesses performed for CSH when the number of iterations is high. Furthermore calculating the distance is the main bottleneck until K becomes high.

Source	Amount
Projection	355*[pixels in both images]
Hashing	(1280+8*I)*[patches in both images]
Hash Tables	160*I*[patches in both images]
Picking Candidates	2*I*C*[patches in source image]
Calculating Distance	2*WH*I*C*[patches in source image]
Picking Candidates	K*I*C*[patches in source image]

Table 4: Amount of memory accesses for patch size of 8. Each access is 32bit. I is the number of iterations performed. C is the number of candidates per patch. WH is the number of projections used, which is 23 for a patch size of 8. K is the number nearest neighbours found.

6 Performance/Accuracy Trade-offs for CSH

In this section I introduce various possible changes to CSH that attempt to improve the performance of CSH. These are distinct from the changes made to go from sequential to parallel CSH in that they do not attempt to improve the use of the underlying hardware, but rather attempt to improve the algorithm itself.

6.1 K Nearest Neighbours

KNN is a modification of the algorithm such that k nearest neighbour fields are produced for each field instead of 1 neighbour per field. This is done for two reasons:

- The ground truth produced using reduced dimensionality is not necessarily the ground truth when using full dimensionality. To mitigate this issue a good reduction is necessary, which is what the use of Walsh Hadamard kernels is supposed to achieve, but this merely reduces the risk. Producing multiple candidates per field is a way of mitigating this issue further as lower ranked candidates may in fact perform better on the full dimensionality set.
- In each iteration of the propagation stage candidates for nearest neighbours are produced from the current set of nearest neighbours. More neighbours allows for more candidates.

In both cases the addition of more candidates comes at a run-time cost when keeping the number of iterations constant. As such increasing K is a trade-off that should increase accuracy, but does so at the cost of an increase to runtime.

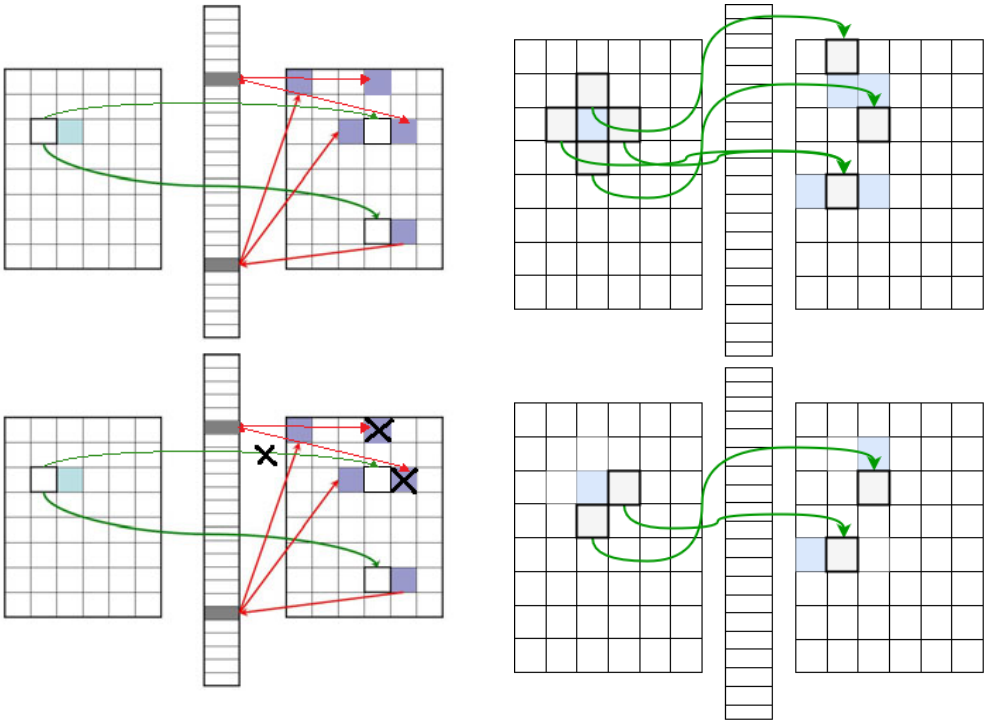
6.2 Generating less candidates

Calculating the distance to each candidate is the most expensive step. Less candidates generated will decrease the cost of such a step. In this section I will show two approaches to generating less candidates as outlined in figure 9.

