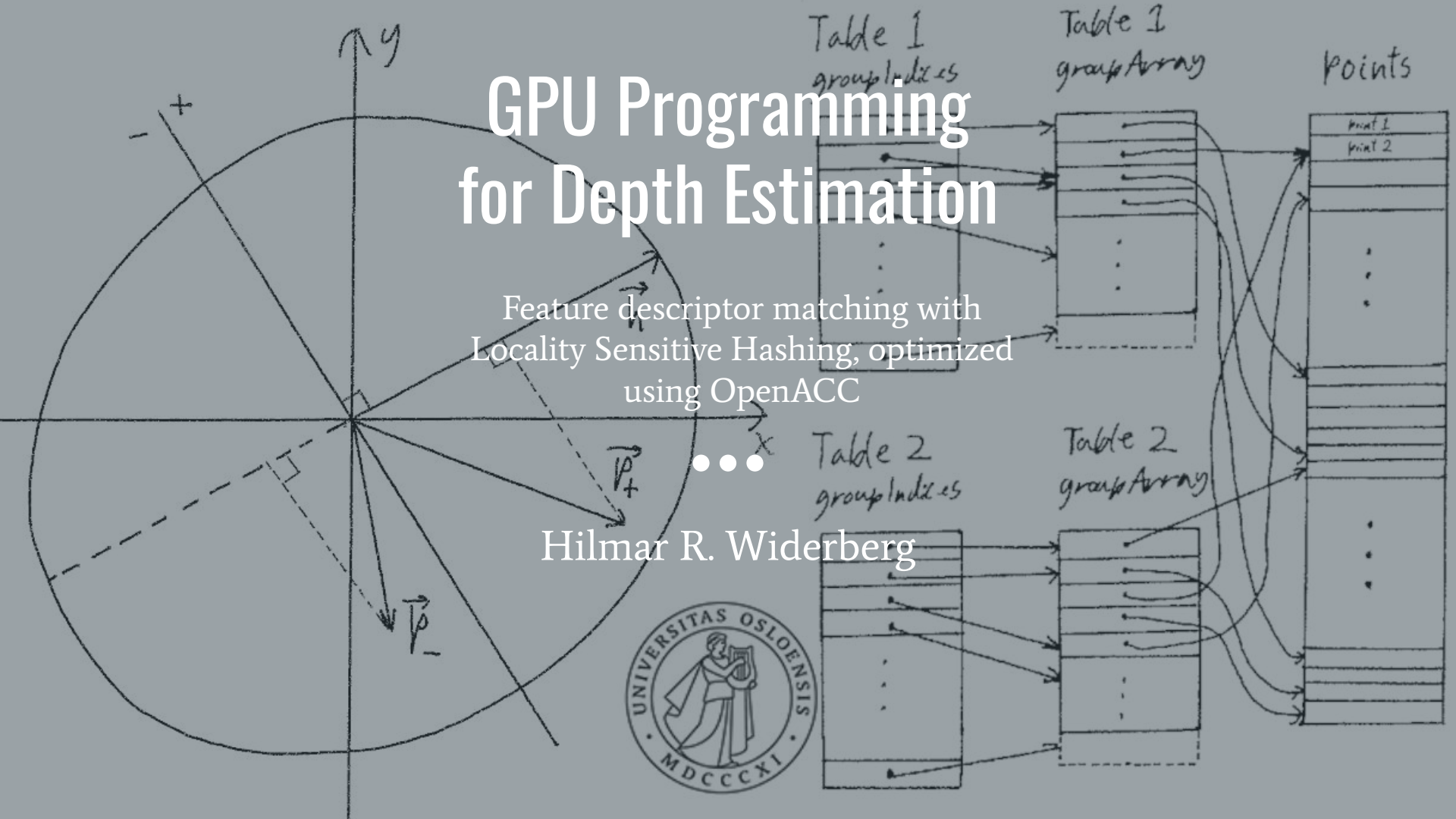
GPU Programming for Depth Estimation

Feature descriptor matching with Locality Sensitive Hashing, optimized using OpenACC

•••

Hilmar R. Widerberg



Contents

SIFT

Paper “Distinctive Image Features from Scale-Invariant Keypoints” published by David Lowe in 2004.

