An evaluation of SIFT feature matching strategies for CPU and GPU

Thomas Parmer & Torkil Ravem

Thesis submitted for the degree of
Master in Network and system administration
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

An evaluation of SIFT feature matching strategies for CPU and GPU

Thomas Parmer & Torkil Ravem

Abstract

Computer vision is a research area for developing formulas that enable computers to mimic human visual perception and interpret two-dimensional images through analysis algorithms. Precise and efficient three dimensional(3D) reconstruction is one of the most explored issues in computer vision today. The term 3D reconstruction as it applies to computer vision is the process of creating a 3D model from 2D images. There are many different approaches as well as many different implementations of the respective steps in the process. In this thesis, we specifically look at a Structure from Motion(SfM)-based reconstruction. SfM firstly refers to the task of interpreting the camera motion, commonly defined as the series of possible translations on an image that produce a positional change from some previous reference point. These translations include linear translation, angular rotation, or a combination of these. Secondly, it refers to the use of the motion interpreted data to construct a 3D scene. The motion interpreted data is created through analysis of an image sequence. SfM uses feature extraction first to find unique points seen in several images, from there it goes to meshing, continuing to mesh refinement and texturing. The matching, also called Nearest Neighbour search(NN) is one of the most time-consuming parts of any SfM software, and although there is much research in this field, there is to date no solutions faster than the exhaustive search that has the same precision. There are, however, many solutions using Approximate Nearest Neighbour(ANN) searches that sacrifice matching precision for effectiveness which many applications prefer due to their apparent runtime benefits. We will in this thesis show that there is possible to use a different metric to improve the runtime of the exhaustive search without diminishing the precision of the matches. Further, we will explore different approximate searches to see the benefits they can give before introducing our implementations and results. To begin with, a general introduction to 3D reconstruction will give the needed background to understand the challenges to this task. There will also be an introduction to some programming themes to show what types of tools are available to improve different types of code.

Contents

List of Figures

List of Tables

Preface

We would like to express our great appreciation to our supervisor Carsten for his valuable insight and constructive advice during the planning and development of this thesis. His office door was always open for a discussion, and he always took time to answer our many questions.

Chapter 1

Introduction

3D is a common term in the world today. It refers both to a field of study that utilises 3D technology, or simply something that exists in three-dimensional space. There are many reasons why 3D reconstruction is useful in today's society. A complete 3D model of our world and everything in it will allow us to see and study everything from micro-sized elements in the human body to massive planets in our solar system. There are things we cannot visit in person because they are either in an inaccessible place or it would take too long to get access to it. If we can create 3D models, we may study and learn about objects without even leaving our own home. The field of computer vision enables us to use computing power for tasks requiring an intelligent understanding of images. Common applications consist of object recognition, motion detection and analysis, and 3D reconstruction. An image-based 3D reconstruction is a computed representation of a 3D shape from several images of a scene. The number of images and camera positions needed for reconstruction is arbitrary, but the higher the number of images, the better the quality of the reconstruction. This comes as a result of more images generating a greater amount of data to create and refine the model. The minimum requirement is two images in different positions, as computing the depth between objects is otherwise impossible when going from a 2D to a 3D scene. The images can come from isolated cameras or a video sequence. An efficient pipeline is necessary for the reconstruction to be valuable in many practical applications, and real-time reconstruction is the final goal. To achieve this satisfactorily, it becomes essential to have a comprehensive understanding of 3D reconstruction.

1.1 Purpose of thesis

The goal of the thesis is to explain 3D reconstruction, and how it can be done accurately and efficiently, getting it as close to real-time speed as possible. More precisely, we will describe the steps that lead from feature extraction to point clouds by exploring the pipeline that makes this possible. We will go through some of the most common steps in 3D reconstruction to get a basic understanding of the complexity of such a process. Later we will go deeper into a more specific setup which is the sole purpose of our work. Our primary task in this is to look at and hopefully improve the runtime and efficiency of a project called BUISAR (Building Site Inspection Supported by Augmented Reality). The idea is that the use of 3D reconstruction and Augmented Reality (AR) should allow clients to produce a virtual reference between reality and virtual reality and augment or enhance this reality to assert that their constructions are within the desired specifications. The project is built up by three steps which mainly is 3D reconstruction, match-moving and AR. The goal for both match-moving and AR are real-time performance; if not they are no longer beneficial.

Since this is a relatively large project, we will base our work on the third part of the project which is the 3D reconstruction. To be able to streamline the process we will need to understand how this set-up is working, and what the different parts of it do. There are three primary functions in the approach we are looking at, PopSift for feature extraction, OpenMVG for reconstruction of the mesh and CMPMVS for mesh refinement and texturing.

3D reconstruction pipeline

Figure 1.1 shows the AliceVision 3D reconstruction pipeline. This pipeline is a complete setup from start to end, and it will be the basis of our work. The first step is to provide images of an object or a scene to the software. There has to be at least two images to use the software, and the more information about the camera used regarding position and settings the better.

The next step is to find points of interest in each image. This step will go through every image and extract unique features. Features are a distinctive group of pixels and are used to find points of interest in the images. After the feature extraction, the features are compared to find the images that are pointing to the same area of the scene.

After the initial steps, the pipeline arrives at the feature matching. This step is a vital one who finds the most similar features between the different images, which will be the foundation of the reconstruction. These matched features will then go through the SfM which will fuse all the matches between the image pairs into tracks. Each of these tracks will represent a point in space which is visible from multiple cameras.

The next section will then calculate the depth value of each pixel and store them into a depth map. The depth map is then used to create a mesh, a dense geometric surface representation of the scene. Last part is the texturing which will define specific characteristics as colour, shininess, transparency, reflectivity, and smoothness [55]. This process paints the 3D mesh model with static images.

Our focus in this setup will be the Nearest Neighbor search in the fea-

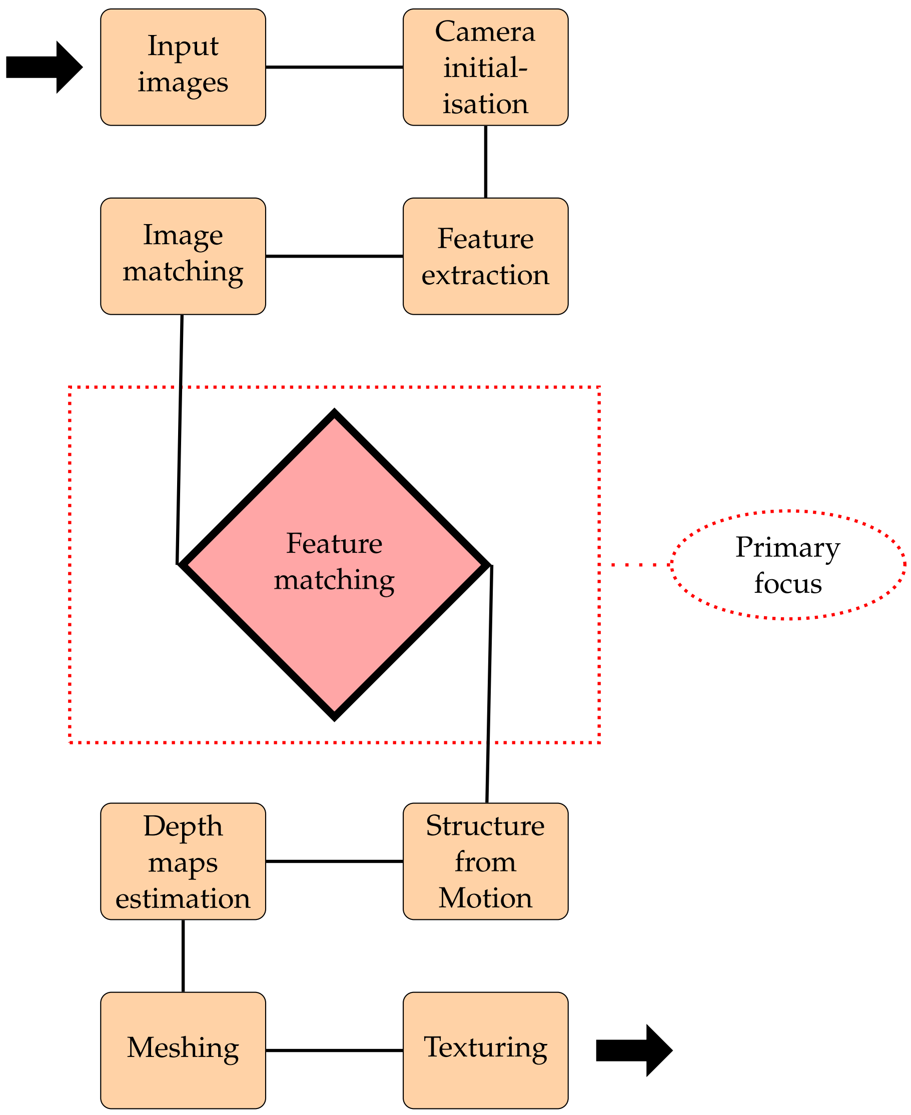

Figure 1.1: AliceVison 3D reconstruction pipeline

ture matching which is a severe bottleneck regarding runtime. This part is highly time-consuming, and if we manage to reduce the time spent on this, we will improve the overall runtime of the program.

1.1.1 Nearest Neighbour search

Algorithmic improvements are substantially more likely to produce comprehensive performance improvements compared to modifying or reworking implementation details. Therefore, our task is first to study the problem as a whole and evaluate the existing algorithms in the context of this problem to determine if a more efficient solution is theoretically possible. Secondly, we will look at the quality of the coded implementation to see if there is room for improvements that will enhance the overall execution performance. This process will consist of reducing complexity, such as eliminating unnecessary abstractions and temporary variables and finding the optimal data structure for the algorithm. We will focus on the matching part of the program also known as the Nearest Neighbour search, which is one of the main bottlenecks in the pipeline. Nearest Neighbour search refers to the process of finding the closest point to the query point from a given set or database of points. Nearest Neighbour search executes after the feature extraction of the input images, where each of them has many descriptors. The concept here is to find the best match between the images, the interest point in one image that has the closest "neighbour" in another image.

There are different approaches to this; one such approach is the exhaustive search method (also known as a full search) which will explore all possible matches and thus guarantees to find the global best match. However, this is a computationally costly process due to many repetitive operations where each descriptor in one image needs to be compared with every descriptor in the other image, and therefore not applicable to more massive datasets. Alternatively, there are a lot of different approximate searches that do this part quicker but cannot guarantee that the matches they find are the correct ones.

OpenMVG has a solution for exhaustive search using Euclidean distance as a metric which will be our primary focus, but it does also have some approximate searches as a k-d tree solution with FLANN (Fast Library for Approximate Nearest Neighbours) and a cascade hashing option. The approximate searches are something that has been explored a lot over the years and is, therefore, more unlikely subject to improvements compared to the exhaustive search. However, we will study the advantages and the drawbacks of the existing ANN algorithms and propose our own ANN algorithm to see what benefits we can achieve. It is interesting to see if we can find a solution to exhaustive search that improves the runtime, as there is to date no known algorithm that does it better without diminishing results regarding the matches.

1.1.2 Goals, achievements

Our initial goal is to explore solutions and changes that will improve the overall performance of the process. At the same time, we want to find sections that we are not capable of improving, and learning why this is so. It is also interesting to see if some of our ideas apply to other problems, which uses the same type of algorithms to achieve their goals.

Our primary focus will be on the matching process and the metrics used on this part, as this is one of the most time-consuming parts of the pipeline. We will explain the concept of this later in our thesis.

Our primary goal is to make the exhaustive search a viable option in comparison to the approximate searches. We will accomplish this by presenting a new metric which is better suited for matching SIFT descriptors than the Euclidean distance. The reason for this is because it is favourable to have an algorithm that does not sacrifice precision for matching speed. The secondary objective is to propose our own approximate search with higher precision than the other ANN algorithms available in OpenMVG while reducing the search time of the exhaustive search.

Chapter 2

Background

2.1 3D reconstruction

Accurate and efficient 3D reconstruction has been a goal in computer vision for decades. There are a great variety of approaches to 3D reconstruction. The best-suited approaches will differ with the captured object or scene, and the required quality of the result. Feature-based 3D reconstruction is typically achieved in three phases [28].

The first phase is to find a set of matching points between the different input images to determinate the common parts of the images. The second phase is then to use the matching points to calculate the location, orientation and other needed parameters of the cameras. The third phase is to compute a 3D mesh using the data from the other phases. This explanation of the process is simple, and there will be more phases in between these depending on the complexity of the captured object or scene, and the quality of the desired result. We will go through some of the tools and methods used in this field today, and explain how they measure up to each other.

2.2 SIFT - Scale Invariant Feature Transform

One vital step in most computer vision technologies is a distinct and reliable representation of points of interest in an image. To accomplish this, we need to provide a suitable algorithm which can identify image features that is recognisable over a wide variety of both geometric and photometric image transformations comparable to those that would occur from a change in viewpoint, illumination and other viewing conditions of a scene. Further, the algorithm must create a unique signature that describes these feature points to identify their appearance compactly and robustly so that they are repeatedly recognisable.

The keypoint detector and descriptor Scale Invariant Feature Transform (SIFT) [40] is an algorithm created to be good at locating (feature point) and labelling (feature descriptor) the same point in the 3-dimensional real world seen in two images. SIFT does this with a Difference of Gaussian

(DoG) based keypoint detector that creates a collection of features that does not change based on scaling, rotation and brightness. These features descriptors are created and labelled based on the gradient orientation distribution in the surrounding region of each feature. These invariant features enable us to recognise similar features precisely from several angles and scales under various lighting conditions to find correspondences across many photographs containing the same scene or object. While the SIFT algorithm has many use cases from object recognition to 3D reconstruction, its core functionality is to extract features and create one or more descriptors for every feature. In our application, we will use it as the first step to determine image correspondence, or simply put, given two images of the same scene, SIFT enables us to find equivalent features in the images before we match them. This matching gives us information about how the images are related to each other. While there are many new keypoint detector and descriptor algorithms available, SIFT is very stable and yields strong results compared to its alternatives [4].

SIFT is often used in applications dealing with object recognition [40, 58], object tracking [21, 71, 73], face recognition [46], face authentication [9, 42], three dimensional reconstruction [45], image stitching [13, 32], and image classification [66, 70]. However, these are only a small sample of possible use cases.

2.2.1 SIFT features

For the SIFT algorithm to be invariant to the noted parameters and able to detect the same feature in two images independently, the feature needs to possess several characteristics. First, a feature needs to be repeatedly recognisable - A feature is viewable from several different angles in such a way that we can recognise the same feature in images taken from different viewpoints despite large-scale change and difference in camera orientation. For a feature to be recognisable over large-scale changes, essentially making it scale invariant, it should be extracted over several scales, ensuring that it is not only present at a particular scale level. In SIFT this is done by using a sub-octave Difference of Gaussian pyramid. Neighbouring scale levels in the pyramid are subtracted from each other to create Difference of Gaussian images. These images are then searched for features that are present in several or all of the DoG images.