6.2.1 LessKNN: Decreasing the impact of KNN on candidates created

The nearest neighbours produced from increasing K may not be useful. Locality is the assumption that similar patches hashes similarly, which is what most of the candidate creation is based on. The nearest neighbours of a field are similar patches, otherwise they would not have become the current nearest neighbours. This in turns means that they are likely to hash similarly, which would result in the creation of duplicate candidates. Duplicate candidates only serve to increase runtime and is something that should be avoided.

There are two approaches to this:



(a) Green arrows indicate matches. Candidates are ignored based on the rank of the match. Corresponds to the LessKNN approach. Shown is a match and the 3 candidates beloning to it being ignored, two of the candidates are duplicates.

(b) Grey boxes indicate neighbours. In the LessDir approach neighbours in certain directions are not visited. For the sake of simplicity the last step, which involves the hash table in the middle, is not shown.

Figure 9: Two ways of reducing the number of candidates generated. The two top images indicate baseline, while the two bottom images indicate the changed variant. Both variants primarily remove candidates created through type2.

- **Sorting the list of candidates and removing duplicates**: Sorting is an expensive operation and would create an irregular sized array of candidates. The size of the hash table was chosen such that only a few patches should hash into the same index. As such the expectation is that only a few duplicates would exist. As the run-time cost of sorting is expected to be higher than the time saved on less candidates, I have not created the variant suggested in this bullet point.
- **Ignoring the lower ranked neighbours**: If lower ranked neighbours are likely to produce duplicates then ignoring the lower ranked neighbours when it comes to creating candidates is a viable option. This approach can be seen in figure 9(a)

Decreasing the number of nearest neighbours used for the creation of candidates should result in a decrease in run-time at the cost of decreased accuracy when compared to the full KNN approach. It does however preserve the advantage of being able use full dimensionality to choose between several nearest neighbours.

6.2.2 LessDir: Decreasing the impact of coherency on candidates created

Candidates are divided into 3 types of which type 2 generates the most candidates. The types were described in section 4.3.1 and figure 6. The summary of type 2: Visit the left neighbour, go to that neighbour's match, go to that match's right neighbour, use that neighbour and the patches that hash to the same entry as candidates. This is repeated for each cardinal direction.

This creates a large number of candidates, which are based on the concept of propagating good matches to neighbours. Reducing the number of cardinal directions visited would decrease the run-time, but would likely do so at a cost of accuracy.

6.3 Early stopping

Early stopping is a way of reducing the average cost of comparing the distance of a candidate to a match. The distances computed using the reduced dimensionality are a lower bound of the true distance which requires the full dimensionality. The distance between two points j and k is $\sum_{i=0}^{N} ||p_{ij} - p_{ik}||$. Most importantly each term in the sum is positive, which means that if $\sum_{i=0}^{m} ||p_{ij} - p_{ik}|| > x$ then $\sum_{i=0}^{M} ||p_{ij} - p_{ik}|| + \sum_{i=M+1}^{N} ||p_{ij} - p_{ik}|| > x$, where x is the already known distance to the current match.

In other words if a lower bound is greater than a number then the whole sum will also be greater than that number. Early stopping makes use of this by making periodic checks when computing the distance, which reduces the average number of operations and memory accesses needed to process a candidate. This approach was used by Korman and Avidan [12] in their sequential implementation of CSH.

The same approach can however not be used for the parallel implementation due to poor performance. Early stopping can be handled by a parallel implementation in two ways. Either by using outer parallelism and letting one thread handle summation and comparison of one pair of points or by using the inner parallelism and letting a warp handle the summation.

The first approach suffers from introducing thread divergence. Threads in the same warp execute in lockstep, so all threads would need to stop early before any of them actually stop early. The likelihood that one thread in the warp has a good candidate is however high and so the end result is requiring more comparisons without any benefit.

The second approach suffers from lack of parallelism. An 8x8 patch has 23 features, which is not enough to fully utilize a warp and so further reducing the amount of features per summation would simply result in more threads idling.

This serves as an example that the optimal approach for a sequential implementation is not necessarily a good starting point for a data-parallel implementation.