Used for Computer vision.

SIFT is divided into two main stages:

- Finding points of interest in an image
- Represent these points in a compact format

How SIFT finds feature descriptors is covered in the thesis.

For this thesis: Main takeaway are the feature descriptors.

SIFT feature descriptors

Describe points of interest in image.

Invariant to changes in translation, rotation, and scaling.
Partially invariant to minor geometrical distortion and changes in illumination.

Recognizing features across images is key to depth estimation.

Feature descriptors are arrays with 128 dimensional elements.

Finding the best match in a large dataset using exhaustive search is computationally intensive.

Goal is to use LSH to improve time complexity and optimize using OpenACC.

OpenACC

Consists of a set of directives and clauses.

Used to optimize programs using acceleration devices (such as GPUs).

OpenACC is

- Easy to use
- Easy to learn
- Maintainable (Programs remain easy to modify / maintain)
- Portable (Possible to use a single code base for all platforms)

OpenACC

Two main ways of programming: *kernels* construct and *parallel* construct

- *kernels* construct: more work for the compiler,
  less responsibility on the programmer
- *parallel* construct: More direct control

Both constructs can be used within the same program.

I chose to use the *parallel* construct.

OpenACC: *kernels* construct

Programmer defines a region where parallelization may be possible using the *kernels* directive.

Compiler chooses whether to optimize program.

Optimizes program if considered “safe”.

Takes optimization hints from programmer.

May automatically do extra optimizations such as data movement.

OpenACC: *parallel* construct

Programmer specifies how program should be parallelized (in a high level way, no low level details). Compiler obeys.

The *parallel* directive indicates the start of a parallel region (where computation may be moved to the GPU)

The *loop* directive indicates a parallelizable loop.

The *parallel* and *loop* directives are often combined:

```
#pragma acc parallel loop
for (int i = 0; i < nPoints2; i++) {
```

OpenACC: *reduction clause*

Loops may be parallelizable even if iterations are not completely independent

A reduce operation is a typical form of combining the results of semi-independent loop iterations. Allowing them to be executed in parallel.

Typical reduce operations: + , * , | , &

Usage:

reduction(*operator*, *variable*)

```
// calculate hash value of the i'th point,
// store resut in indexGroupMap
#pragma acc loop reduction(|:hashcode)
for (int j = 0; j < nPlanes; j++) {
    float* hplane = &hyperplanes[j * vectorLength];
    // calculate point * hplane
    float vecMul = 0;
    #pragma acc loop reduction(+:vecMul) seq
    for (int k = 0; k < vectorLength; k++) {
        vecMul += point[k] * hplane[k];
    }
    // set i'th bit to one if point is on
    // the "positive" side of hyperplane
    if (vecMul > 0) {
        hashcode = hashcode | (1 << j);
    }
}
indexGroupMap[i] = hashcode;  // save the hashcode
```

```
//diff = sum((points2[i][:]-points1[idx][:]).^2)
float diff = 0;
#pragma acc loop reduction(+:diff)
for (int k = 0; k < vectorLength; k++) {
    float tmp = points2[i * vectorLength + k]
              - points1[idx * vectorLength + k];
    diff += tmp * tmp;
}
```

OpenACC: memory model

In OpenACC, the CPU is called the host, and the acceleration device (GPU) is called the device.

Data is split between host and device.

Data should be considered as existing exclusively on either the host or the device. (Not operated on by both at the same time)

OpenACC: data directive

Automatic copying of data is done inefficiently. (Particularly when using the *parallel* directive)

Data can be explicitly moved between host and device using data directives.

One is unlikely to see performance improvements without data directives.

OpenACC: data directive clauses

Clauses:

- copy - Copy memory between host and device.
- copyin - Copy memory from host to device.
- copyout - Copy memory from device to host.
- create - Reserve memory on the device, but do not copy it between host and device.
- present - Indicate that the memory is already present on the device.

Example of using data directives (present-or-copy is used instead of copy):

```
#pragma acc data \
  pcopyin(points[nPoints*vectorLength]) \
  pcopyin(hyperplanes[numTables*nPlanes*vectorLength]) \
  pcopyout(indexGroupMap[numTables*nPoints])
 {
```

OpenACC programming model

OpenACC splits computation into 3 granularity levels: gangs, workers, and vectors.