A feature also needs to be unique - This means that it should be sufficiently distinct from all other features such that we can compare and match them with ease. In consequence, a good feature will rarely reside within a textureless region as precisely localising the feature point will be challenging if not impossible [62]. A region with large contrast change is a much better option; however, a strong one-directional contrast change is not enough due to the aperture problem [2, 30, 41]. This is because all features from a straight line on a blank background would be identical, i.e. we would not be able to determine an exact position along the line. Considering this, we want features that are distinct from their neighbourhood in both horizontal and vertical direction. A corner will often satisfy this characteristic. Good features should be local, they should not cover a large area, and preferably be a single pixel. This will increase the algorithms ability to handle occlusion, a concept where one object is partially or completely hidden from one image to another due to viewpoint changes. Several algorithms offering good repeatability over a variety of image transformations has been proposed [5, 38, 40, 51, 57]. Among these, SIFT is highly popular. The SIFT detector is partially invariant to affine transformations. While there exist algorithms that are thoroughly invariant to affine transformations like the Hessian-affine, Harris-affine detectors [48, 49], the SIFT detector and descriptor produce similar invariance characteristics [50, 72].

2.2.2 SIFT pipeline

The overall SIFT procedure for recognition has two main components: Feature extraction and feature description. Each component consists of two separate steps. The first two steps are scale-space extrema detection and feature point localisation. These two steps are done by looking at extrema in the Difference of Gaussians and identifying the exact location of the feature point. The last two steps are orientation assignment and feature description, which means that we are going to define an orientation before using this orientation in combination with the feature to create a description.

1. Scale space extrema detection
2. Feature point localization
3. Orientation assignment
4. Feature description

2.2.3 Feature detection

To detect a feature in scale space, we search the complete image in several different scales to find groups of pixels that are significantly different from their neighbouring pixels. To find these positions, we will create several versions of the image, each smoothed with an increasingly stronger Gaussian blur. Even though resolution technically is the same (same number of pixels) after the blur is applied, it will mathematically have the same effect as increasing/decreasing image scale. The visual effect of a Gaussian blur on an image is seen in figure 2.1 on page 10.

We then compute the Difference of Gaussians (DoG) by taking pairs of these and do a pixel-wise subtraction. This new image will heighten the visibility of places in the image that has substantial intensity changes, such as corners and edges. As we are after corners, we look at the surrounding

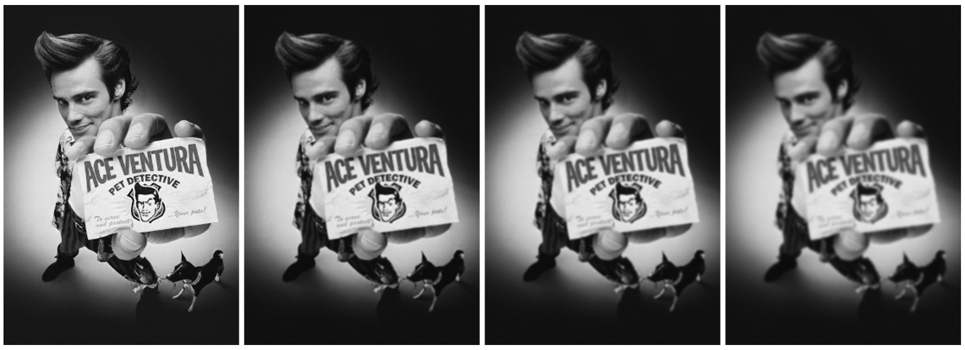

Figure 2.1: An example of Gaussian blurring an image. From left to right: the original image in grayscale, Gaussian filter with $\sigma = 3$, with $\sigma = 5$, with $\sigma = 10$.

neighbourhood of a point to see if our point is an extremum in horizontal and vertical direction. We will know if the extremum is invariant to scale if it is present in all sizes of the DoG image. To qualify as a SIFT feature, it must be an extremum to its 26 neighbours: 8 in same DoG level, 9 in the less blurry level above, and 9 in the level below. To get features that have better stability, we apply a threshold on minimum contrast, as low contrast points are prone to noise errors. Contrast is the absolute DoG value of the extremum. After applying a threshold on minimum contrast, the final features are calculated by an additional threshold on the ratio of principal curvatures. Here the 3D curvature volume through the extremum is computed to help remove features that are unstable to small amounts of noise. The extremum can move by up to a pixel left or right, up or down, DoG level up or DoG level down.

2.2.4 Feature description

Through finding the overall orientation of the feature based on local image properties we can describe it. To calculate the individual pixel orientation, SIFT uses a 16x16 pixel window around the feature in the level of the Gaussian pyramid where the feature was originally detected. The pixel orientation is the direction of largest change in the gradient within a neighbourhood of pixels.

SIFT counts each pixel in a histogram with 36 possible orientations. These 36 different orientations cover 360 degrees. Each pixel is weighted by a Gaussian fall-off function, such that pixels closer to the centre of the feature have greater weight than pixels further away. The overall feature orientation is the highest column in the histogram. See figure 2.2 on page 11 for visualization. Note that the image and its caption are misleading: There are 16 quadratic "quadrants" around a feature point. The number of pixels in each is unknown as it depends on the DoG scale level we find the keypoint, and on rotation. However, these quadrants overlap by 50% in each direction which is significant because it is the basis of SIFT's affine tolerance.

If there are other columns with above 80% of the height of the main peak, these will be created as separate features to improve matching. These new features will be at the same location, but with different orientations. These 256 computed orientations are added to histograms of size 2x2x2 with trilinear interpolation to get a descriptor more robust against a wrong assessment of orientation [62]. After these computations, we get 128 non-negative values representing the SIFT descriptor.

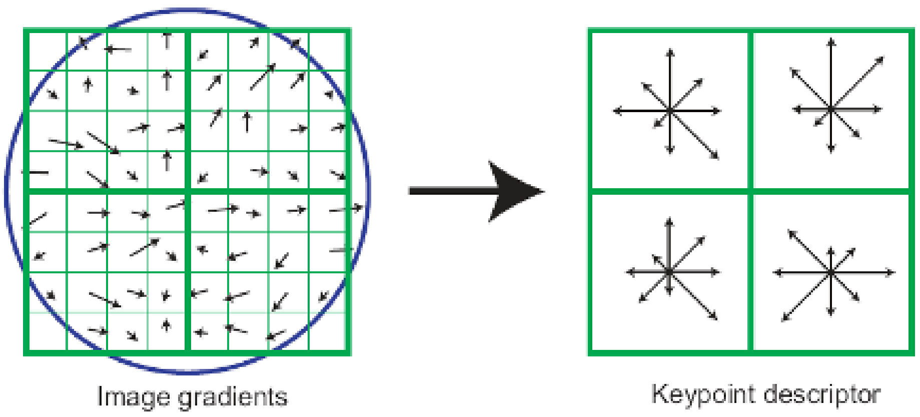

Figure 2.2: Figure from [40]. Example of individual pixel orientation used to calculate a descriptor array. The blue circle represents the Gaussian fall-off function. Note that all sizes are divided by two (8x8 instead of 16x16, 2x2 instead of 4x4) in this example to clearly visualize the orientations.

2.2.5 Descriptor representation

The SIFT descriptors are normalised to unit length with L2 norm to reduce the effect of contrast. Further, it is truncated with 0.2 as the threshold to reduce the effect of other photometric variations on the descriptors stability and then renormalised again to unit length. This way, each of the 128 dimensions in a SIFT descriptor should theoretically contain a value in the interval [0, 1]. However, in practise SIFT descriptor dimensions are in the interval $[0, \frac{1}{2}]$. It is also quite common to convert the floating point vector into a vector of unsigned chars. If each element in the vector is in the interval [0, 1], multiplying by 256 accomplishes this conversion, as this is the maximum size of an unsigned char. This method would lose some precision as each of the 128 dimensions are in the interval $[0, \frac{1}{2}]$. Thus we can instead multiply by 512 to get better distribution with less precision loss. Note that this conversion still loses some precision.

2.3 Stereo Matching

Stereo matching also known as disparity mapping is a method for finding corresponding points from one image to another. It is one of the most common parts of computer vision and has a variety of solutions. Stereo methods consist of several steps, most of them use matching cost computation, cost aggregation, disparity computation/ optimisation and disparity refinement [60]. The sequence of these steps varies from the algorithm used in a specific case (local- and global algorithms).

The most difficult parts are often occlusions, object boundaries and fine structures, which can appear blurred. It could also be challenging due to low or repetitive textures [29]. Stereo matching is one of the most demanding parts in the pipeline regarding speed and efficiency, and there is often a balance between precision and speed.

2.3.1 Matching cost computation

In simple terms, a matching cost computation measures the similarity between different pixels. Typical cost functions are squared difference (SD), absolute difference (AD), mutual information (MI) and normalised cross-correlation (NCC). MI is a method for handling complex radiometric relationships between two images. It relies on the entropy of the images underlying probability densities [19].

2.3.2 Disparity computation/refinement

Disparity originally describes the difference in location of corresponding features when seen by the left and right eyes. Computer vision literature describes this as inverse depth [60]. Disparity computation is a "winner takes all" algorithm, and the corresponding pixel is the most similar pixel in the searched set. For every pixel select the disparity with the lowest cost. Disparity refinement is used for removing peaks, checking consistency, interpolating the gaps or increasing the accuracy by sub-pixel interpolation [29].

2.3.3 Cost aggregation

A problem with disparity computation is that there can be too much noise in the images (or if there are too many similar pixels). The solution is then cost aggregation. Instead of using just one pixel, it uses a "matching window" around the pixel of interest and creates an area-based matching cost. Global algorithms typical skip the cost aggregation step and define a global energy function that includes a data term and a smoothness term [29].

2.4 Multi-view reconstruction

To construct 3D models from multiple photographs is the most common way to get the best result as possible, at the same time the more photos added, the more time consuming and complex the process will be. Multi-view reconstruction usually consists of three steps [23]. The first step is to acquire multiple images from a scene or an object; this can be images taken with a camera or downloaded from the internet. The second step is to define the camera parameters; this is done in advance while capturing the images or calculated later. The last part is to build the 3D model (dense geometry reconstruction).

2.4.1 Image acquisition

Image acquisition is always included in 3D reconstruction and is the basis for the process. There are different ways to acquire images, and they will be implemented concerning the scene we are capturing. There are three typical image acquisition setups [23]: Fixed camera on a rotating object or fixed object and a rotating camera. Often used for small objects in a controlled environment and will make it possible to calibrate cameras from images of a calibration chart. Moving camera around an object or a scene. Used for capturing small-scale outdoor scenes (buildings, people) and will often need to get camera parameters from an algorithm. Downloading images from the internet. This is used for large scenes, there are some drawbacks like less control of the object and light conditions. This setup must get camera parameters from an algorithm.

2.4.2 Camera Parameter Estimation

Camera parameters consist of extrinsic, intrinsic and distortion parameters [23]. Extrinsic is rotational and translational pose information of the camera and will change when the camera is physically moved. Intrinsic is information about pixel sensor size, a principal point and magnification factors. Distortion parameters capture higher-order effects from the lens.

Since the camera is pre-calibrated in our project, and our scene is in a controlled environment, we only need to do a camera pose estimation (extrinsic) and can dismiss the intrinsic and distortion parameters. Thus we save a lot of work and time in the project.

2.4.3 Dense geometry reconstruction

The last part of the reconstruction is to use the images to find common points to build a 3D model. This is done by comparing the images and finding corresponding points that match. The points will then be used to create the complete model.

2.5 Structure from motion (SfM)

The idea of recovering an unknown scene also known as structure from motion is one of the most difficult problems in computer vision. There are many different approaches, and there is a great number of potential solutions. SfM is a concept that takes a set of images as input and outputs a complete 3D reconstruction. It consists of several steps, solved in different ways. The first step is to obtain multiple images of a scene or object. The next step is to extract unique feature points from each image. After this, the images need to be matched against each other, before estimation of the camera poses, and the triangulation of 3D points in the images. A bundle adjustment is then applied before the final reconstruction is ready. In this section, we will describe some of the methods used in a SfM pipeline.

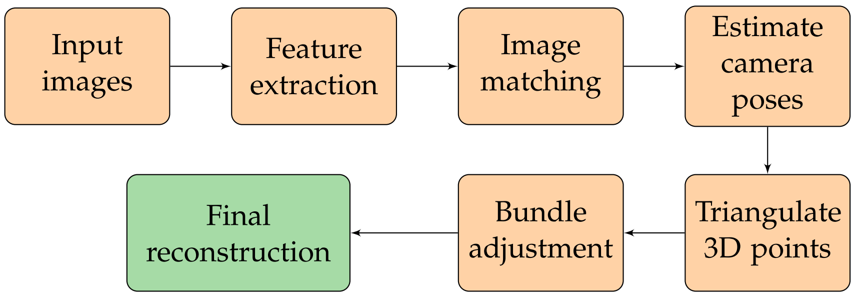

Figure 2.3: Flowchart of SfM

2.5.1 Pinhole camera

Imagine we have a film and we expose it to the light projected from a scene. Because all points in the scene will project to every position on the film, this will not give us an image representation of the scene. The problem is that every location in the film sees the light from more than one location, no single value of light comes from a single place in the scene. The pinhole camera model provides a solution to this problem by putting in a barrier between the scene and the film. This barrier is light proof with a small hole in the centre where rays of light can come through to the film. This hole is called an aperture. This barrier affects that each location on the film can only be reached from one location in the scene through a straight line. All of the other projected light from this location will hit the light proof barrier, and therefore it will not be part of the image. An interesting thing to note here is that the image projection on the film will be an inverted version of the scene. This inversion has a direct correlation with the model. To give an example, if we draw a projection line from a point in the upper right corner of a scene through the aperture, this line will have a downwards right direction. This direction will continue through the aperture. Therefore the point will be projected in the bottom left corner of the film. Most structure from motion techniques starts with adopting a perspective projection model. This model is illustrated in Figure 2.4 on page 15. Here, a 3D point P is projecting on a two-dimensional image plane with perspective rays coming from the Center Of Projection (COP) which also is the origin of the coordinate system. The focal length f is the distance from COP to the image plane along the principal axis.

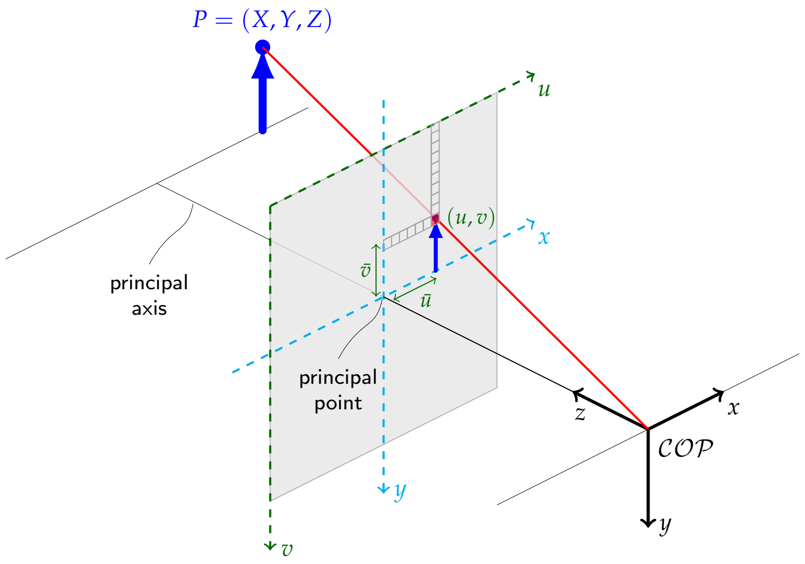

Figure 2.4: camera perspective projection model

2.5.2 Epipolar geometry

The epipolar geometry is the geometry between two views [26]. There are several relations between a real-world point and how it projects in different 2D views that lead to constraints between these views. Given a point in 3D space X that is seen in two views as x in one image and x′ in the other. Then the relation between these two corresponding points is the following: If we draw a line between the two views centres of projection and from view one to x and view two to x′, then these rays would intersect at X. The camera centres, X, x and x′ will also be coplanar. Given we only know the location of x and the camera pose but not x′, we can use epipolar geometry to constrain where x′ lies in the second plane. If we draw the plane that consists of the two centres of projection and x, we know that x′ lies on this plane. More precisely, x′ lies on the line where the plane intersects the second views projection plane. This line is known as the epipolar line corresponding to x [26].