7 Results

In this section results and their implications are shown and discussed.

7.1 Experimental setup

All experiments in this section were performed on Intel system with 20 GB DRAM, 16 Intel(R) Xeon(R) CPU E5-2650 cores using 2-way multithreading. The GPU used is a Geforce RTX 2080 Ti with 11 GB DRAM, 4352 cores running at 1.35Ghz under CUDA 11.0. It has a maximum bandwidth of 616 GB/s.

The Data-parallel CSH code has been written in Futhark, compiled using Pyopencl, and run using Python. The KD-tree code was written in Futhark, compiled using opencl, and run using Python. The performance of sequential CSH is not measured in this report as the run-time is far worse than that of data-parallel KD-tree [13]. The accuracy reported by Korman and Avidan [12] is used for comparison. As dataset I used the VidPairs dataset used in [12], which consists of

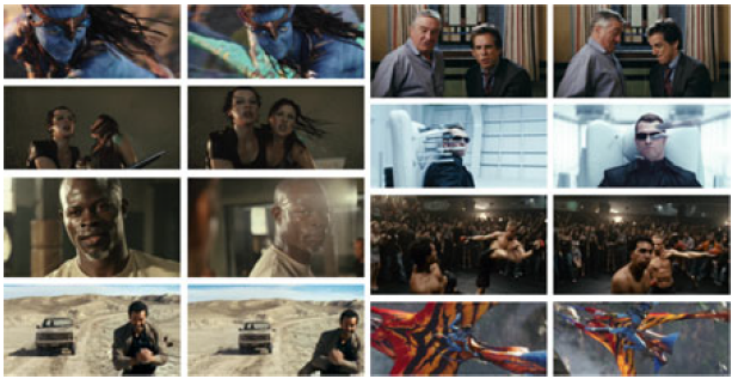
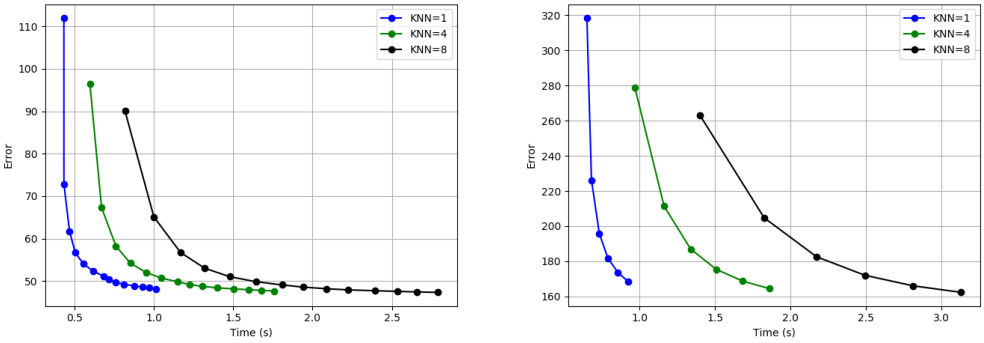


Figure 10: 8 pairs out of the 133 image pairs used as dataset.

133 pairs of images taken from movies. Each pair is taken from the same scene with a number of frames between each image(figure 10). This allows for some motion and perspective changes to happen between the images.

Results are obtained by running each pair of images once per entry and taking the average time and error as the results. In both cases a Python module is used to measure the time elapsed. The error is found as the average of the average distance between patches for each pair. The images are of roughly the same size and so any issues due to averaging results based on variably sized images is for the most part avoided.

7.2 Baseline



(a) Patch size is 8x8.

(b) Patch size is 16x16.

Figure 11: Error/Time tradeoffs of CSH using 3 different values of K for K nearest neighbours. Averages are over the 133 pairs of the VidPairs data set. Markers indicate the result after each iteration. Errors are given as the average l_2 distance between patches using the best match of the K matches.

The goal of this experiment is to investigate the effect finding a varying amount of nearest neighbours has on the run-time and accuracy on CSH. Figure 11 shows that increasing K has a positive effect on accuracy, but comes at a large cost to run-time. This is the case for both patch sizes.

The cost of run-time scales linearly with the amount of candidates created, which is not surprising. Increasing the number of candidates does increase the amount of parallelism, which can be exploited when it comes to determining the number of candidates to keep, however the primary cost is computing the distance, which does not benefit from the increased parallelism.