Gangs consist of workers, workers operate on vectors.

In CUDA terminology:

- Gangs are blocks.
- Workers are groups of threads working together.
- Vectors are the threads themselves *(kind of, vectors are technically viewed as vectorized instructions, with workers executing in a SIMD fashion, but this is how it is from the perspective of CUDA)

OpenACC loop optimization

One can specify whether computation should be partitioned among gangs, workers, vectors or a combination of them.

gang, *worker*, and, *vector* clauses are used to specify this

seq clause - indicates sequential computation of loop

One can specify the size of each level of granularity using the *num_gangs, num_workers,* and *vector_length* clauses.

```
#pragma acc parallel loop gang worker \
    num_workers(1) vector_length(32)
for (int i = 0; i < nPoints2; i++) {
```

Nearest neighbor

Exhaustive search takes linear time per query point.

In one dimension the search can be shortened using binary search.

In a handful of dimensions K-d trees may be used
(but it breaks down with high dimensionality)

Best-bin-first extends the number of dimensions possible in K-d trees. (Though it is an approximate NN algorithm.

128 dimensions are still a lot of dimensions.

For this thesis, LSH is used.

Locality Sensitive Hashing (LSH)

Came out in the paper “Approximate nearest neighbors: Towards removing the curse of dimensionality“ by Piotr Indyk and Rajeev Motwani in 1998.

Calculates a hash value for a point based on its position in space.
(As its name suggests)

Points close to each other have increased chance of being grouped together.

Hashing is done using hyperplanes, each dividing search space in two.

Idea being that points close to each other are more likely to fall on the same side of a hyperplane.

Hyperplane representation

A hyperplane is represented by a vector orthogonal to itself.

Which side of a hyperplane a point falls on is found by the point's projection onto the hyperplane. (Checking the sign of the projection)

Using the projection, one can also easily find the distance from the hyperplane.

Points hashed using hyperplanes by taking the projection of the point onto hyperplanes,

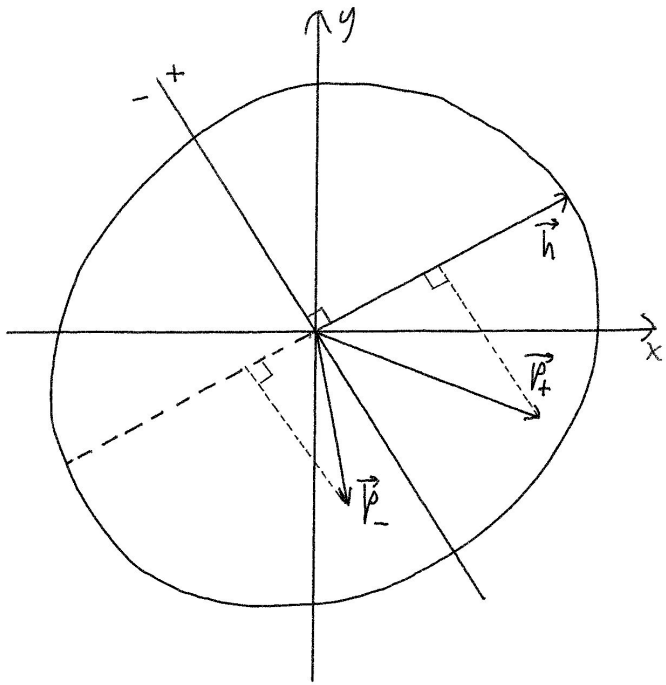
then checking the sign.