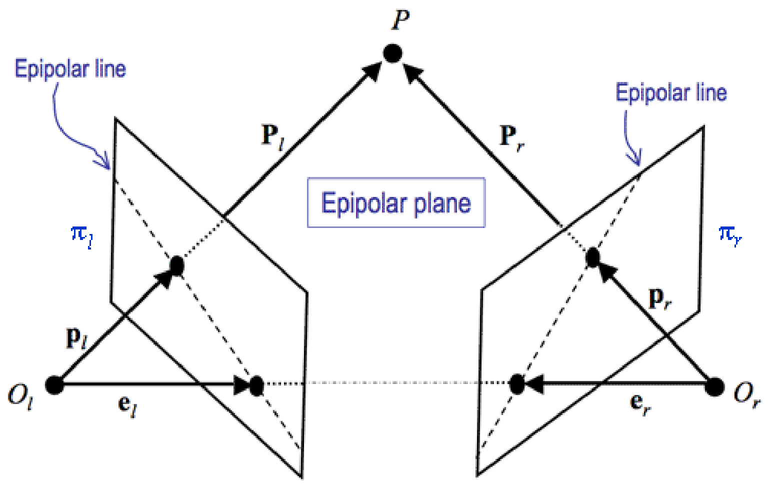

Figure 2.5: Epipolar line
O_l and O_r is the projection centers, P_l and P_r
are the vectors through the projection plane,
e_l and e_r are the epipoles while π_l and π_r is the image plane.

2.5.3 Fundamental matrix

The fundamental matrix F is an algebraic notation of epipolar geometry. It is used for finding a points corresponding epipolar line in a different view. F is a square matrix with nine entries and size 3x3. By multiplying a point in one view with the fundamental matrix, we will find its corresponding epipolar line in the second view. If a world 3D point is set to x in one view and x′ in another view, then we can create an equation $x'^T * F * x = 0$. We can use this equation to relate x to x′. Two points that satisfy this equation will be coplanar, which they need to be for point correspondence [26].

2.5.4 The eight-point algorithm

The eight-point algorithm is a way to estimate the fundamental matrix from eight or more feature correspondences in two views. As stated above, the fundamental matrix is defined by the equation $x'^T * F * x = 0$ for matching features in two views. When we have eight such correspondences, this information can be used to calculate the rotation and translation between our two views in the form of the fundamental matrix. For each pair of matches, we formulate a homogeneous pair of linear equations with nine unknowns. F, x and x′ will look like the following:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \qquad x' = \begin{pmatrix} x'_1 \\ x'_2 \\ 1 \end{pmatrix} \qquad F = \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \tag{2.1}$$

With the number of correspondences $N >= 8$, we get a linear system $Af = 0$ where A is the equation matrix formed by stacking the data vectors, and f is a nine vector containing the entries of the fundamental matrix F.

We can solve this with a singular value decomposition. Since we may have noise errors in our correspondences, we will get our best estimate from a least squares solution. The solution will be the Eigenvector corresponding to the smallest Eigenvalue $A^T * A$. From this we can find the singular value decomposition of F, get the Eigenvalues of $A^T * A$ before recomputing F. As our feature matches are likely to be imperfect, we want to increase accuracy and stability by preceding the algorithm with a very simple normalisation. This normalisation will be in the form of translation and scaling of the coordinates of our matched features, such that they have an average of zero and standard deviation of one. On algorithm completion, we will denormalize fundamental matrix F [39][27].

2.5.5 Triangulation

At this point, we have calculated the relative orientation of the images, and we want to use this information to compute the 3D coordinates of our corresponding points. The first step in doing this is to compute the 3D location relative to one of the view frames (typically the first view). A geometric approach known as triangulation is used to perform this calculation. The baseline for the triangle is given by the projection centres of our two views. These two points have a direction vector each that point to the matched features in the 3D world. The last point of the triangle is at the intersection of these lines. To find the exact location, we need to estimate what the scaling parameter is. As this cannot be determined mathematically, we need extra knowledge. However, we know the angles, so if we can measure the distance between our views through a GPS or some other tool, we have the necessary information. Unfortunately, this approach does not always work, because, in the presence of noise, our lines will not be guaranteed to intersect as we may not have a perfect correspondence between the images. This means that the provided points do not satisfy the epipolar constraint $x'^T * F * x = 0$. The result of this will be that the lines do not intersect in 3D. There are many ways to fix this, one way is to draw a new line perpendicular to our two lines at the point where they are closest to each other, and set the last point in the triangle in the middle of this new line [26].

2.5.6 Absolute Orientation

With triangulation, we have found out where are the points are concerning the camera projection centre. The next step is to figure out where the points are in the world. To do this, we need to relate the point cloud that we computed to an external reference frame. There are several ways of doing this, the most accurate one is to use known ground truth points, but this is often not available. Another way is the two-step approach where we first find the plane at infinity to calculate an affine transform, before identifying the absolute conic to get the metric reconstruction. One way to find the plane at infinity is to look at three intersections of parallel lines in our scene. The points at these intersections will give us the plane at infinity [26].

2.5.7 Bundle adjustment

Bundle adjustment is an algorithm that reconstructs the 3D structure and camera parameters from a series of scene images. Its main usage is to refine an existing but uncertain 3D reconstruction and camera parameters to produce optimal accuracy in the reconstruction. Bundle adjustment improves the reconstruction by minimising a cost function. This cost function is the reprojection error, which will give us a maximum likelihood estimation of the camera parameters and the 3D reconstruction [64]. It refers to the 'bundle' of light rays from each 3D feature converging on each camera centre, and are 'adjusted' ideally concerning both feature and camera positions. This is different from the independent model's methods that merge partial reconstructions without changing their internal structure. Bundle adjustment adjusts both the structure and camera parameters together in one bundle.

Chapter 3

Concepts in optimization

3.1 Complexity

To evaluate an algorithm, we need to have a concept to describe the complexity that is universal. It will not be enough to just run timed test to see the benefits of a solution, as results will vary depending on many different factors.

3.1.1 Big O notation

To describe the complexity of an algorithm computer scientists uses what we call a big O notation. It is used to describe the worst case scenario of the execution time required. It isn't necessarily a measure of time itself. It rather describes the growth of a problem as the input increase. When referring to big O, it is written with O and the growth inside parentheses after it, for example, $O(1)$. This case suggests that the problem is constant, regardless of the size of input it will always take just one iteration to execute the algorithm. Imagine if you want to find the largest number in a sorted list, regardless of the size of the list, you only need to look at the last number to find what you are after. Figure 3.1 on page 20 illustrates a complexity graph.

Linear time

$O(n)$ suggest that the problem is linear, which means that it grows simultaneously with the size of the input. If we were to do the same as in the previous example except that the list is not sorted, we would need to go through the whole list before we could be certain that we had found the largest number.

Logarithmic time

$O(\log n)$ tells us that a problem has a logarithmic runtime. Take an old-fashioned phone book, there are n people listed in it, and they are all sorted. If you want to find a specific person by name in it, you could pick a name at some point in the middle of it to start with. The next step would be to

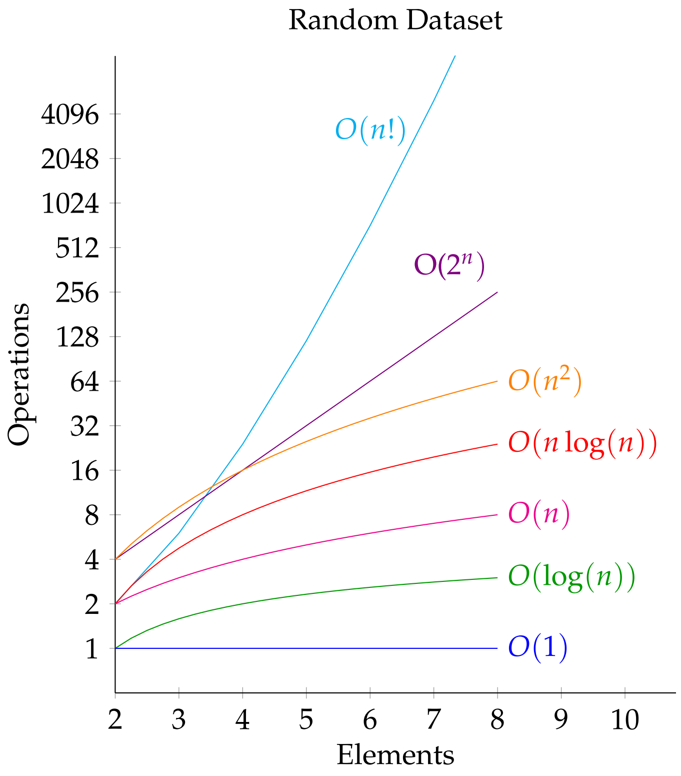

Table 3.1: Time Complexity

see which side of the middle the name we are searching for would be, then we do the same for this side, picking the middle of it and repeat this until we have found the name we are searching for. If the name does not exist it would have used log n iterations to conclude this.

Polynomial time

$O(n^2)$ describes a problem that has a quadratic growth as the input grows larger. If we, for example, need to find a duplicate number in a list, we would need to first find one number from the list and then search through it again to see if there are any duplicates. Of course one might be lucky and find it quickly, but big O shows us the worst case scenario where we would need to search the list in n^2 iterations.

Exponential time

$O(2^n)$ tells us that the problem is exponential and grows drastic for larger inputs. Big O notation also excludes constants because it is calculated as the input moves towards infinity. So for example if you have a problem that has n + 100 iterations, n will eventually grow so much larger than the constant does not have any importance at all, so instead of writing O(n+100) we just write $O(n)$.

3.2 Good code versus bad code

Programming software today is often based on just making it work, and not necessarily doing it the correct way. There are many roads to Rome, and there are many different ways to code a program. It can make a huge difference to the flow of a program depending on how it is coded. Bad code and memory management can result in a slow and expensive program compared to optimized code. Some solutions turns to parallel computing for improved runtime, but parallel computing consists of serial code and will perform better if the code is done properly.

3.2.1 Small optimizations

A simple way to optimize code is to make sure that the same memory is used at the same time. Take this two for loops into consideration:

```
for (int i = 0; i < 1000; i++)
    x += a[i] * b[i];
for (int i = 0; i < 1000; i++)
    y += a[i] + b[i];
```

Both these loops will run 1000 times each, and although there are different operations, the same memory is read each time. So the correct and best way to do this would be to merge the two loops into one:

```
for (int i = 0; i < 1000; i++) {
    x += a[i] * b[i];
    y += a[i] + b[i];
    }
```

Another example is to read or write to memory the best possible way. If you have a two dimensional matrix and need to access the elements in them, you will need to run a double for loop to access all the data.

```
for (int i = 0; i < 1000; i++)
    for (int j = 0; j < 1000; j++)
        y += a[j][i];
```

In the code above, we access each element column wise reading $a[0][0], a[1][0], a[2][0], ..., a[999][999]$. It looks like a decent solution, but since the data in memory is stored row-wise in languages of the C family (C++, Java, C#, Go), this is actually an ineffectual way to do it, as each iteration has to discard the cache that has been built up and read a new line from memory to access the new query. The solution is to program after the memory layout, and access the matrix row-wise reading $a[0][0], a[0][1], a[0][2], ..., a[999][999]$.

```
for (int i = 0; i < 1000; i++)
    for (int j = 0; j < 1000; j++)
        y += a[i][j];
```

By doing it this way, the cache will contain the data we are accessing more often, reducing the need for reading data and thus improving the efficiency of the program. Good compilers will do improvements like these automatically on a CPU, but not on a GPU as it will assume the setup is intended by the programmer. Getting the best performance out of a program is not just about using the ideal algorithm or finding the best pipeline, it is also about programming optimally. There can be a huge difference between "good" code and "bad" code, and it is key to all programming to understand and use the best possible code for each scenario.

3.3 Parallel processing

3.3.1 The CPU

With the technological advancements of recent years, there has been a shift from single core computation to multi-core computation. Part of the reason for this advancement was that developers and manufacturers understood it was necessary to look for new methods to increase computational power. This understanding came as a result of the traditional methods of increasing computational power such as increasing the processing units clock speed no longer was a viable option because of heat- and power supply restrictions. Parallel computing was the proposed solution, as it allows computing speed to increase linearly with the number of extra cores. Therefore it is no surprise that new computers, whether it is a stationary or a laptop, almost always are created with multi-core central processors.

3.3.2 The GPU

The GPU (Graphical Processing Unit) is a tool specializing in processing graphics, such as the creation of an image before it is displayed on the connected monitor and various other graphical applications. It was popularized in 1990s due to the increasing popularity of operating systems with graphical user interfaces. As the GPU is in its core components is built to excel at rendering independent pixels in an image, its structure is massively parallel. This structure makes the GPU surpass the CPU on tasks that allow for high levels of parallelism. The general-purpose GPU (GPGPU) trend started with people who did shader programming: re-formulating matrix multiplication problems to look like 3D texture projections. That led to better tool support, which led to more interest, which led to GPUs with more CPU-like capabilities.

3.4 CUDA - The Compute Unified Device Architecture

In recent years, there has been considerable progress in programmable interfaces targeting the GPU. One of the tools created is the parallel computing platform and application programming interface (API) model created by NVIDIA, named The Compute Unified Device Architecture, or just CUDA [54]. CUDA is a vast improvement from earlier techniques as it is simply an extension to the C++ programming language. CUDA allows the GPU to be applied as a general purpose computing tool. Compared to a CPU the GPU easily outperforms it on tasks that allows for high parallelism as the GPU itself is built to run concurrent operations. Depending on your GPUs compute compatibility, CUDA C++ is structured into a grid of blocks, each block sharing memory which depending the specific GPU can run up to 1024 threads in parallel. Allowing for parallelism to be split up in a large amount of blocks and threads in this manner greatly simplifies a programmer's ability to tailor an algorithm to their needs. As an example, GEFORCE® GTX 1080 parallelism and memory capacities can be found in table 3.2 on page 23. With seemingly infinite parallelism it is easy to find

CUDA Capability Major	6.1
Total global memory	8111 MBytes
(20) Multiprocessors, (128) CUDA Cores/MP	2560 CUDA Cores
GPU Max Clock rate	1757 MHz (1.76 GHz)
Memory Clock rate	5005 MHz
Memory Bus Width	256-bit
L2 Cache Size	2.0971 MBytes
Total amount of constant memory	65536 bytes
Total amount of shared memory	49152 bytes
Total number of registers available per block	65536
Warp size	32
Max threads per multiprocessor	2048
Max threads per block	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid block (x,y,z)	(2147483647, 65535, 65535)

Table 3.2: Parallelism and memory capacity of GTX1080

algorithms that will benefit greatly from this technology, as many scientific problems are computationally intensive, and allow for high parallelism.