There are three distinct levels of parallelism when it comes to computing the distance of which only the feature level can avoid uncoalesced access. Using a warp for finding and summing the differences allows coalesced access as all data accesses for that warp will access the same set of features, which can be found consecutively in memory.

The alternatives has a thread compute the distance on it's own. This means threads in the same warp are accessing candidates, which aren't located consecutively. They would still be accessing source patches, which are located consecutively and be accessed in a coalesced fashion. Increasing the number of candidates technically helps with that last part, however the number of patches is above 1.5 million, which is more than enough to saturate the outer parallelism.

7.3 Less candidates based on KNN

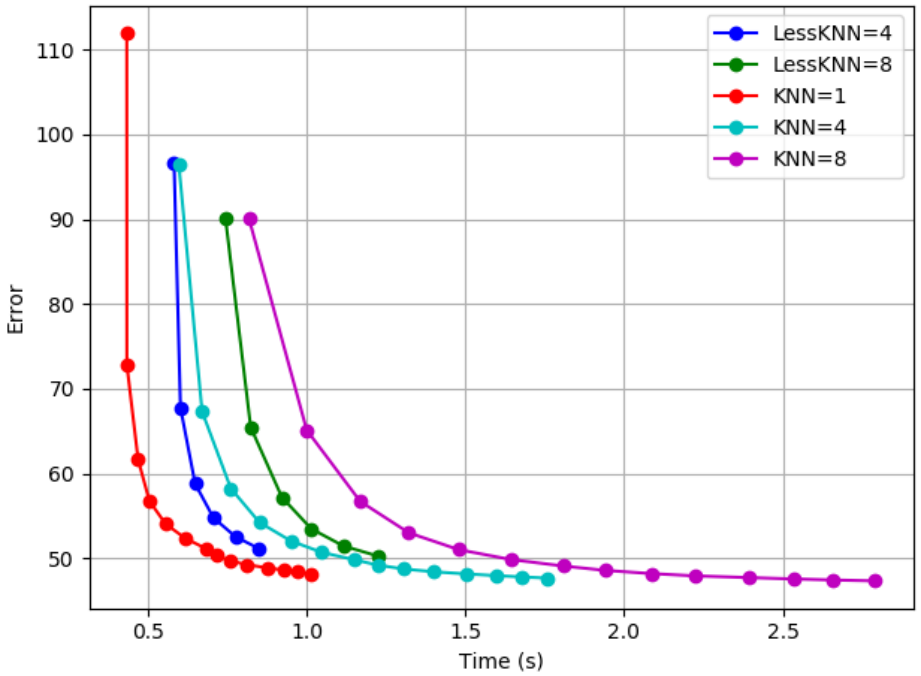


Figure 12: Error/Time tradeoffs of CSH. LessKNN is a method that disregards part of the K matches found when it comes to creating candidates. KNN is the baseline approach.

This experiment is to investigate the effect of reducing the number of matches used for creating candidates. LessKNN is a variation that only generates candidates using the better half the matches. The better half is determined on a per patch basis. Figure 12 shows that the

run-time is lower, which is predictable as half the number of distances need to be computed. The change has had a small negative effect on accuracy.

Predictably iteration 1 is unchanged. Good candidates for iteration 1 will come from either locality sensitivity, which does not depend on knn, or on propagation from randomly generated matches. Random generation is unlikely to result in good matches, which makes it extremely unlikely for a patch to have more than one good match before propagation. As only the best matches are used for generating candidates this makes it very likely that all good random matches are used for generating candidates for iteration 1.

From iteration 1 and on it is to be expected that some patches have many good matches, which would explain the difference between LessKNN and KNN from iteration 2 and on. Even the worst half of the matches candidates that are useful.