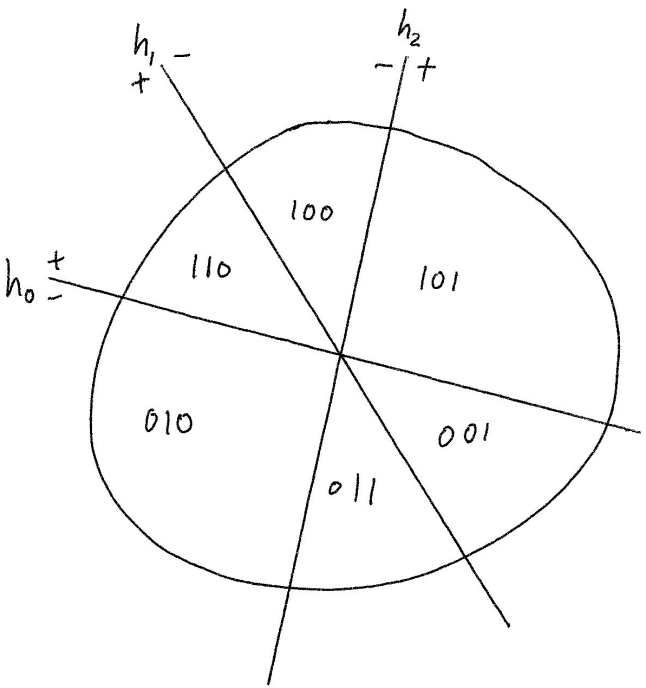
(Can also find distance)

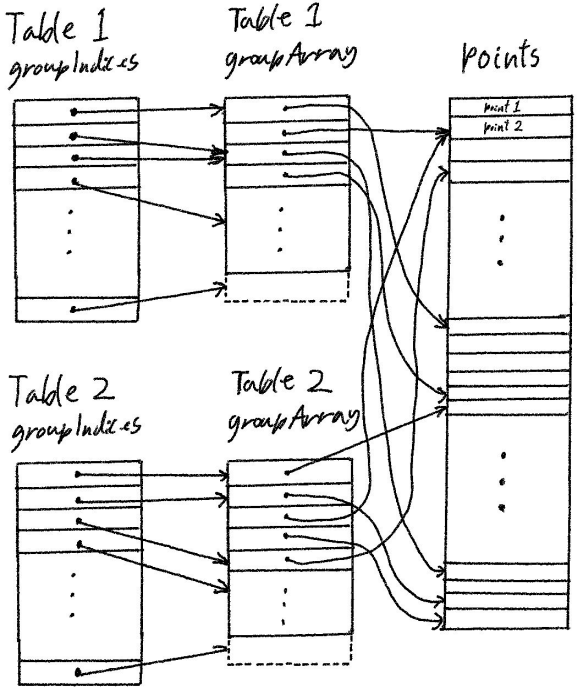
Which side a point falls on can be represented by a 1 or 0.

These values are combined into a single hash value by bitshifting and bitwise-or'ing.

The points are organized into LSH tables according to their hash code.







Good hyperplanes

We want that:

- The hyperplanes together split the search space into roughly equal pieces. (similar number of base points in each box)
- Points have a large distance from the hyperplanes.
- Different LSH tables divide the search space in substantially different ways.

SIFT feature descriptors consist of only positive elements.

Hyperplanes should therefore have a good mix of positive and negative elements.

I chose to have the elements of the vector representing the hyperplanes sum to zero.

LSH optimization

There are roughly five major stages to LSH: *(As implemented by me)

1. Calculating hash values for base vectors
2. Constructing LSH tables
3. Calculating hash values for query vectors
4. Finding potential matches (indices of base points located in the boxes that a query point is hashed to).
5. Matching potential matches (performing raw point comparisons)

All parts are parallelizable in different ways (particularly using more coarse grained forms parallelization).

Parts that are most easy to parallelize using GPUs are:

- Hashing (which consists largely of vector multiplication + some logic)
- Matching potential matches (which consists mainly of vector comparisons + some logic)

The rest are more difficult to parallelize using GPUs.

Multi-table LSH optimization stages

Optimization using OpenACC is split into the following stages:

1. Move computation from CPU to GPU.

2. Optimize data locality (add data directives for effective memory movement.)

3. Optimize loops (explicitly delegate work among *gangs*, *workers*, and *vectors*)

4. For hashing: Optimize data locality further by reducing the number of memory movement phases.

Multi-table LSH results

Parameters: 16 hyperplanes, 32 LSH tables

Nr. of query points: 10 000

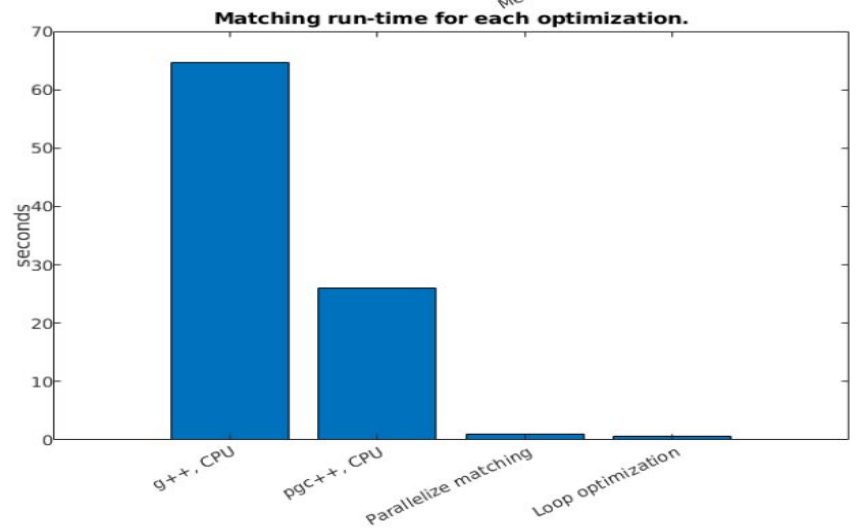
Nr. of base points: 1 million

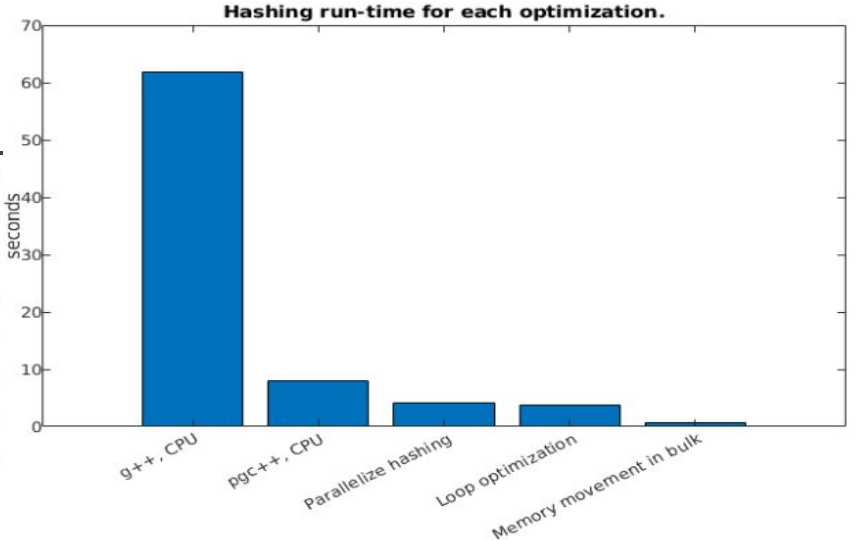
Potential matches found	239775018
Comparisons per query vector	23977.5018
Average portion of search space searched	2.3978%
Correct classification ratio	83.03%

Table 5.1: Details (same for all optimization levels presented)

	Total time	Hashing	Finding matches
CPU program, g++	129s	61.9s	64.6s
CPU program, pgc++	36.2s	7.93s	26.0s
Parallelize hashing	33.9s	4.19s	26.3s
Parallelize matching	7.39s	4.11s	1.01s
Loop optimization	6.65s	3.68	0.584
Moving memory in bulk	3.46s	0.726	0.574

Table 5.9: Run-time results





Single-table LSH

One can increase accuracy without using more tables by checking several boxes per table.

What boxes to check?

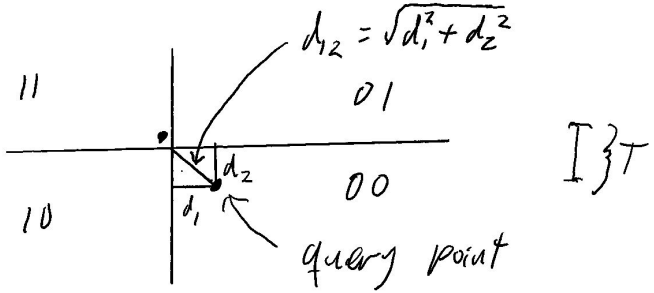
One can calculate distance from points to hyperplanes.