3.4.1 Memory

CUDA global memory

Global memory is allocated from the CPU and shifts memory between the CPU and the GPU. No other memory type can do this, as all data not in global memory will disappear once the kernel execution completes [43]. Global memory is likely to be very large but is dependent on the specific GPU, and not its compute compatibility.

CUDA local memory

Local memory is an abstraction of global memory and has high latency and low bandwidth [20]. It is not a memory type as it is something the compiler does with global memory. Local memory is however slightly faster than global memory on parallel operations due to interleaved addressing. When organised such that continuous four-byte data are accessed by continuous thread id's we get Interleaved memory.

CUDA constant memory

With the massive parallel abilities of the GPU, the program bottleneck in addition to allocation and synchronisation is often memory bandwidth. The solution to this can be the constant memory that can lower memory traffic. Constant memory tells the compiler that this memory is read-only. Reading from constant memory is beneficial compared to reading from global memory, as it broadcasts the read to threads with similar indices. Reads of same address will also be fast, as constant memory is cached. This caching means that constant memory is very effective when several threads read from memory that is close together. Constant pre-calculated tables is an example of efficient use of constant memory.

CUDA shared memory

Shared memory is often used and has several advantages. Firstly, the threads in a block share this memory. This way it is easy for threads to operate on the same data in unison with each other. Since shared memory is located on-chip, it is faster than both local and global memory [56]. There are however limits to it both in memory size and in memory access, and there is often a need for synchronisation that will slow it down. When there are no bank conflicts (Simultaneous access to the same memory bank), the speed of the shared memory is comparable to registers memory.

CUDA register memory

Register memory is limited in size and is only accessible by one thread. However, with the speed of which accessing a register takes (zero clock cycles), this is by far the fastest memory CUDA provides. In some instances such as a read depending on a write to be completed first, we may have some extra latency. However, this latency will be hidden entirely if enough threads run in parallel. Additionally, there are fast but functionally limited shuffle operations that allow threads to exchange units of register memory. If there are not enough registers available for a specific kernel, the local memory will take over.

CUDA texture memory

Texture memory is read-only. On specific read orders, it can enhance performance. As texture memory uses the on-chip cache, it will reduce memory requests to off-chip DRAM. In particular, texture caches are designed for graphics applications where memory access patterns are located close to each other in both horizontal and vertical direction. It is a layer that modifies the way in which we read global memory. For 1D, 2D or 3D arrays, texture memory allows for efficient cache-line fetching in all dimensions.

3.4.2 CUDA warps and occupancy

A warp consists of 32 threads and is a Single Instruction Multiple Data (SIMD) processing unit. Thus, a warp is a set of 32 threads which share instruction pointer, and the threads within a warp will execute the same code [18]. The threads within a warp all have sequential indexes, from 0-31, the next warp starting from 32 to 63. If the programmer asks for a specific thread within a warp to operate, the remaining threads within the warp will wait for the completion of this operation before continuing [25].

The definition of occupancy is the number of warps that can run concurrently on a multiprocessor divided by the theoretical maximum number of warps that are allowed to run concurrently [35]. To reach 100% occupancy the number of warps currently executing must be equal to the theoretical maximum.

3.4.3 CUDA synchronization

There are several synchronisation directives in CUDA. cudaDeviceSychronize() is called from the host and synchronises the host with the device such that the CPU is waiting for all pending GPU activity before continuing. This synchronisation is often used after a kernel call before the result data is copied to host memory. It is not always needed to do this explicitly as certain operations such as cudaMemcpy() have built-in blocking device synchronisation mechanisms. It is also possible to purposely run the host and device code asynchronously in cases where it is beneficial to have them run concurrently.

_syncthreads() is used inside the kernel and is often used as a barrier in algorithms where threads read and write to the same shared memory. It can also be used after branching to assure that updated and not updated values are not used together. _syncthreads() only synchronizes threads within a block, not between blocks.

3.4.4 Thrust

Thrust is a template library for parallel algorithms and data structures on the GPU. It provides an interface for parallel computing focusing on overall programming productivity and real-world performance. The algorithms are implemented in such a way that thrust itself figures out the optimum number of blocks and threads to run [36]. Therefore it is quite easy to use, as no greater understanding of the architecture is necessary for efficient code. It will be easier to apply a variety of algorithms that provide high performance and significant optimization on tasks that are otherwise considered to be computationally demanding. Thrust is especially good at solving the type of problems which can be implemented in a good way without having to go into the depth of architecture. Other algorithms that fit well are those that the programmer will doubtless be able to improve or have no time to look at [7]. Using Thrust, the programmer can solve his problem in a good way using high-level algorithms and make decisions about specific implementation details completely controlled by the library. Since the library is responsible for providing an effective implementation, Thrust will investigate different options, choosing the best implementation that is available for the given parameters. Another benefit of this is that the developer can spend more time on other tasks, giving the development process increased productivity. Thrust allows the developer to run complex GPU accelerated algorithms in just a few lines of code. High-level libraries have long been widely used in fields where high performance is important. An example is the BLAS standard, an abstract linear algebra interface that is about 30 years old [37]. BLAS is still used today, and many of the reasons are because it uses platform-specific optimizations that are handled by the interface, without the help of the programmer. In contrast to BLAS, which mainly deals with numerical linear algebra, Thrust is an interface for some specific parallel algorithms such as sorting and reduction. Since Thrust uses C ++ templates to make its algorithms generic, it is possible to use operators and types defined by the programmer [7].

Chapter 4

The project

4.1 OpenMVG

OpenMVG is a library written in C++ which provides different tools, techniques, and algorithms for computer vision and structure-from-motion. Its main purpose is to provide an easy to use, and easy to learn collection of modules that is open for all to use. The different parts are independent and can be either used by itself or as part of a larger set-up. Created with the philosophy "Keep it simple, keep it maintainable" [52], it sets out to be a software that inspires to be used and modified to extend its quality and flexibility.

4.1.1 Design

There are multiple goals that OpenMVG sets itself. An easy access to accurate implementation of multiple view geometry algorithms, a simple and understandable source code library and a set of tools that can be used to build complete applications. You will find functions for image loading and processing, feature detection and matching, multi-view geometry solvers and easy access to linear algebra and optimization frameworks. It also delivers a collection of modular core features in small libraries that covers different areas of computer vision. The modules are independent, but can also be used as part of a complete pipeline to perform 3D reconstruction from images (SfM) or localize images into an existing 3D reconstruction.
To achieve the goal of being simple and maintainable, openMVG uses test-driven development. This will help to ensure that algorithms and code works as expected, and that external users can implement their own methods and verify them via tests.

4.1.2 Functionality

OpenMVG offers a wide spectrum of functionality to its users, covering many of the key elements in 3D reconstruction.

Functions	
• Image Processing: A simple image handling module that acts as a 2D template pixel container based on the Eigen matrix structure	• Feature extraction and Description: Detecting distinctive, repeatable image points and descriptors
• Feature and image collection matching: An abstract nearest neighbour search framework	• Multiple view geometry: Allows to check some multiple view geometric constraints
• Robust estimation: Removes outliers in noisy data	• Structure from Motion: Complete 3D-reconstruction pipeline, both incremental and global
• Localization: Enables the possibility to find the camera pose and orientation from images of an existing reconstruction	

Table 4.1: Functions and modules in openMVG

4.2 PopSIFT

PopSift is a GPU implementation of SIFT in CUDA. It is created with extra attention towards speed as well as following the algorithm outlined in the original SIFT paper [40] in detail. PopSift can process 25 frames per second or more with a modern GPU on HD images. Due to its complex nature, this is quite remarkable, as SIFT has been thought of as a slow algorithm in the computer vision community to the point where several other algorithms [5, 57] has been created to reduce the time requirements for feature extraction and description. However, these algorithms do not yield comparable results. Several GPU implementations of SIFT has already been created, however many of them are not publicly available.

Among those implementations which are available, none of them achieves the speed requirements of real-time feature extraction and description in addition to attaining comparable accuracy. An example of this is CudaSift [10] which is much faster. However, it finds entirely different keypoints, which affects scale independence negatively, in addition to making it impossible to combine with e.g. VLFeat [65], OpenCV SIFT [12] or PopSift.

PopSift is available for use under a permissive open source license.

4.3 RANSAC - Random Sample Consensus

This section is based upon the original RANSAC paper [22] as well as lectures from Aaron F Bobick [11]. RANSAC, Random Sample Consensus [22] is used for interpreting and grouping data containing a significant percentage of errors. Automated feature detection and matching can be prone to errors. Thus, RANSAC is a handy tool to filter out incorrect matches. These false matches are known as outliers. We can compute the relative orientation between images from a set of corresponding features obtained by SIFT or some other feature extractor. However, as our matching produces a percentage of feature matches with orientations distant from the majority consensus, we need to filter out these false matches (outliers), so we can determine the relative orientation between the images with certainty. We Filter outliers by proposing an orientation before counting how many matches have values close to the proposal. We repeat this process until we have an orientation with many feature matches. As outliers likely will be randomly distributed, this method can handle a large number of them. How many feature matches we need to create the initial model depends on what information we are looking for; we need two points to calculate the translational shift between two images, four points to calculate the homography, and 8 points to calculate the fundamental Matrix. A classic example of RANSAC can be found in figure 4.1 on page 30.

4.3.1 General RANSAC algorithm

1. Randomly select S points (number required by the model)
2. Create the model from the selected points (draw a line/compute homography/fundamental matrix)
3. Score according to counted points within the distance threshold
4. Repeat N times or until the termination condition is reached and keep the best score

4.3.2 RANSAC algorithm for SIFT

The RANSAC algorithm has four main steps. The first step of RANSAC is to pick a sample subset of random feature matches from the complete set of matches. The number of points in the subset is equal to the minimal number of points required by the model. In our application, we want to compute the fundamental matrix, so our minimal subset consists of 8 feature matches. The fundamental matrix relates points in one scene (image) to lines in another. The second step is to compute the fundamental matrix that fits this sample subset. The third step is to use this model to determine which feature matches from the complete

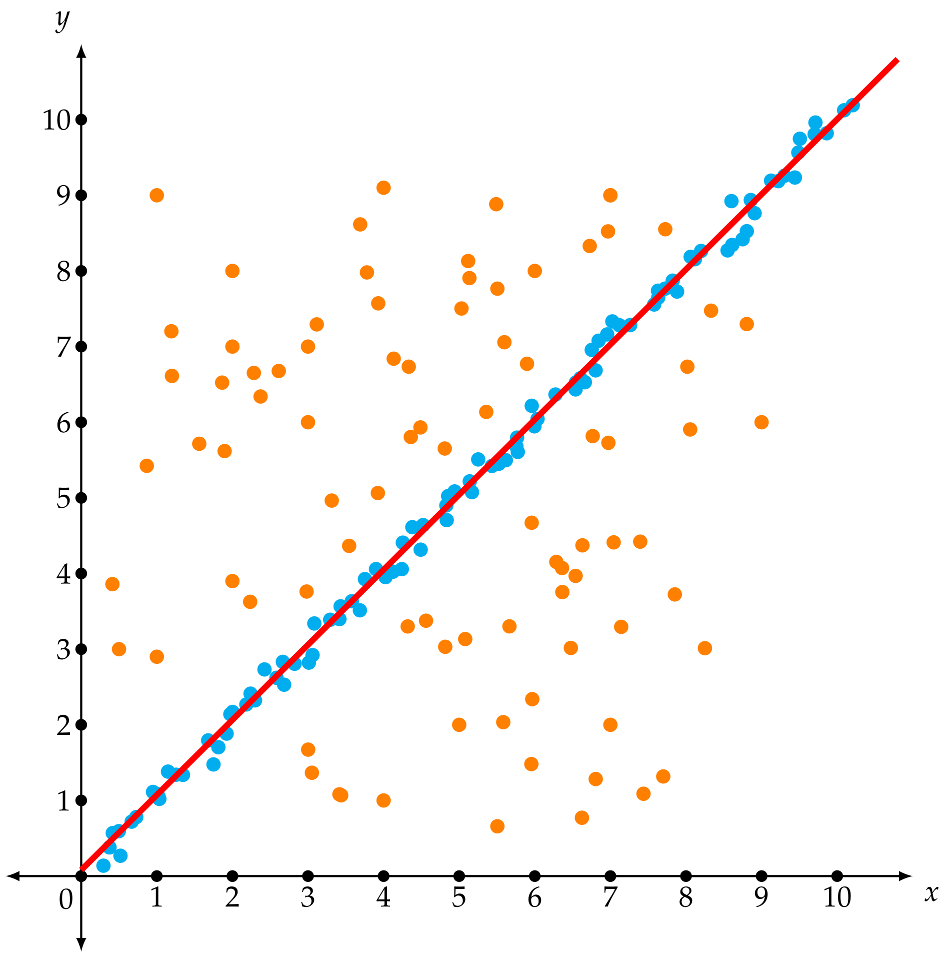

Figure 4.1: Ransac model fitting

set are consistent with the fundamental matrix of the initial subset. A feature match will be regarded as an outlier if its deviation from the model is above some threshold. The feature matches within bounds of the threshold are known as the consensus set. The fourth part is to repeat these steps N times, or until the consensus set is sufficiently large, determined by some preset threshold. On completion, the fundamental matrix is re-estimated to improve the result further. The re-estimation is based upon all feature matches in the consensus set. As a side note, the estimated fundamental matrix might not be entirely correct as the theory is based on the pinhole camera model, while in reality, cameras often have lens distortions. Regardless of this, the estimated fundamental matrix is efficient at filtering out outliers from the correct SIFT matches.

4.3.3 Choosing parameters

Points in subset

The points in the subset are decided by the model (line, homography, fundamental matrix) and are the minimum number of data points required to obtain this model.

Distance threshold t

If the distance threshold parameter is set very high, it is difficult to accurately determine the correctness of the model as we will get many misclassifications, where points not part of our model are considered inliers. On the contrary, if the distance threshold parameter is set too low, the resulting estimate tends to be unstable, as many points that in reality are part of our model are classified as outliers. [63] We solve this by setting the distance threshold t such that a high percentage of inliers have a distance d within the threshold $d < t$. If $t^2 = 3,88\sigma^2$, then there is a 95% probability that a points distance from the model is lower than the threshold $d < t$ when the point is an inlier (σ = standard deviation). We can change the constant to change the inlier percentage.