7.4 Less candidates based on coherency

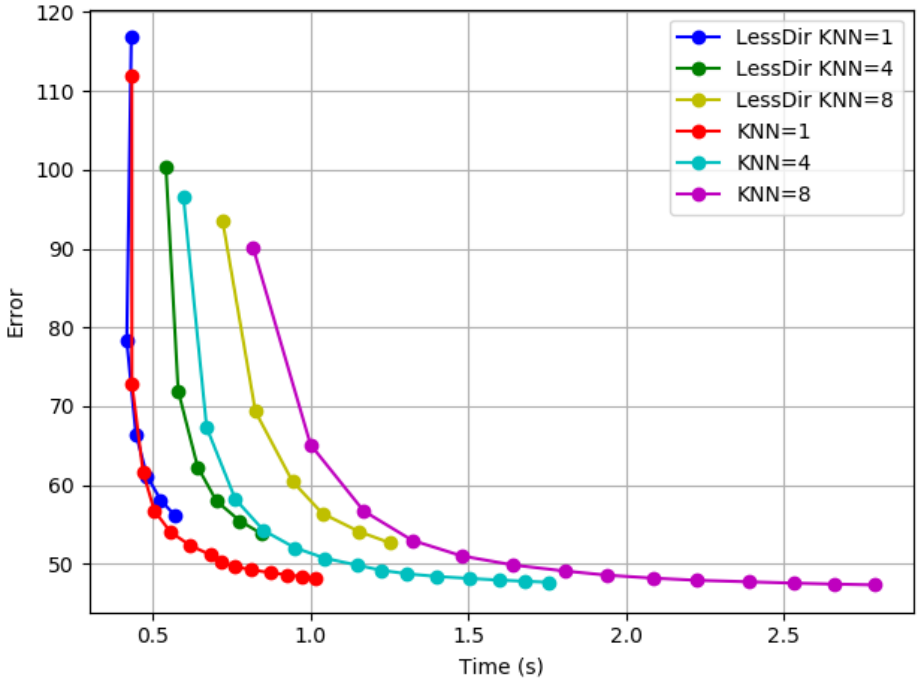


Figure 13: Error/Time tradeoffs of CSH. LessDir KNN is a method that propagates in less directions when it comes to creating candidates. KNN is the baseline approach.

This experiment is to investigate the effect of reducing the amount of type 2 candidates created. Type 2 candidates utilize the coherency of the image to propagate good matches to neighbouring patches. LessDir KNN is a variation of CSH that propagates in the right and down directions. Figure 13 shows that the run-time is lower for similar iterations. This comes at a rather large hit to accuracy.

It should be noted that the accuracy in iteration 1 is lower for LessDir KNN. Good candidates for iteration 1 are generated via local sensitivity, which is unchanged or via propagation from good matches found via random initialization. Furthermore the trends that can be seen shows that LessDir starts out with a better run-time/accuracy ratio, but eventually loses that edge. This is a clear indication that the coherency aspect of CSH works.

7.5 Hardware utilization

	Total ms	Propagation ms	Total GB/s	Propagation GB/s
KNN=8 I=1	613304	292783	84.6	133.6
KNN=8 I=5	1708588	1316262	123.3	147.9
KNN=8 I=10	2974319	2482298	137.9	156.2
KNN=1 I=1	266940	43403	67.3	126.4
KNN=1 I=5	503598	203377	81.5	135.9
KNN=1 I=10	791669	396932	88.3	137.3

Table 5: KNN is the number of nearest neighbours found, I is the number of iterations in the propagation stage. The first two columns indicate the amount of time it took the opencl version to execute the entire program and all iterations of the propagation stage respectively given in ms. The last two columns indicate the amount of memory bandwidth usage of the GPU. The GPU used has a maximum bandwidth of 616 GB/s

This experiment is to investigate how well the baseline implementation of data-parallel CSH utilizes the underlying hardware. CSH is a memory bound algorithm, the amount of operations per memory access is low and the nature of the propagation step prevents temporal locality from being of much use. As such I am primarily interested in the utilization of the memory bandwidth for the entire algorithm and for the propagation step.

To find this I am running the program on a single pair of images of size 1920x800 and using the inbuilt profiler to obtain the exact run-times. This does not cause a loss of generality as it is expected that best and worst case run-time is close.

As can be seen in table 5 the utilization of the memory is not ideal. For a large amount of iterations the propagation stage is the largest contributor to the run time. As increasing the number of iterations is the primary source of increased error this part has been the focus of optimizations.