When using orthogonal hyperplanes, one can easily find the distance to the intersection of those hyperplanes using the euclidean theorem.

We check on the other side of all combinations of hyperplanes with a combined squared distance lower than the threshold T^2.

This should theoretically guarantee finding points less than T units away.

Distances to hyperplanes are generally shorter than distances to points, so instead of using this as an exact nearest neighbor algorithm, I use it as an approximate nearest neighbor algorithm by setting a low threshold.

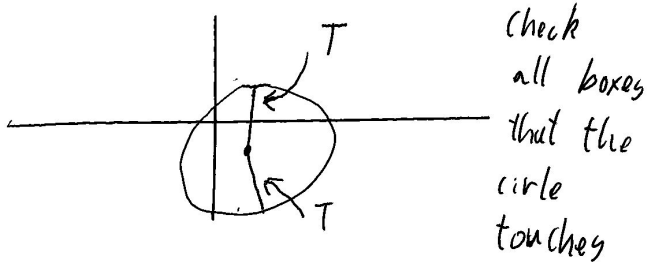


Check 00

if $d_1^2 < T^2$, check 10

if $d_2^2 < T^2$, check 01

if $d_{12}^2 < T^2$, check 11

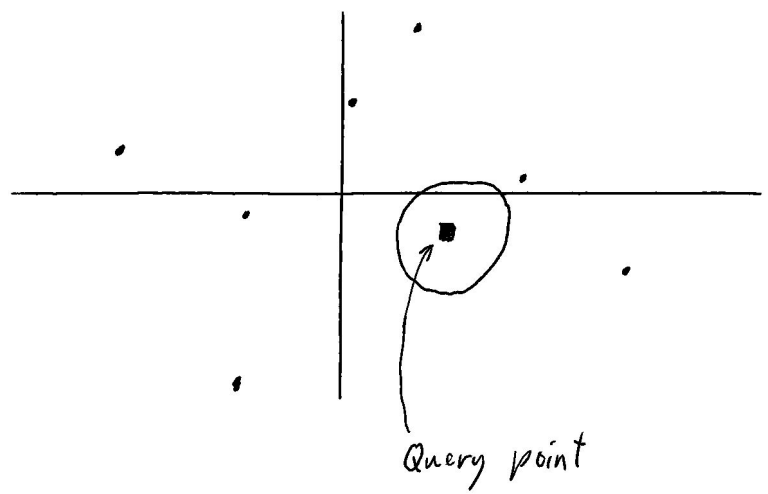


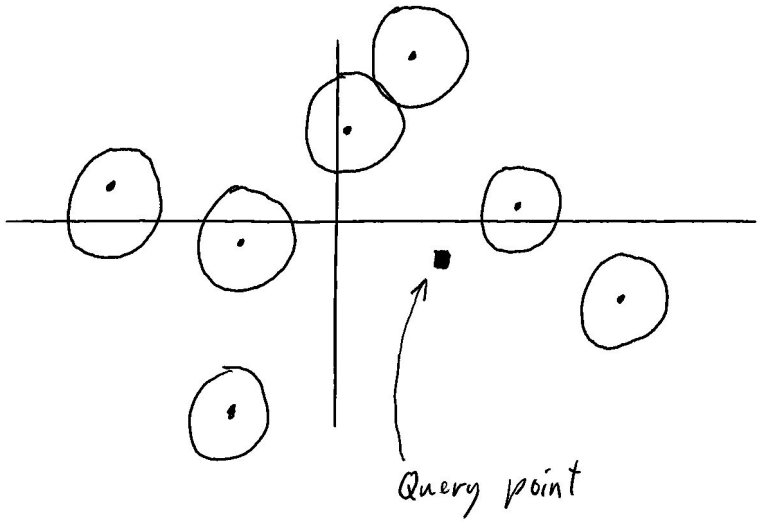
Single-table LSH: alternative

Instead of checking multiple boxes during query phase, one can replicate base points across boxes in the LSH table.

This can quickly become impractical for large thresholds to to high memory requirements.

I therefore used the previous version.