Number of samples N

N is the number of samples to be processed. Depending on the size of our dataset, to exhaustively try all possible samples is often not attainable in practice, even with high-performance computers. However, neither is it necessary. Instead, we will choose the number of samples such that we can ensure with probability p that at least one of our random sample sets is just made up off points coming from the correct model (e.g. p = 0.99). To do this, we need to set the number of samples N based on the outlier ratio e.

Calculate N

The size of N will be estimated based upon the outlier ratio e, the size of the minimal sample set s, and the probability P for a sample set to contain inliers only.

- s is the number of points in the set
- P is the probability of success
- e is the proportion outliers, so % inliers = (1 - e)
- P(sample set with all inliers) $= (1 - e)^s$
- P(sample set will have at least one outlier) $= (1 - (1 - e)^s)$
- P(all N samples have outlier) $= (1 - (1 - e)^s)^N$
- We want P(all N samples have outlier) $< (1 - p)$
- So $(1 - (1 - e)^s)^N < (1 - p)$
- From this we get our equation: $N > log(1 - p)/log(1 - (1 - e)^s)$

Analysis of the N formula

When using the above formula, N is relatively steady whether the proportion of outliers e is high or low, while it increases steeply with the number of points in the set s. Another important thing to note here is that the number of points in the image, or more precisely, the number of features, is not included in the formula. Thus, the number of samples needed has nothing to do with the number of possible matches. It is only concerned with the percentage of wrong ones, effectively making the algorithm scale invariant. This scale invariance is a fundamental reason for the success of the RANSAC algorithm.

The main problem with the above formula is that the proportion of outliers e is not known. If we knew which points were outliers, we could simply ignore them altogether. As mentioned in the previous paragraph the number of samples N is relatively steady whether e is high or low. We can use this knowledge to our advantage and set the initial e to be a worst case value without any significant increase in algorithm run time. E.g. 50% outliers (e = 0.5) and adapt if more inliers are found. The adaptive procedure would look like the following:

- Adaptive procedure
- $N = \infty$, sample_count = 0, $e = 0.5$
  - while $N >$ sample_count
  - Choose a sample and count the number of inliers
  - Set $e_0 = 1 - \frac{numberofinliers}{totalnumberofpoints}$
  - If $e_0 < e$ Set $e = e_0$ and recompute N from e
  - $N = log(1-p)/log(1-(1-e)^s)$
  - Increment the sample_count by 1

Termination condition - Lower bound on an acceptable consensus set

RANSAC terminates after running through the steps in the algorithm N times, or when the lower bound on an acceptable consensus set is reached. A threshold is set to determine if a subset is sufficiently large. Reaching this threshold will allow the algorithm to terminate. Thus, the chosen size of the termination threshold must be large enough to ensure that the computed model is correct (relative orientation between the images). The consensus set should be large enough to satisfy the needs of the re-estimation procedure for model improvement. As the correctness of a consensus set cannot be guaranteed, the researchers suggest setting the threshold such that the probability for the set to be incorrect is lower then 5%.

Pros	Cons
• The algorithm is robust and will yield a correct result, even with a high percentage of outliers • Not sensitive to the number of feature points in the image • Allows for parallelisation with minimal modifications to the original algorithm • A general algorithm fundamental in computer vision, applicable to many kinds of model fitting problems	• Runtime increases drastically with the size of the set s and is therefore heavily reliant on the model used • Multiple fits increases probability of failure, for example, multiple planes. Therefore a carefully set threshold is required. • Needs a specific model, approximate models (for example a plane, which is not a plane will likely not work). Important that the model fit the underlying structure

Table 4.2: Pros and cons of RANSAC

4.3.4 RANSAC conclusions

4.4 Approaching the problem

There are many approaches to improving a program today, from modifying the code structure in the original implementation to replacing algorithms with more effective ones.There are also different hardware like GPUs that can improve the efficiency drastically. The purpose of the code often defines what tools and approaches may be available.

It is also essential to find the bottlenecks of the code, and try to improve them; speedup in these sections has a more significant impact on the runtime in comparison to the parts that have a lesser say in the overall runtime. In general we look for repeating patterns in the code, parts that do the same or similar operations several times in the program. When we find these parts we start analysing if there is some unnecessary repetitions or if the code is written badly. It is quite astonishing what simple changes in the code can do for the efficiency of a program. The next approach is then to analyse what the code actually does and see if there are other algorithms or solutions that may offer the same result more efficiently than the one present. After a successful implementation, apply new tests to see if the bottlenecks have shifted elsewhere in the program and improve them as well.

Early on in OpenMVG we quickly found a bottleneck in the matching process. The program has to find corresponding features from one image to another by exploring all image descriptors given. This part is essential for the process and is quite time-consuming given a considerable input of images. To improve this part we needed to understand how the process works, and if there are any alternatives to it. In the next chapter, we will explain what the matching process is, and what tools and algorithms that are available to implement it.

4.5 Implementation

As we mentioned earlier, we thought that the exhaustive search would have the biggest potential for improvement. In OpenMVG, this part is done relatively straightforward, by iterating through each descriptor in one image and comparing it to all the other descriptors in the corresponding image. The exhaustive search is a costly method with complexity $O(n^2)$.

The comparison is made by calculating the squared Euclidean distance between each descriptor and keeping the two best matches. If we are to improve the exhaustive search, we will need to change or replace this computation to something more efficient. We will explain the complexity of the Euclidean distance later in section 6.1 on page 45.

The next approach will be to implement an approximate search which has a better complexity than the $O(n^2)$ the exhaustive search provides. This solution will have to sacrifice some precision, and our main goal will be to limit this sacrifice compared to the other ANN searches available in OpenMVG.

There are both a CPU implementation and a GPU implementation of the exhaustive search. We will begin to change the CPU part first, and if the results are positive, we will then implement them on the GPU version. As we want to keep the quality of the search, we will begin to look at the calculation and keep the structure of the exhaustive search for now.

Chapter 5

Assessment of feature matching

Previously we discussed how to detect SIFT feature points as well as how we can describe them. In this chapter, we will go over some commonly used algorithms for matching, as well as their positive and negative attributes. These algorithms are not limited to matching sift descriptors, but function on a wide variety of descriptors.

5.1 Distance threshold

To find the most likely match for a descriptor we identify its nearest neighbour in the matching set. The literature defines the Nearest Neighbour (NN) search problem in the following manner: given a set S of points in a space M and a query point q ϵ M, find the closest point in S to q [17]. The closest point is defined based on the metric used. The Euclidean distance metric is considered the gold standard of metrics, but others exist. However, not all descriptors will have a good match. One way to reduce this would be to set a global distance threshold on the distance to the nearest neighbour. However, this method is not used in practice, as descriptors vary in how discriminative they are [40]. Instead, Brown and Lowe [14, 40] proposes a distance threshold between the nearest neighbour and second nearest neighbour. This Nearest Neighbour distance ratio [47] is defined as follows:

$$Distanceratio = \frac{d_1}{d_2} = \frac{||D_A - D_B||}{||D_A - D_C||}$$

In the above equation d_1 and d_2 are the shortest and second shortest distances, D_A is the target descriptor, and D_B and D_C are the two closest neighbours. This threshold between the two nearest neighbours removes a potential match if there are two close possibilities. When two close candidate matches are closer than the threshold allows, we discard the descriptor with the reasoning that the likelihood of the match being a false positive is too high.

5.2 Nearest Neighbour using Exhaustive search

Assuming that the Euclidean distance comparison can be used to rank potential matches, the first step is to check whether some axes in the descriptors are more reliable than others. If so, these axes are re-scaled ahead of time by comparing them against verified matches to determine divergence [31, 62]. The nearest neighbour to a point calculated using Euclidean distance is the point with the shortest Euclidean distance from the query descriptor. As the feature descriptors will be highly distinctive on average, each feature will have a high probability of finding its correct match. We match features after creating feature descriptors from two or more images. By finding the Euclidean distance between one feature descriptor in image one, and all feature descriptors of image two we can match them. Since we want distinct stable matches, we do not consider a feature to have a match if the two closest matches are too similar (distance threshold). Thus, we order all the descriptors in an array, from best match to worst match and set a threshold. The distance between the best match and the second best match must be above this threshold for the features to be considered a match. This procedure repeats until the steps complete for all feature descriptors in image one. The exhaustive search algorithm is considered to be the best solution for Nearest Neighbour searches because it guarantees to find the best match. However, due to the quadratic runtime to the number of feature descriptors, it is not often used in applications with large descriptor numbers [62].

5.3 Approximate Nearest Neighbour search

Depending on what the matching purpose is, it might be acceptable to guess which descriptor is the nearest neighbour. Some of our matches will be incorrect due to other factors, so a slight increase in this number will not always matter. An algorithm like RANSAC can be used to filter correct matches from incorrect matches.

5.4 FLANN

An exhaustive search in matching will explore all possible matches and find the best matches correctly. The main problem is that there are often a large number of points to search with high dimensionality. This means that the matching process will slow down the program and in some cases make it unusable. As a consequence of this, there has been a significant development of different solutions that sacrifices precision to achieve a better overall performance. FLANN (Fast Library for Approximate Nearest Neighbours) is a matching library designed for higher performance, sometimes at the cost of accuracy. It contains several solutions for different scenarios, including K-Nearest Neighbour search (KNN), which will provide a fixed number of matches, and Radius Nearest Neighbour search (RNN), which provides the matches that satisfy a given threshold.

KNN is defined as:

$$KNN\,(q, P, K) = A$$

where A:

$$|A| = K, A \subseteq P$$

$$\forall x \in A, y \in P - A, d(q, x) \leq d(q, y)$$

KNN returns exactly the K best matches from the neighbours if there are enough points in P.

RNN is defined as:

$$RNN\,(q, P, R) = \{p \in P, d(q, p) < R\}$$

RNN will return anything from zero to the complete set of points depending on the value of R.

KNN and RNN can combine with Radius K-Nearest Neighbour search (RKNN), where K limits the number of points from R [53].

RKNN is defined as:

$$RKNN\,(q, P, K, R) = A$$

where:

$$|A| \leq K, A \subseteq P$$

$$\forall x \in A, y \in P - A, d(q, x) < R \text{ and } d(q, x) \leq d(q, y)$$

There are three main categories of algorithms in Nearest Neighbour searches: partitioning trees, hashing techniques and neighbouring graph techniques.

5.5 K-Dimensional Tree

A k-d tree is a space-partitioning data structure that organizes points in a k-dimensional space. It is a multidimensional binary search tree [8] that can be used for numerous applications such as range searches and Nearest Neighbour searches. The main idea behind it is to make searches more effective by organizing the data structure beforehand. There are many variations of it, depending on the size and type of data used. The tree is built by finding the median in one dimension, and then splitting the points into two branches, in each of these branches the same is done for the next dimension. This is repeated until each dimension is used resulting in a binary tree that contains all points in the leaf nodes. Each leaf node will represent a point; however, some implementations allow each leaf node to contain more than one point [61]. The fully built tree will have a height of $\log_2 N$ where N is the number of points in the dataset. Inserting a new node to the tree is done by looking at the first dimension, and see if the value is smaller or larger than the root node, then the same is done for the next dimension in the next branch until a leaf node is reached. Search is done exactly the same, by using each dimension to find the leaf node, resulting in a complexity of $O(\log n)$ for finding the leaf node. However, there is no guarantee that the nearest neighbour is in the leaf node reached, so the algorithm then finds the best point in this node, and if there are potentially better points, it will backtrack and search in the other nodes. The recommended method for this part is the priority search [6], where the backtracking will be done in cells in the order of their distance from the query point. This will in the worst case scenario result in a time complexity of $O(n)$.

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Table 5.1: Complexity of k-d trees

In high dimensional spaces, the k-d tree Nearest Neighbour becomes inefficient. In general, with k-dimensionality, the number of points in data N, should not be $N \gg 2^k$. If this is not the case, then the algorithm will evaluate most points in the data set, resulting it to have the same runtime as an exhaustive search. This is a result of the algorithm needing to backtrack further as the number of dimension increase. If the backtracking is then reduced by limiting it, the certainty of finding the best match is sacrificed resulting in an approximate search instead. The quality of the result diminishes as the dimensions increase. Below in Figure: 5.1 on page 38 is an illustration of a k-d tree built from the points: (1.9), (8.8), (2.3), (7.2), (4.1), (9.6), (3.7), (7.9), (5.4), (6.8). And in Figure: 5.2 on page 39 the illustration of the sectors the points are in.

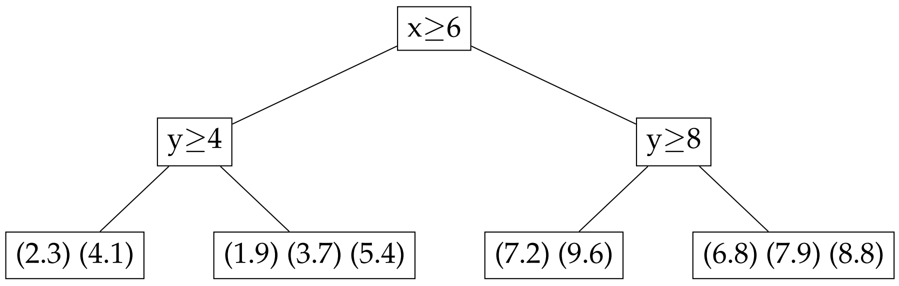

Figure 5.1: k-d tree illustration

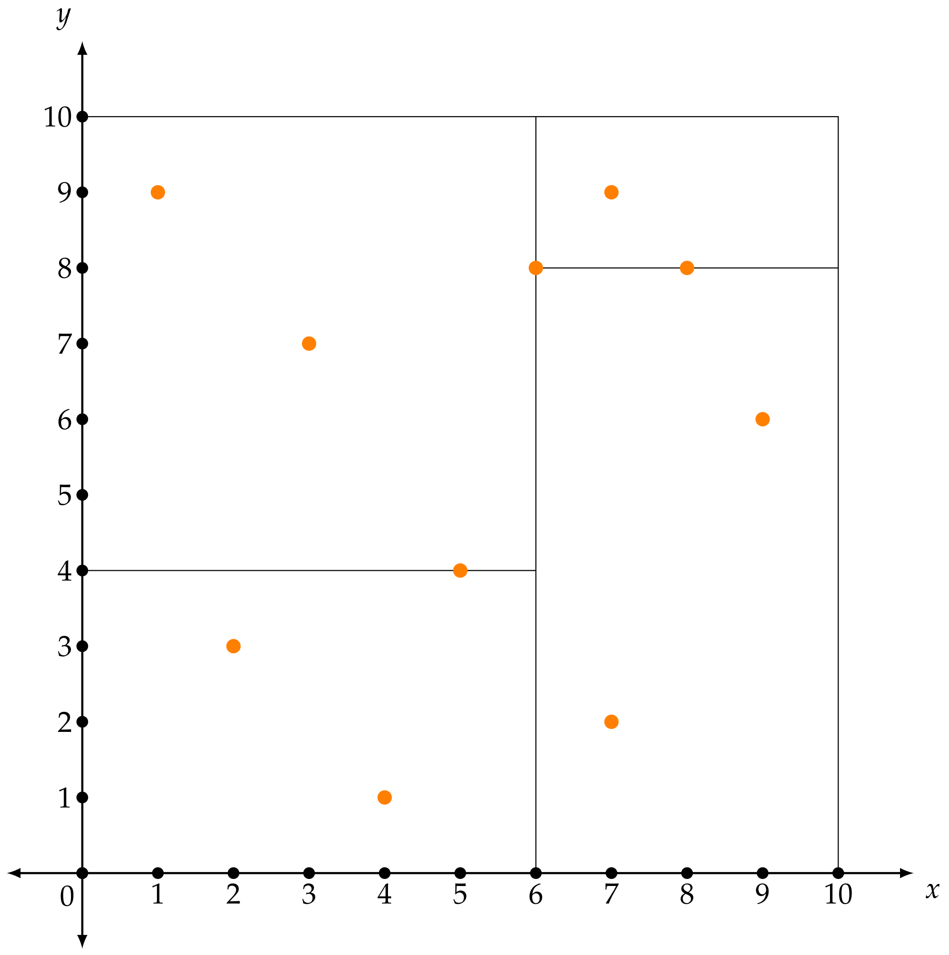

Figure 5.2: k-d tree illustration showing points in sectors

There are many different implementations of approximate Nearest Neighbour searches in k-d trees. In 1998 Arya [3] proposed a solution that imposed a bound on the accuracy by using the notion of ε-approximate Nearest Neighbour. Where a point $p \in X$ is an ε-approximate Nearest Neighbour of a query point $q \in X$ if dist(p, q) $\leq$ (1 + ε)dist(p*, q) where p* is the true nearest neighbour. This means that p is within relative error ε of the true nearest neighbour. This is also referred to as "error bound" approximate search [53].
Another solution proposed is a "time-bound" approximate search called Best Bin First which sets a fixed limit to the number of leaf nodes the algorithm searches [6]. This solution has been found to yield better results than the "error bound" search [53].