The bottleneck and the primary source of global memory accesses in the propagation stage is the part where the distance is computed. All versions used does not utilize intra-group parallelism for it. This means that the distance between a patch and a candidate patch is computed by a single thread. However candidate patches are not expected to be located consecutively in memory, which means threads in the same warp are not accessing memory in a coalesced fashion.

The intent of the Futhark compiler is to generate multiple versions of the same kernels, where each version is a different way to utilize the parallelism of the kernel. This is then combined with an autotuner to figure out which version works better when given an input of a specific size. Currently this does not use the intra-group version that is desired.

Isolating the function that computes distance has been done and Futhark is capable of generating and picking an intra-group version. This version achieves the expected speedup.

The critical code is:

```
1 let candidatesl2 = map2 (\x ys ->
2     map (\y -> dist2 proj_s[x,:]
3         proj_t[y,:]
4         ) ys
5     ) (iota patch_count) candidates
```

This code snippet assumes the existence of a projection for each image named proj_s and proj_t, the number of patches and the list of candidates. When isolated and given input of appropriate size the performance seen in table 6. This kernel is run once per iteration and is the source of the majority of the cost of propagation. The exact percentage is not feasible to discover as Futhark fuses the different parts of propagation into a single kernel. Futhark, and the languages it

compiles to, can't accurately profile the parts of a fused kernel. Preventing Futhark from fusing the kernel lets us reason about this relation, but in this case the combined cost of the unfused kernels is far greater than the cost of the fused kernel, which makes it a poor option.

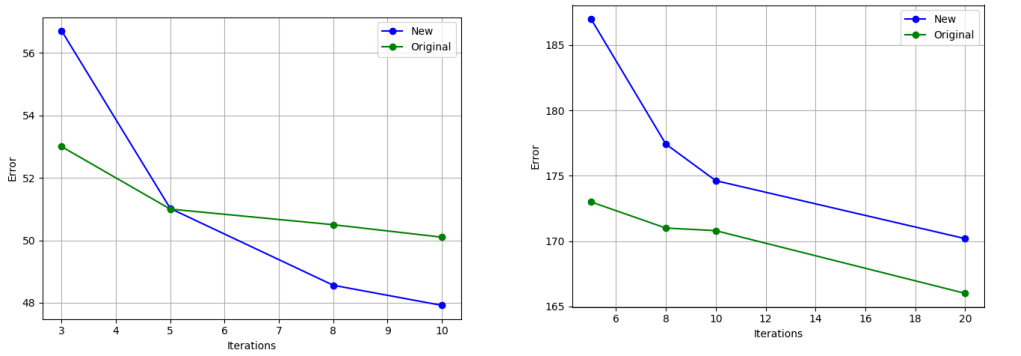
Inspecting the generated opencl code shows that the final program does not make use of the intra-group version, but rather corresponds to the version without autotune that is indicated in table 6.

With Autotune	Without Autotune
23ms	45ms

Table 6: The effect of autotune on a performance critical kernel. Autotune does not work on this kernel when used as part of the full program and is the reason why the bandwidth usage is poor. This kernel was run with input appropriate for a 1280x800 image using KNN=1. It was generated and run using opencl.

7.6 Comparison with CPU-CSH

This subsection compares the accuracy of parallel CSH to the implementation of CSH reported by Oancea et al. [13]. This paper is used instead of the original implementation by Korman and Avidan as the original implementation used downsizing and sampling often for experimental results, which makes direct comparison difficult.



(a) Patch size is 8x8. This was ran on the full 133 pairs of the VidPairs dataset.

(b) Patch size is 16x16. This was ran on the first 25 pairs of the VidPairs dataset using the alphabetical increasing order.

Figure 14: Error/Iteration tradeoffs of CSH. New is the data-parallel version described and implemented in this paper, while original is the CPU version it was based on. Both methods use KNN=8

The new data-parallel and the original CPU implementation of CSH is compared in figure 14. The results between the two are quite close in performance. The early advantage for the sequential version is presumably a difference in initialization, while the later difference comes from the propagation step. This is a bit odd however as the changes made to the baseline version has been focused on making it parallel or are pure run-time optimizations and so can't explain an increase in accuracy. However as has been discussed in the data-parallel section there are differences in how it was created, which may have had an impact on the quality of the candidates.