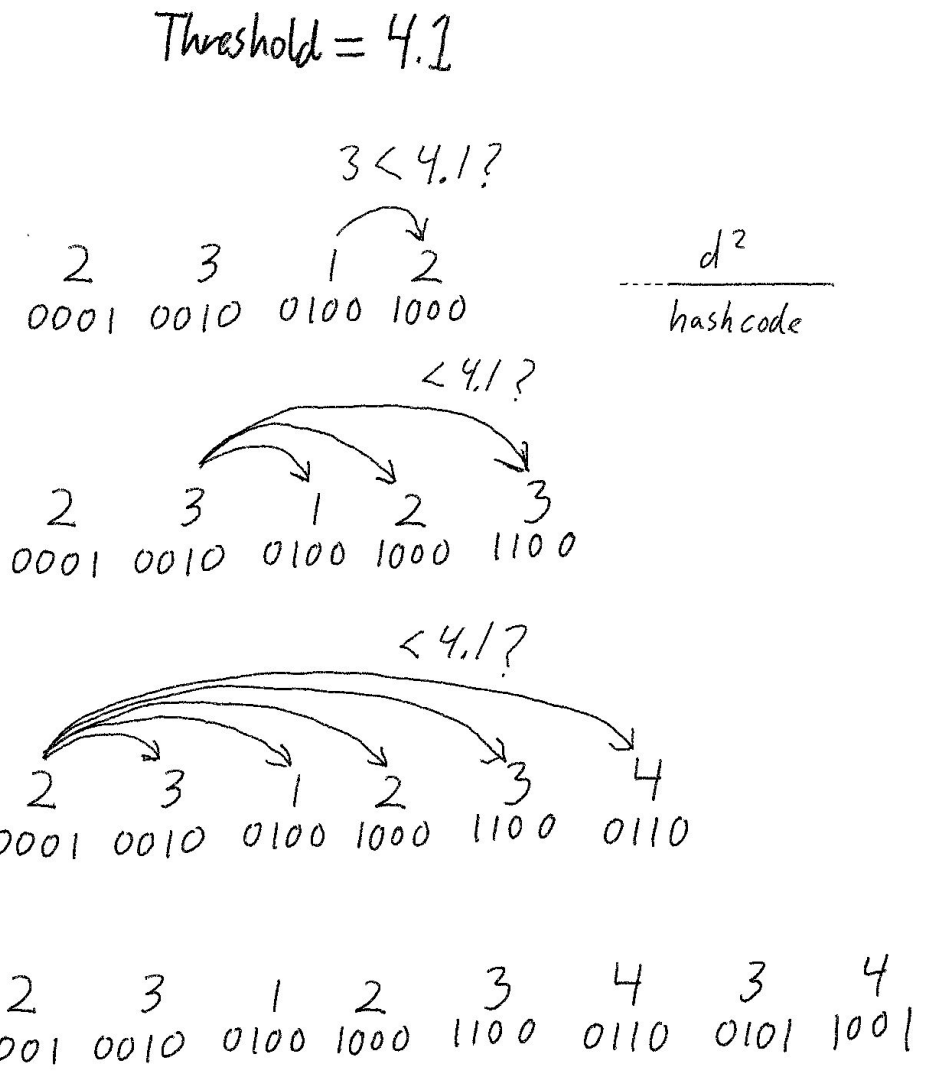
Accumulating squared distances and finding nearby boxes

Define a threshold.

Check for combinations of squared distances below threshold.

If below threshold, combine into new element by adding squared distances and or'ing hashcodes, then add the new element to the end of the list.

Every element in the original list compares itself with all elements after it in the list.



Single-table LSH optimization stages

Roughly the same optimization stages as multi-table LSH, except:

- Hash values are calculated for only one table
- Distances between query points and hyperplanes must be stored.
- Process for finding potential matches is very different

Functions to optimize are still the same. (Even though finding potential matches might be more parallelizable than earlier it is not optimized here)

Single-table LSH results

Parameters: 16 hyperplanes, threshold = 35 (~37% of the average feature vector length)