5.6 Multiple randomised k-d trees

Multiple randomised trees [61] is proposed to improve the original k-d tree algorithm. These randomised trees are searched simultaneously to avoid

the problem of diminishing returns in high dimensions.
The method will create an m number of different k-d trees where each has a different structure so that searching them will be independent. Limiting the number of nodes to be searched to n nodes, will on average result in n/m nodes simultaneously searched in all of the m trees. If one is to achieve independent tree searches, there is not enough to build the trees by selecting random dimensions. One of the solutions is to create the trees with different orientation by rotating the data [61]. The search is then carried out concurrently with a pooled priority queue. After each of the m trees has been ascended once, and each search have found their first potential best match, the nodes are compared across so that every match is ranked both with the other potential matches within each tree and with the nodes in the other trees.

5.7 Best Bin First(BBF)

BBF is a shape-based indexing method suited for a moderate number of dimensions, up to 20. It was created as a solution to improve the k-d tree solution which suffers from a higher dimensionality. It uses feature vectors from an image to rapidly find possible matches from an index structure [6]. Each query looks at a fixed number of bins before ending the search. The bins are explored after their distance from the query point. BBF will on average return 90% correct matches, and the remaining 10% will otherwise be close neighbours.

5.8 Locality Sensitive Hashing (LSH)

Usually, a hash function maps data into distributed space. However, hash functions are exact, designed for precise data structures. Locality Sensitive Hashing [34] takes distance into the calculation. If two points are close together in the original space, it is likely they are close together in the hash structure. Similarly, two descriptors far away from each other in the original space will be likely to be far away from each other in the hash structure.

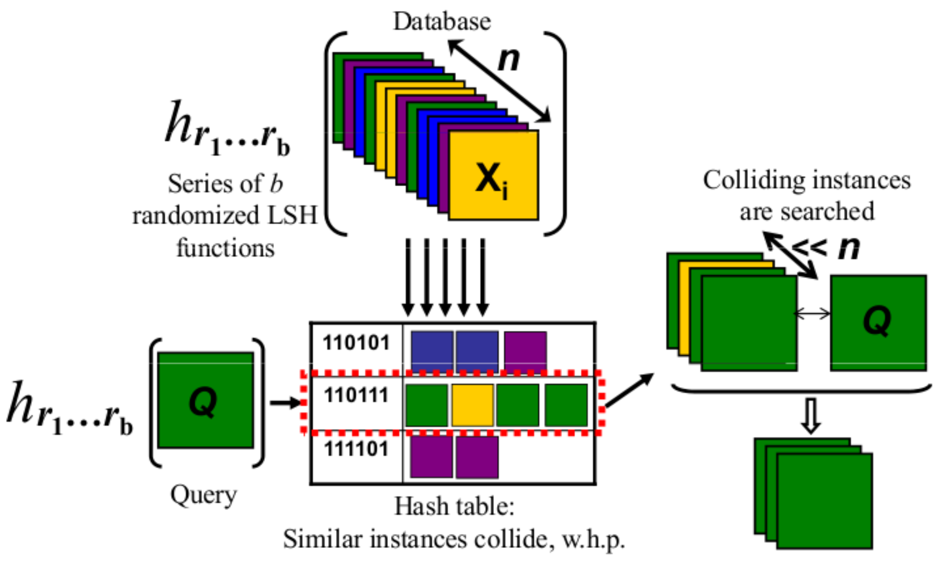
Database
n
$h_{r_1 \dots r_b}$
Series of b randomized LSH functions
X_i
Colliding instances are searched
$<< n$
Q
110101
110111
111101
$h_{r_1 \dots r_b}$
Q
Query
Hash table: Similar instances collide, w.h.p.

Figure 5.3: Illustration of Locality Sensitive Hashing

LSH aim to project the data into a low dimensional binary hamming space. The hash key is each data point mapped to a b-bit vector, and this will potentially allow an approximate Nearest Neighbour search in sub-linear time. By applying n binary hash functions h_1,....,h_n to every object in the database, we will get all the hash keys. All of the hash functions h must satisfy the locality-sensitive hashing property:

$$Pr[h(x_i) = h(x_j)] = sim(x_i, x_j)$$

where $sim(x_i, x_j) \in [0,1]$ is a similarity function. The idea here is that similar point will get the same hash key and be stored together, which means that when a query finds a bucket, it will get a collection of related points. The query time for retrieving (1 + ε) nearest neighbours is bounded by $O(n^{1/(1+\epsilon)})$ for the hamming distance [59], which enables the possibility of speeding up the query time at the cost of accuracy.

5.9 Wavelet-based hashing

Wavelet indexing uses hash tables. This solution uses a Haar wavelet which is a sequence of rescaled square shaped functions that form a wavelet family [24]. A wavelet is a repetitive sequence formed as a wave that increases and decreases from zero to zero. The Haar wavelet is the purest form of a wavelet, and it is not continuous. A three-dimensional lookup table indexes each feature, and the dimensions are the corresponding three first non-zero wavelet coefficients [15]. There are ten overlapping bins per dimension; thus at least half of a bin match each query. A query is exhaustively matched to all features in a query bin and will produce many nearest neighbours. As there is a possibility that the actual nearest neighbour is outside one of the three dimensions, this classifies as an approximate Nearest Neighbour search. The algorithm potentially gives a 125 times speed-up with only a 10% loss of matches assuming an even feature distribution.

5.10 Cascade Hashing

Cascade hashing [16] is a proposed three-layer structure of hashing lookup, hashing remapping and hashing ranking.
The first step is to construct a hashing lookup with short codes to do a coarse search. Each feature is represented in bits made from locality sensitive hashing and sorted in buckets from the bit sequence. When searching a particular feature point from the hash table, all features from the corresponding bucket is returned. The number of bits to represent each feature is freely set, a small number will give large buckets and many potential points both similar and dissimilar. While a large number will reduce the samples in each bucket and reducing the chance for dissimilar points, it will also reduce the number of the best matches to fall into the same bucket thus increase the possibility to miss vital features. This means that the number of bits would have to be tuned to achieve the best possible result.
The next step could be to calculate the distance from all the points returned from the bucket, however, as the calculation part is the most time consuming, and the number of samples is still potentially large, this would not be an optimal solution. Instead, a hashing remapping is introduced, by using hamming distance on a larger amount of bits from the candidates, and then sorting them accordingly.
The final step will then be to create a third hash table which will be constructed from the hamming distance from the query point to all of the features in the other image. Each bucket is sorted after the hamming distance from 0 and upwards. Then all the points from bucket with hamming distance 0 will be extracted, if the number of points is less than the number of points the second step provided, the next bucket with hamming distance 1 will also be used. This part will be repeated until the number of points is equal to the potential points found by the hash remapping. The two best candidates from these are calculated, and the method moves on to the next query point.
The tests done with this implementation has yielded great results giving up to 300 times speed-up compared to the brute force method [16].

5.11 Conclusions

There are obviously many different approaches to handle feature matching, often depending on the size and type of problem. Many of the NN solutions are good in lower dimensions; however, when the dimensionality grows, the search time grows exponentially with it. So to achieve effectiveness in these cases, the Approximate Nearest Neighbour search will improve it a lot compared to the exhaustive search. The SIFT descriptors in openMVG have a 128 dimensionality, which means that the exhaustive search and many of the solutions with tree structures will probably be inefficient implementations. Regarding the dimensionality, it will also be interesting to see how the quality of the matches will be on the cascade hashing and k-d tree solution which we will test our implementations against later on. The k-d tree solution is interesting because the number of dimensions means that it has to sacrifice the number of correct matches or suffer regarding the speedup [53]. For the cascade hashing it will be interesting to see what the dimensionality does for the quality since it at some point uses hamming distance to find matches. Hamming distance does give an indication for equality, but it does not consider the significance of a bit, only if it is wrong or not.

We believe that there is most likely that the exhaustive search would have the biggest improvement potential as this would be a serious bottleneck due to the dimensionality. It will be interesting to see how close we can get the exhaustive search to the approximate search by improving it. Another approach could be to combine some of the different solutions into a new method and see if there is any possibility for improvement through that. In any case, our solution would have to be judged not only by the one we are improving but also against other solutions available.

Chapter 6

Exhaustive search

6.1 Squared Euclidean distance

If we are to improve the exhaustive search we have to look at the calculations for matching the descriptors. As mentioned in section 4.5 on page 34 the matching calculation done in OpenMVG is by default squared Euclidean distance. Euclidean distance gives us the Euclidean length (straight line distance) between two different points in Euclidean space. The Euclidean distance of a vector $v \in \mathbb{R}^n$ is the square root of the summed squared vector elements.

$$d = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

By examining this formula it becomes apparent that it is a straightforward development to the multidimensional problem of the Pythagorean theorem, which can be seen in figure 6.1 on page 46. The algorithm takes two vectors and sums the squared result of each dimension subtracted from each other. The OpenMVG implementation of exhaustive search using Euclidean distance as metric takes each descriptor in one image and calculates the disparity between each of the descriptors in the corresponding image. When the calculation completes, it saves the two best matches and moves to the next descriptor repeating this until all descriptors in the image have two matches in the corresponding image. The reason the best and next best descriptor is saved is that the disparity between them should be of a certain distance. If they are below a distance threshold (The threshold is pre-set to a default value, and can be tuned if needed), they are dismissed as potential matches because they are not unique enough. The complexity of this algorithm consists of n subtractions, n square operations and n sums, which gives us a complexity of $O(3n)$, this means that it has a growth of $O(n)$. This complexity will be the benchmark we will need to go under if we want to improve the efficiency of the matching process.

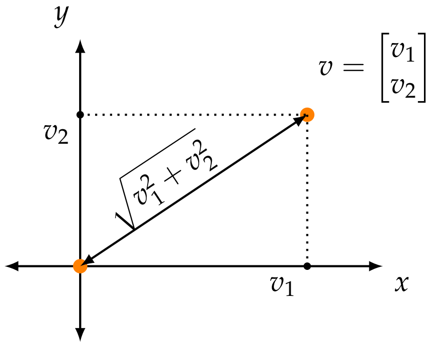

Figure 6.1: Squared Euclidean distance between two points in two dimensional space computed through the Pythagorean theorem

6.2 Linear algebra model

As a basis for our Nearest Neighbour algorithms, we will use vectors to represent the SIFT descriptors. We explain this choice as follows: A vector v is a set of numbers, where each number $v_1, v_2, ..., v_n$ will represent one of the 128 dimensions of the SIFT descriptor. This representation allows our 128-dimensional vectors to represent coordinates of points in 128-dimensional space, and since each component of a SIFT descriptor is a floating point number, i.e. $d_i \in \mathbb{R}$ then v is a floating point vector of dimension 128 which we mathematically express as $v \in \mathbb{R}^{128}$. Representing the SIFT descriptors as a coordinate has a series of benefits as it allows us to experiment with different linear algebra models which may improve the efficiency and correctness of the Nearest Neighbour problem in comparison to the Euclidean distance metric. Especially the dot product, also known as the scalar product, inner product or the projection product, has an interesting effect: It is directly correlated to the angle between two vectors and introduces the concept of vector norm (the length of the vector in linear space). This effect allows us to look at a vector as a linear function and enable us to project the vector representation of the SIFT descriptors against some defined model or subspace, such as a line defined by the vector representation of a different SIFT descriptor.

6.3 Hypersphere

A hypersphere is an entirely round geometrical object with the same mathematical properties as a sphere, but it has an arbitrary amount of dimensions. While a standard sphere relies in three-dimensional Euclidean space, a hypersphere relies in N-dimensional Euclidean space. One of the main properties of a sphere is its equal distance from the centre to all edges. This property is deduced from the mathematical definition of a sphere. A hypersphere is visualised in figure 6.2 on page 47 and is defined as follows: the set of points that are all at the same distance r from a given point in

N-dimensional space [1]. We represent normalised SIFT descriptors as points on a hypersphere. As a SIFT descriptor has 128 dimensions, the hypersphere on which we project the descriptors will be 128 dimensional as well.

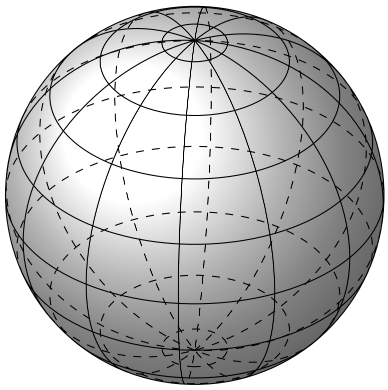

Figure 6.2: Hypersphere illustrated in three dimensions

6.4 Arc length

Considering the hypersphere representation of SIFT descriptors mentioned above, we can find the disparity between descriptors by looking at the arc length between the points representing them on the hypersphere. We compute the arc length through the angle between the points. Our thoughts were that by calculating the arc length between two points, we would have another method to figure how close to each other two descriptors were, which would hopefully achieve an accuracy that exceeded the measurement done by the Euclidean distance metric as well as performance improvements. Below is an explanation of the calculation. A single vector consist of n coordinates:

$$\overrightarrow{u} = [x_1, x_2, x_3 ... x_n]$$

The length of the vector is retrieved by taking the square root of the sum of all coordinates squared:

$$\| \overrightarrow{v} \| = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{\overrightarrow{v} \cdot \overrightarrow{v}}$$

This results in n square operations and n sum operation, which gives 2n calculations in total.

Dot product between the vectors is the sum of each corresponding coordinate multiplied with each other:

$$\overrightarrow{u} \cdot \overrightarrow{v} = \sum_{i=1}^{n} x_{ui} \cdot x_{vi}$$

Again this gives us 2n calculations as there is n multiplications and n sums.