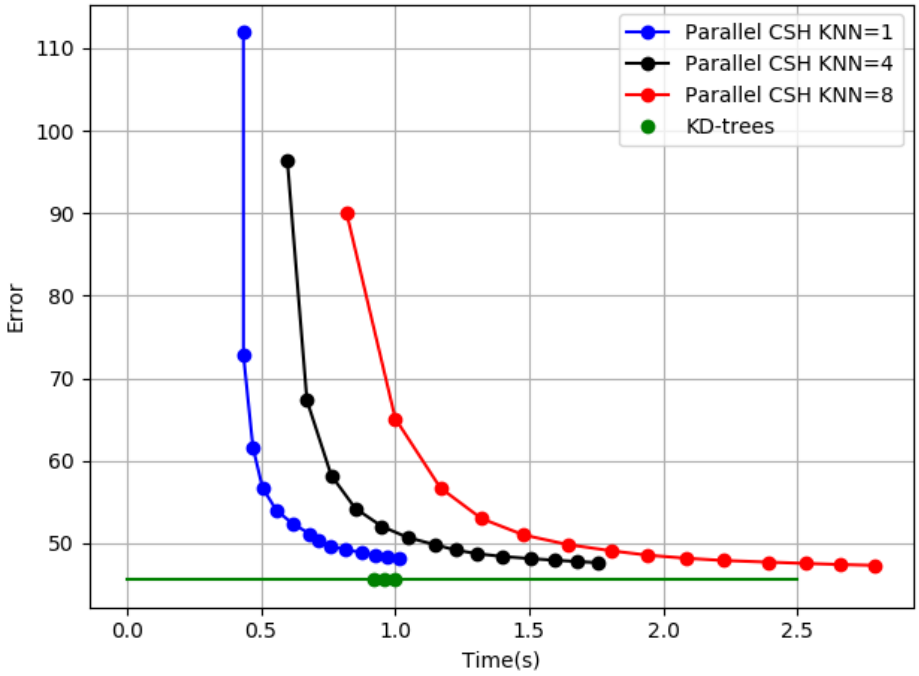


Figure 15: Error/Time tradeoffs of CSH and KD-trees when tested on the 133 image pairs of the VidPairs dataset. Both methods use reduced dimensionality of 23 and patch size of 8x8. CSH markers corresponds to iterations 1 through 14. KD-tree uses KNN=1,4,8 with higher KNN corresponding to higher run-time. The green line is the accuracy of KD-tree with KNN=8.

7.7 Comparison with KD-tree

Parallel KD-trees has previously been noted to have better accuracy and run-time than sequential CSH [13]. Predictably the conversion from sequential CSH to parallel CSH has not changed the accuracy. CSH is quicker than KD-trees when both the number of iterations and KNN is low, but the accuracy is significantly worse within this region. The accuracy of CSH approaches that of KD-trees when the run-time is more than 3 times greater than that of KD-trees. Previously in this section I have mentioned that the hardware utilization of CSH is around 25%, which suggests that a fully optimized CSH may have accuracy and run-time comparable to that of KD-trees.

8 Improvements and issues

8.1 Possible Improvements

- No pruning of candidates is performed. Two variants of pruning were shown in the experimental section. Other variants, such as random sampling, may perform better.
- The initial set of matches are determined through random means. As CSH uses coherency to propagate good matches it may be worth spending computational power to create an initial set of matches that rank well. This may be done by computing an exact set of matches for a subset of the patches.

- Creating the hash codes by sorting the full set of patches. Using a set of representative instead may reduce the time needed for indexing.
- The hash tables are created using the first two patches for each index. Swapping to random patches may improve accuracy by increasing the diversity of the candidates.
- Various values relating to the setup of hash codes are currently determined offline. Creating a variant or function that allows for rapid prototyping of these would allow for easy exploration of parts of the trade-off space that has not been visited in this thesis.

8.2 Issues

- Computation of the distance to candidates is done without intra-group parallelism. This prevents coalesced access and is a large run-time issue.
- Duplicate candidates are allowed, which increases the number of operations and memory access without any tangible benefit.