Nr of query points: 10 000

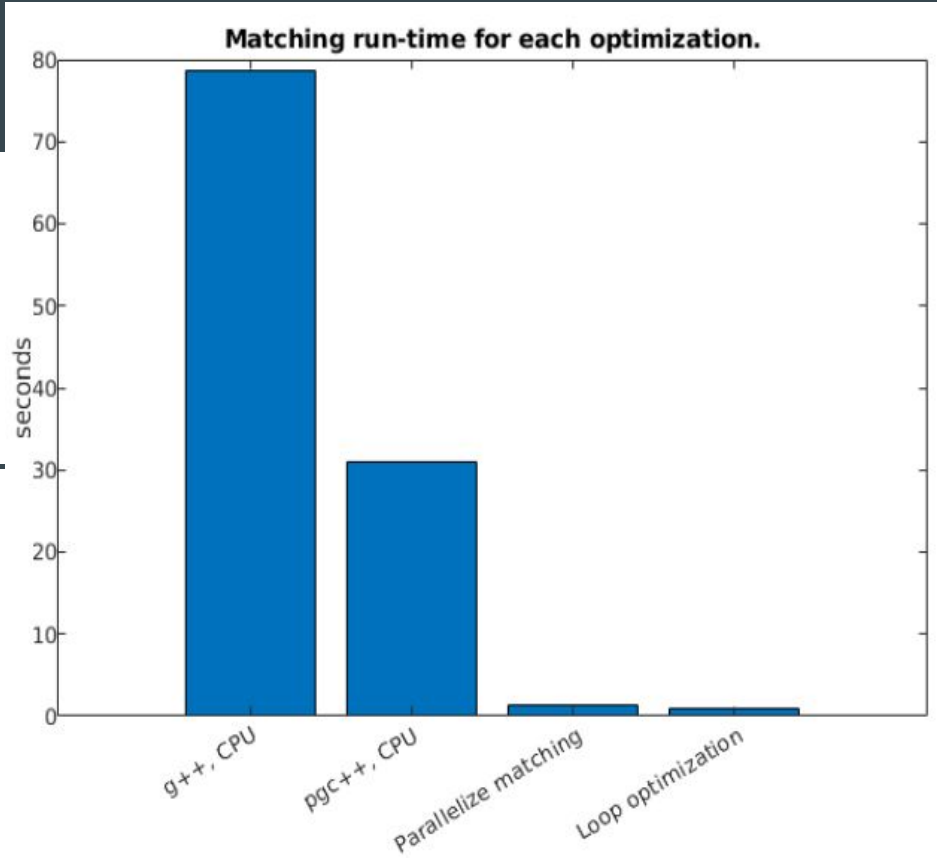
Nr of base points: 1 million

Potential matches found	272585646
Comparisons per query vector	27258.5646
Average portion of search space searched	2.7259%
Correct classification ratio	83.36%

Table 7.1: Details (same for all optimization levels presented)

	Total time	Hashing base points	Matching
CPU program, g++	81.9s	2.25s	78.6s
CPU program, pgc++	33.5s	0.257s	30.9s
Parallelize	2.67s	0.304s	1.31s
Loop optimization	2.44s	0.283s	1.01s

Table 7.8: Run-time results for each optimization stage.



Performance comparison

(LSH: Using the time for all major phases combined)

Performance of ...

Multi-table LSH
(16 hyperplanes, 32 tables)

- Accuracy: ~83.0%
- Time: ~ 3.09s

(Average % of search space searched: 2.40%)

Single-table LSH
(16 hyperplanes, threshold=35)
* hyperplanes improved

- Accuracy: ~83.4%
- Time: ~ 1.96s

(Average % of search space searched: 2.73%)

cuBLAS
(32 bit floats used)

- Accuracy: ~88.3%
- Time: ~ 0.903s

cuBLAS
(16 bit floats used)

- Accuracy: ~87.7%
- Time: ~ 0.188s

Conclusion

In this presentation I have (very briefly) discussed:

1. The problem of matching feature descriptors.
2. Two versions of LSH.
3. OpenACC and how it can be used to optimize LSH.

For the application of matching SIFT feature descriptors these implementations of LSH were not able to compete with the most optimized versions of exhaustive search.

Factors that can be detrimental to LSH:

- Query points far away from their best match
- Absolute best match being necessary
- Points not being distributed evenly around the search space

Applications where approximate best matches are acceptable, or where query points are located closer to their best matches, will benefit more from LSH.

That’s all folks

•••

Thank you for your time and attention