The angle is retrieved by taking the dot product of the vectors divided by each vectors length multiplied with each other:

$$\theta = \cos^{-1}\left(\frac{(\overrightarrow{u} \cdot \overrightarrow{v})}{(\| \overrightarrow{u} \| \cdot \| \overrightarrow{v} \|)}\right)$$

Which gives us the arc length computation:

$$Arclength = \frac{\theta^\circ}{360^\circ \cdot 2\pi r}$$

Complexity-wise this gives us 4n for the product of the length of the vectors, and 2n for the dot product of the vectors. Ignoring the last calculations which are constant regardless of the size of n, we get $O(6n)$. Thus, we perform six operations times the descriptor size for each calculation. This algorithm complexity suggests that the arc length metric is not potentially faster than the Euclidean distance metric, so the only thing we can achieve is higher quality on the calculation itself and hopefully not lose too much performance.

6.4.1 Implementation

Since we only change the calculation on this part, we could keep the general set-up and only change the calculation itself. We declare three variables to hold the intermediate results, one for the dot product between the descriptors and two for the dot product of each descriptor. Further, the inverse cosine function is computed based on the dot product divided by the product of the two length calculations, before multiplying it with 2π. The result is then returned and the main part will use it as if it was done by Euclidean distance to compute the distance threshold. The initial results for this method was actually quite positive, it seemed like the run time was more or less stable, and the matches provided seemed to be more precise in comparison to the Euclidean distance metric as there were fewer outliers. However, if the number of features increased the solutions proved to be less effective and not a good solution overall. We will not go deeply into the results for now, as we will compare our different solutions later in the thesis.

6.4.2 Comparing arc length to squared Euclidean distance

The arch length solutions proved to be slower in theory, and that was also the case when testing it. But it did provide some changes to the matches, removing some of the outliers that we experienced with Euclidean distance. This suggest that the solution is more precise and that there may be uses for it in some cases. However, upon further testing we understood that the reduction of false matches was due to the increased distance threshold from measuring along an arc instead of a direct line. When we adjusted for this, the matching results were the exact same. This can be proved through the following observations for the l_2 normalized vectors $\overrightarrow{v}$, $\overrightarrow{w}$ which have length one:

$$\| \overrightarrow{v} \| = 1, \text{ and } \| \overrightarrow{w} \| = 1$$

We can show that the Euclidean distance between these two vectors is proportional to the arc length between them in the following manner [69]:

$$\begin{aligned}
\| \overrightarrow{v} - \overrightarrow{w} \|^2 &= \sum_{n=1}^{128} (v_n - w_n)(v_n - w_n) \\
&= \sum_{n=1}^{128} v_n^2 - 2 v_n w_n + w_n^2 \\
&= 2 - \sum_{n=1}^{128} 2 v_n w_n \\
&= 2 - \sum_{n=1}^{128} 2 cos\angle(v_n, w_n) \\
&= 2(1 - \sum_{n=1}^{128} cos\angle(v_n, w_n)
\end{aligned}$$

It is straightforward to see that Euclidean distance and arc length are mathematically proportional as it is trivial to convert between the Euclidean distance and the angle used to calculate an arc length. The two calculations are not equal due to the square operation; however, as squaring is an operation that preserves the order for a set of positive numbers, this can be ignored. Another problem was due to the initial test programs use of a relatively small number of descriptors. When we increased the data size, we experienced that the runtime suffered more and more in comparison to the Euclidean distance metric.

6.4.3 Arc length improvement

Since we had concluded that the algorithm did not have any advantages considering the number of operations, the next step was to see if there were any unnecessary calculations in our implementation. Since the vectors we are working with are normalised to unit length, calculating the length of them should not be necessary to get the correct result from the algorithm. Therefore, we can ignore the calculation that finds the length of each vector and use a constant number instead. By doing this, we can change the complexity of the algorithm from $O(6n)$ to $O(2n)$ which would be potentially better than the original implementation. When implementing this on the unsigned char representation of the SIFT descriptor discussed in section 2.2.5 on page 11, we noticed that due to loss of precision during the conversion, the new vector length was between 507 and 508 instead of the expected 512. Therefore we used this constant as vector length and got identical matching results. The remaining bottleneck with this implementation was the expensive inverse cosine operation. One way to improve this is to create an inverse cosine lookup table; we decided against doing this in a CPU implementation for memory efficiency purposes.

6.5 Dot product

As discussed in section 6.2 on page 46 the dot product on R^n is directly associated with the concept of an angle between two vectors. We can see this association from the following example: Given two vectors x and y each representing points in a Cartesian coordinate system, we can draw lines from the centre of the coordinate system the points specified by each of the vectors to create an angle. Further, we can create a triangle by drawing a third line between the two points. The dot product between two vectors by its algebraic definition is written

$$\overrightarrow{v} \cdot \overrightarrow{w}$$

which by its geometric definition is equivalent to

$$|\overrightarrow{v}||\overrightarrow{w}|\cos\theta$$

where $|\overrightarrow{v}|$ is the length of $\overrightarrow{v}$ and θ is the angle between $\overrightarrow{v}$ and $\overrightarrow{w}$. If the dot product is negative, θ is larger than 90° telling us that the two vectors are far apart from each other. When θ is 90°, the dot product is zero since $\cos 90° = 0$.

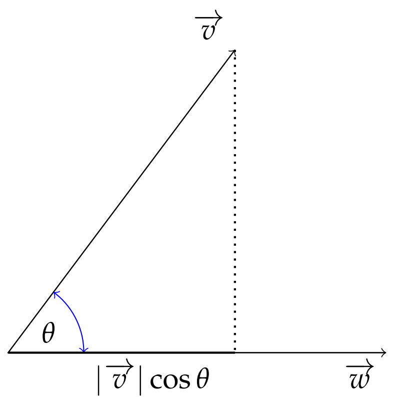

Figure 6.3: Projection found by the geometric definition of the dot product

This means that when the dot product is positive, θ is less than 90°, and it will increase as θ decreases. When the two vectors are aligned, the maximum value of the dot product is achieved. Therefore, by calculating the dot product between our descriptors, we can find the best matches by keeping the ones that have the highest value after the calculation, and achieving the same distribution as the Euclidean distance would provide.

The pipeline for the dot product calculation is illustrated in figure 6.5 on page 53.

Euclidean distance has 3n operations for each calculation. Since each descriptor consists of 128 dimensions, the total amount of operations with squared Euclidean distance will be 128 subtractions, 128 multiplications and 128 sums, which is 384 operations. Since we do not need the exact distance between the vectors, but rather finding the ones that are closest to each other before doing the calculation, we can instead calculate the dot product between the vectors, since it gives us an angular relationship between the two vectors. Thus, we will still be able to find the two best matches from each descriptor, and we will do it with fewer operations than if we used the squared Euclidean distance formula. We compute the dot product in the following manner:

$$d = \sum_{i=1}^{n} (x_i \cdot y_i)$$

This formula takes 2n operations which will result in 128 multiplications and 128 sums, the total being 256 operations which are 128 operations fewer than the original implementation. It does perhaps not sound like a big deal, but since this part is done m*n times for each m descriptors in the left image and n descriptors in the right image, we will save many calculations as the number of descriptors increases. To avoid complication with the other parts of the program, we do not return the dot product result. Instead, we find the two best matches from our calculation and return the result after doing squared Euclidean distance from these two. Hence we will improve the runtime to find the matches at the same time we will still be able to use the original implementation for calculating the difference between them. This solution should provide an improved run-time since the number of calculations will decrease. By our accounts, it is a great chance that it will also improve more as the data set increases, as the number of calculations we save will increase as well.

6.5.1 CPU Implementation

The first natural step for us to take with this idea was to implement our solution in the CPU based part of the program. Although the best implementation probably would be on the GPU, it is sensible to start off with the CPU, making sure that the implementation work and see the advantages it gives regarding run-time and if the quality will suffer since we do a different computation. Implementing a new line of code can be done in many ways, our approach was to do small incremental changes and verify them step by step. We did this to ensure that our changes would not impact the program in the wrong way, and simplify elimination of errors.

What we knew for sure was that the result of the algorithm still should provide two matches we could compute the distance between the best and

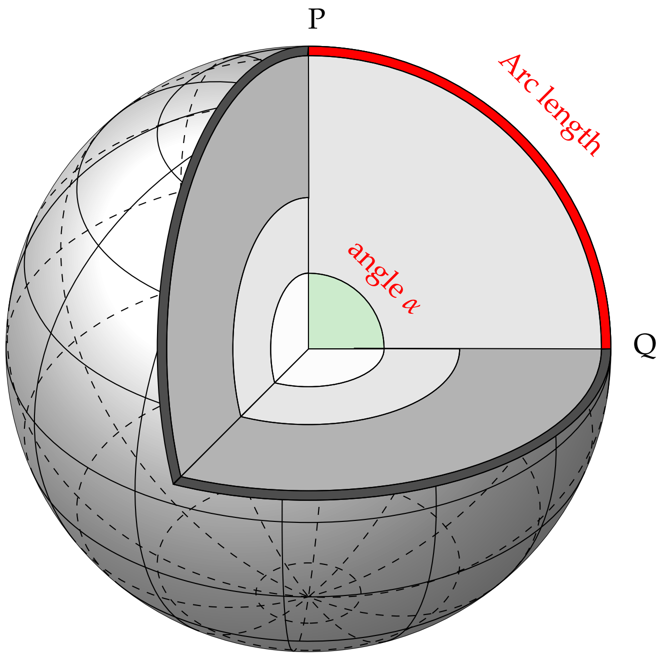

Figure 6.4: Angle and arc length between two points on a three dimensional hypersphere

second best match. We compute this distance with the Euclidean distance metric in the last part of the method from the two best descriptor indexes. Then we looked at how we get the two descriptors, we knew that we needed to search through all the descriptors from the second image, so we set up a standard iteration of each descriptor in the first image, and then we iterated through all descriptors in the corresponding image in a standard exhaustive search pattern. Further, we required the calculation of the dot product between these descriptors.

In our first implementation we iterated through every dimension in the descriptors one at a time, however, by loop enrolling, calculating four dimensions each iteration, we could minimise the overall runtime of this part. Therefore we iterate four dimensions at a time, saving the result of four multiplications from each descriptor, and summarise them when all 128 dimensions have been measured, before jumping to the next descriptor. When we have a complete calculation of a descriptor against our query point, we check if the sum of them is larger than the previous best result. If it is, we save the index and the value as the new best result. We do this both for the best match and the second best match, making sure we always have control over the two best potential matches. After all descriptors in the second image have been checked, we provide the two indexes to the Euclidean distance calculation to check if the distance is above the required threshold, and continue to the next descriptor in the left image. This process repeats until we have compared all of the descriptors in the left image with all descriptors in the right image.

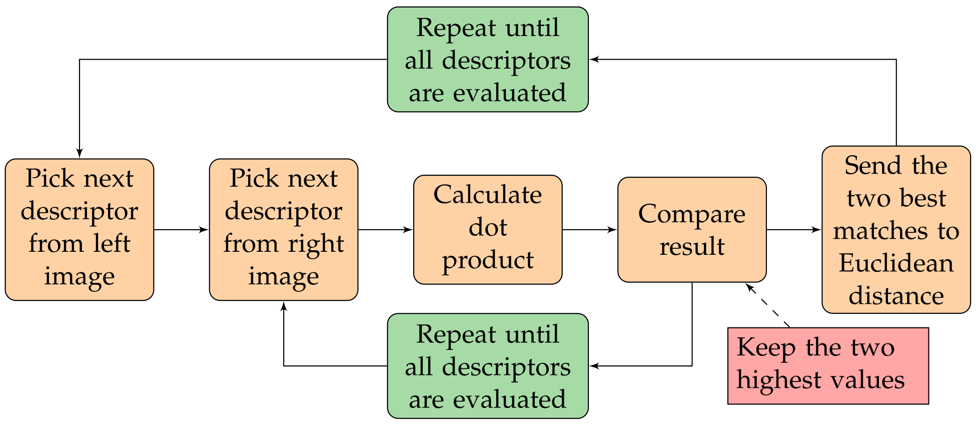
Repeat until all descriptors are evaluated
Pick next descriptor from left image
Pick next descriptor from right image
Calculate dot product
Compare result
Send the two best matches to Euclidean distance
Keep the two highest values
Repeat until all descriptors are evaluated

Figure 6.5: Flowchart of exhaustive search using dot product

The results from this solution was encouraging, the run time had a significant improvement, and we couldn't see any negative impact on the quality of the results. The improvement was seen both on a small scale as well as on a large scale of input. So we quickly decided that we also wanted to do our implementation on the GPU part to see if we could achieve the same improvement.

6.5.2 GPU Implementation

As was the case for the CPU implementation, we did this part in small iterations to avoid complications on the way. The set-up on the GPU is quite different to the CPU implementation, and the room for errors is bigger as we need to consider race conditions. However, the principle is the same; we want to compute the Euclidean distance only two times per run to calculate the distance threshold between the nearest and second nearest neighbour. We will find these two descriptors through the dot product computation.

The GPU implementation is done in PopSIFT, as PopSIFT had a more proper set-up for our problem than what we experienced in openMVG. Largely because the descriptors already exist in GPU memory, thus no memory copy from CPU to GPU is needed. Our implementation in PopSift runs with one block and 32 threads for each descriptor in the query set. Doing this, we take advantage of the GPUs ability to run threads and blocks in parallel. The choice of 32 threads per block is due to the warp size, which we utilise for efficient thread communication. We also implemented this algorithm for 64/128/256 threads per block, but as the general process is the exact same for all sizes, we only explain the 32 thread version. The only difference is that the number of blocks is divided by the number of warps.

We compute the dot product in two basic steps. First, the 128 floating point values are distributed between the 32 threads within a block, giving each thread four floating point values from its query descriptor. Then all the threads iterate over all the descriptors in the database set, calling a function that calculates the dot product between the query descriptor and each of the descriptors in the database set. In this function, each thread computes the dot product of its four floating point values with the corresponding four floating point values in the database descriptor. We then have 32 values, which is added together in thread zero using the shuffle_down operation available within a warp. Thread zero will then update the best and second best result by comparing the new result with the previous best and second best. We repeat this procedure for all descriptors in the query set. When we have compared a query point to all the descriptors in the database set, we compute the Euclidean distance on the two best potential matches to compute the distance threshold before adding them to the match set. This computation is again computed by 32 threads in parallel, in a similar manner as the dot product. Since we assign each query point to a block, this operation is concurrent.

GPU implementation Conclusions

The dot product method proved to be faster on the GPU as well, with the running time cut down by about $\frac{2}{3}$ over the Euclidean distance search already implemented. We were pleased with this result, as it proved that not only the algorithm in itself was better as was seen on the CPU, but our implementation, in general, was also better as we reduced the running time by more than the algorithmic improvements would suggest.