9 Conclusion

In this thesis I have implemented a massively parallel version of Coherency Sensitive Hashing for computing Approximate Nearest Neighbour Fields. This version has been implemented such that it makes good use of the underlying hardware, although some issues with the chosen language has been encountered, which prevents said version from running. I have explored trade-offs relating to CSH exploring the impact of iteration count, K nearest neighbour and two variants of pruning. I have compared parallel CSH to Propagation-Assisted KD trees, which shows that Parallel CSH is a competitor in terms of run-time and is close in terms of accuracy.

A Glossary

Name	Description	Type
Image	Images are represented in 2D format with 3 channels per pixel.	[n][m][3]u8.
Field	A field is a square of pXp pixels from an image. It is identified by the upper leftmost pixel. All pixels have a corresponding field unless their field would extend outside the bottom or right edge. The number of fields in an image is P_n	[p][p]u8
Patch	Patch is the index of a field. Patch is used to signal that two pieces of information refer to the same field.	i32
P_n	The number of patches in an image. Value depends on the image and the patch size p. P_n = (n+1-p)(m+1-p). This value is useful as we typically work on the flattened set of patches.	i32
WH Kernel	A matrix that can be applied to a field to create a transformation. Is a 2D matrix unless otherwise specified and has the same size as the field.	[p][p]i32
Projections	The result of applying selected WH kernels to the image. As not all WH kernels are used, this achieves reduced dimensionality.	[P_n][r]f32
Hashes or hash codes	Each patch is assigned a number of hash code. The amount of hash codes depends on the number of iterations i in the propagation stage	[i][P_n]i32
Hash table	Contains k representatives for each unique hash code. The number of unique hash codes is determined offline and for a field size of 8x8 is 2^{18}. The hash codes are changed every iteration i of the propagation stage, which also applies to the hash tables. There exists a hash table for each of the two input images.	[i][P_n][k]i32
Source	The image that contains the fields we want to find nearest neighbours for.	
Target	The image that we look in for fields that are nearest neighbours to the fields in the source image.	
Feature	One of the values of a patch. E.g with a patch size of 8x8 a patch is comprised of 64 pixels with 3 channels, which makes a total of 192 features. These patches are later projected onto WH kernels, which reduce them down to 23 features.	
Match	A match is the (approximate) nearest neighbour field in the target image for a patch in the source image.	i32
Candidate	A candidate is a patch in the target image, which is under consideration as a possible match. Candidates are ranked compared to matches and either replace a match or are discarded.	i32

Table 7: The description of various terms used as part of CSH. If applicable the size and type is shown

References

[1] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, November 1998. 1

[2] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. *ACM Trans. Graph.*, 28(3), July 2009. 1, 4, 5

[3] G. Ben-Artzi, H. Hel-Or, and Y. Hel-Or. The gray-code filter kernels. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(3):382–393, 2007. 9, 10, 11

[4] 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, page 253–262, New York, NY, USA, 2004. Association for Computing Machinery. 1, 3

[5] Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. Buffer k-d trees: Processing massive nearest neighbor queries on gpus. *Proceedings of the International Conference on Machine Learning, ICML*, 1:172–180, 01 2014. 1

[6] Fabian Gieseke, Sabina Rosca, Troels Henriksen, Jan Verbesselt, and Cosmin E. Oancea. Massively-parallel change detection for satellite time series data with missing values. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 385–396, 2020. 2

[7] Kaiming He and Jian Sun. Computing nearest-neighbor fields via propagation-assisted kd-trees. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 111–118, 2012. 2, 5

[8] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. Compiling generalized histograms for gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. 6

[9] 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, pages 53–67, New York, NY, USA, 2019. ACM. 6

[10] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604–613, New York, NY, USA, 1998. Association for Computing Machinery. 9

[11] Nikos Komodakis and Georgios Tziritas. Image completion using efficient belief propagation via priority scheduling and dynamic pruning. *IEEE Transactions on Image Processing*, 16(11):2649–2661, 2007. 1

[12] S. Korman and S. Avidan. Coherency sensitive hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(6):1099–1112, 2016. 2, 9, 12, 13, 27, 28

[13] Cosmin Oancea, Ties Robroek, and Fabian Gieseke. Approximate nearest-neighbour fields via massively-parallel propagation-assisted k-d trees. pages 5172–5181, 12 2020. 2, 5, 28, 32, 33

[14] Jia Pan and Dinesh Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, page 211–220, New York, NY, USA, 2011. Association for Computing Machinery. 3