6.5.3 Comparing Squared Euclidean distance to dot product

By changing the metric from the Euclidean distance to the dot product calculation, we have achieved an exhaustive search that is faster than previous solutions without diminishing the results. Since the complexity of the dot product is $O(2n)$ versus $O(3n)$ for Euclidean distance, there will always be n fewer operations with this implementation regardless of the size of the input. Thus, the larger the data sets provided is, the better the dot product will perform against Euclidean distance, making it a preferred algorithm to use as a metric. As we still use the Euclidean distance metric for the last calculations, it is still a relevant algorithm in this type of work even though we could completely remove it through the use of an inverse cosine table.
The dot product can also improve some approximate searches if they do calculations on a relatively large number of descriptors. Alternatively, it can make some of them more precise by giving them the option of doing more calculations than previously without losing performance.

6.6 Results

The tests are not to be taken literally, as there will be different results based on external factors, but they will say something about our solutions nevertheless. In the CPU test, we include running-time comparisons to two approximate searches: Cascade hashing [16], and randomised k-d trees [53]. For the GPU tests, FLANN [53] provides a CUDA version of the ANN randomised k-d trees algorithm. However, this solution only supports three dimensions [68]. Although it would be possible to perform a dimension reduction, we choose not to do this as it would significantly reduce the matching quality. Instead, we will use the CPU version of randomised k-d trees provided by FLANN [53] as a comparison.

6.6.1 CPU INFO

All CPU experiments were computed with the specifications provided in table 6.1 on page 55.

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	8
On-line CPU(s) list	0-7
Thread(s) per core	2
Core(s) per socket	4
Socket(s)	1
NUMA node(s)	1
Vendor ID	GenuineIntel
CPU family	6
Model	158
Model name	Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
Stepping	9
CPU MHz	4200.000
CPU max MHz	4500,0000
CPU min MHz	800,0000
BogoMIPS	8400.00
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	8192K
NUMA node0 CPU(s)	0-7

Table 6.1: CPU information

6.6.2 Test data

Our tests are done with three image pairs with different size so we could see where our solutions would have the best impact. The first image pair has a low resolution which is below the recommended standard for openMVG. The second pair is in HD resolution which is the recommended standard for openMVG, and the third pair is in UHD resolution. The exact resolution and number of features in each image can be seen in table 6.2 on page 56

Image	Size	Features OpenMVG	Features PopSift
Ace_0	500x500	477	1448
Ace_1	640x480	641	2449
Col_0	1920x1080	5028	25018
Col_1	1920x1080	5556	27517
NY_0	3264x1836	12269	60237
NY_1	3264x1836	10130	48438

Table 6.2: Test data

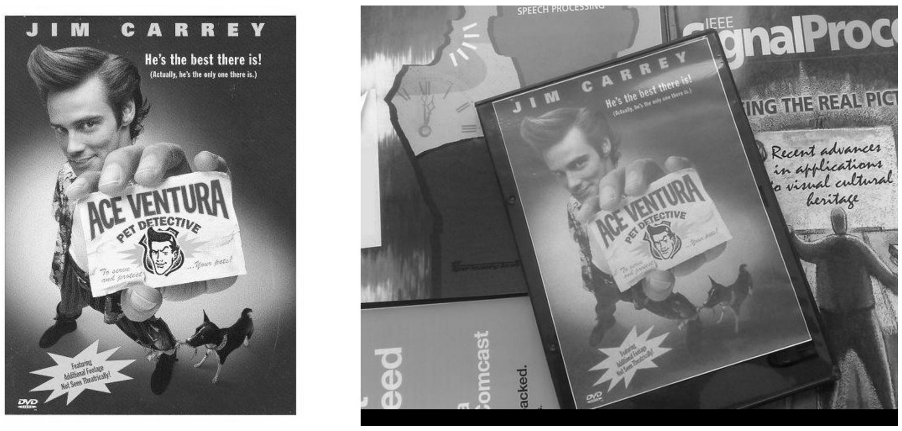

Figure 6.6: Picture set Ace

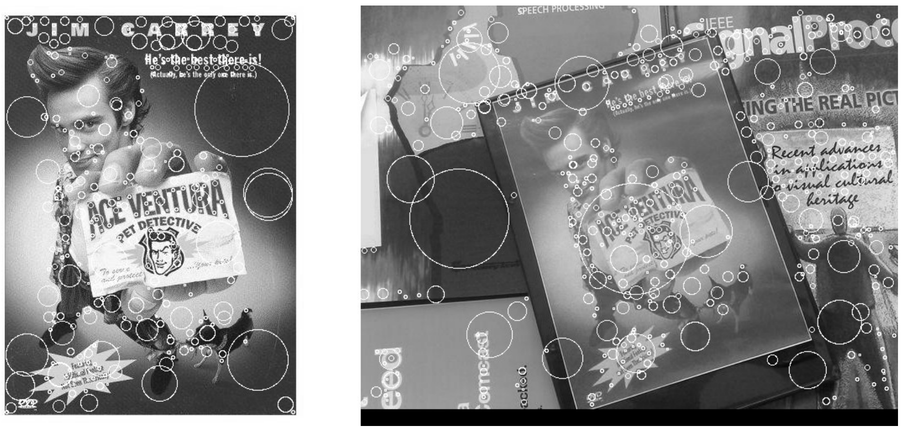

Figure 6.7: Features in image set Ace

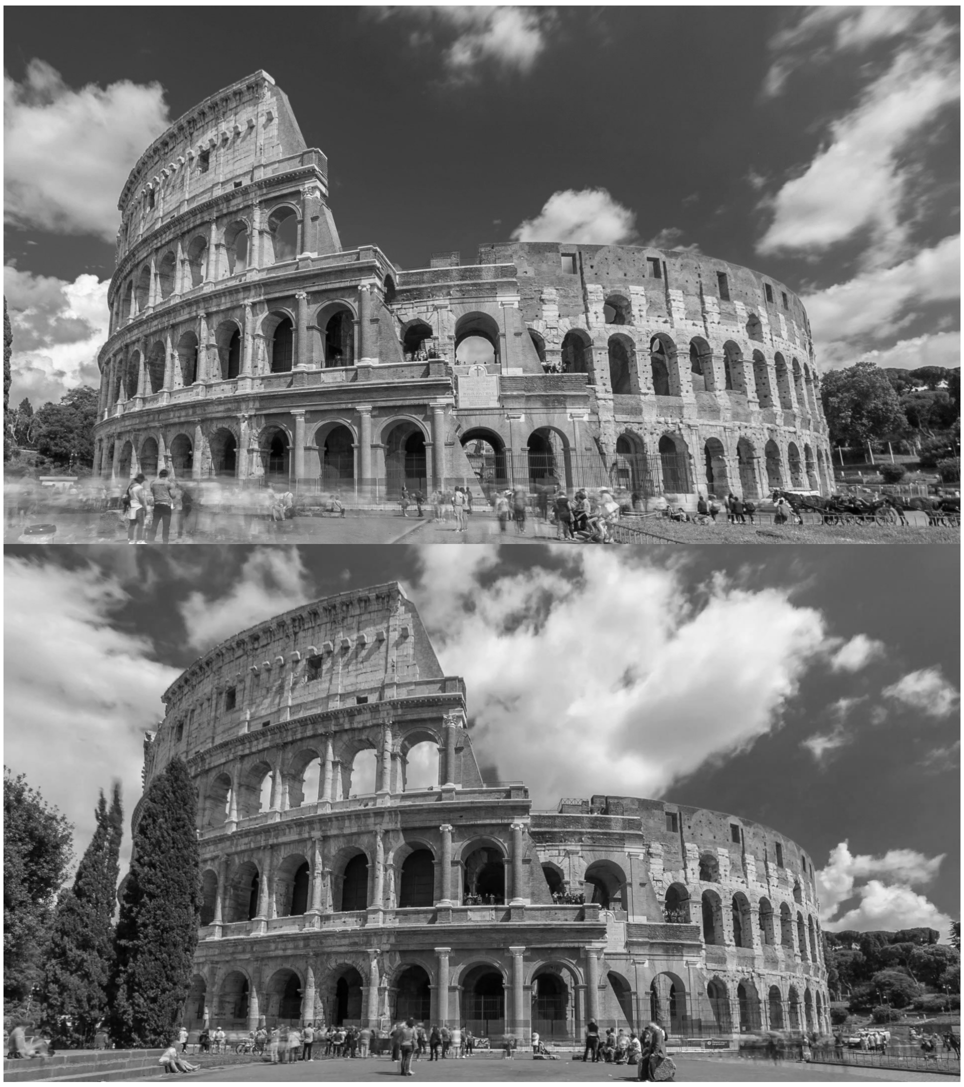

Figure 6.8: Picture set Col

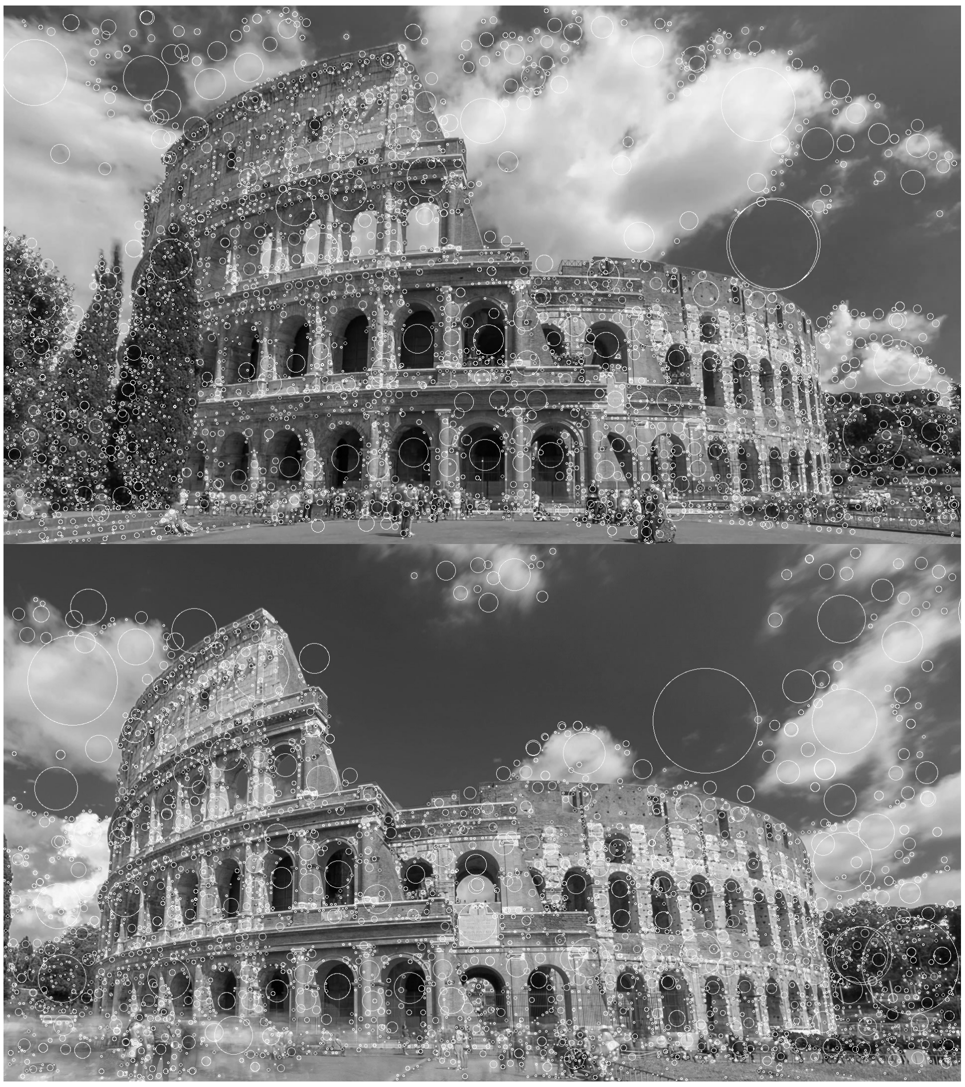
Figure 6.9: Features in image set Colosseum

New York

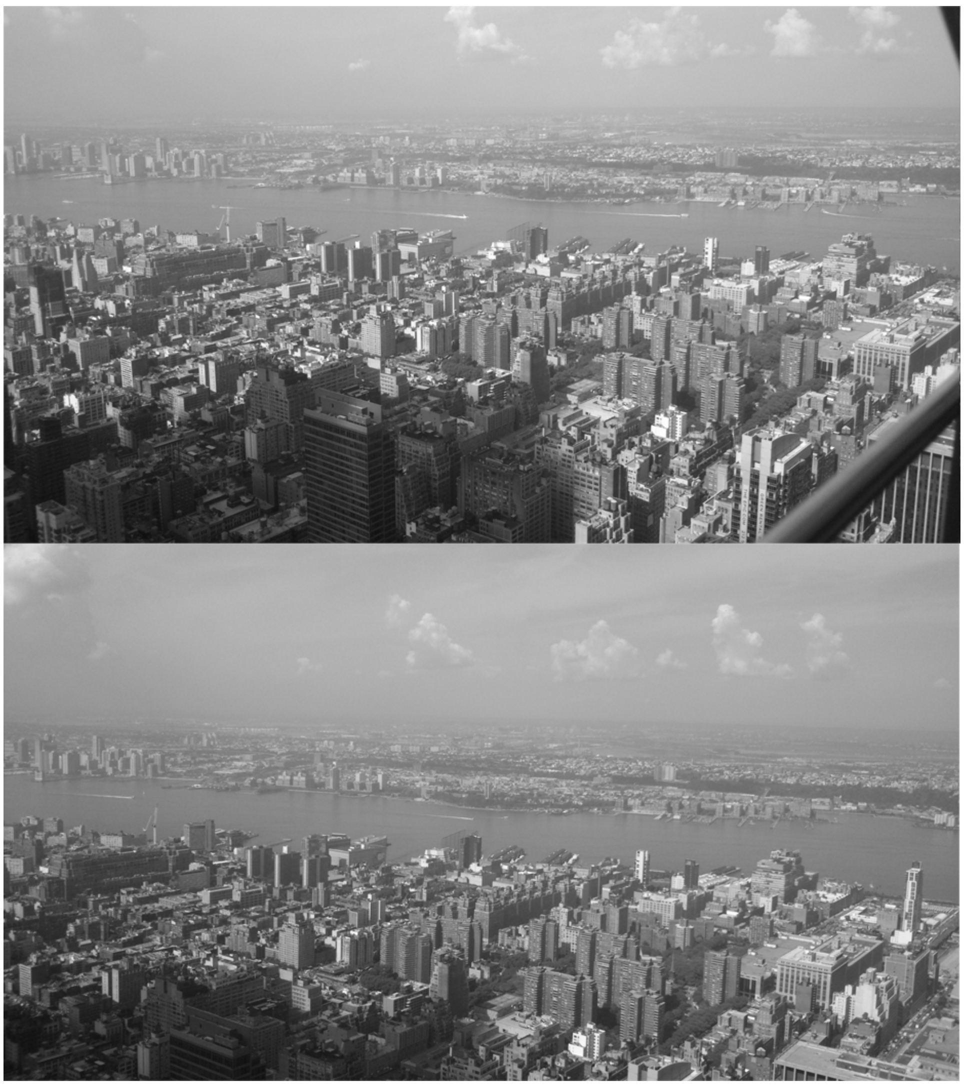
Figure 6.10: Picture set NY

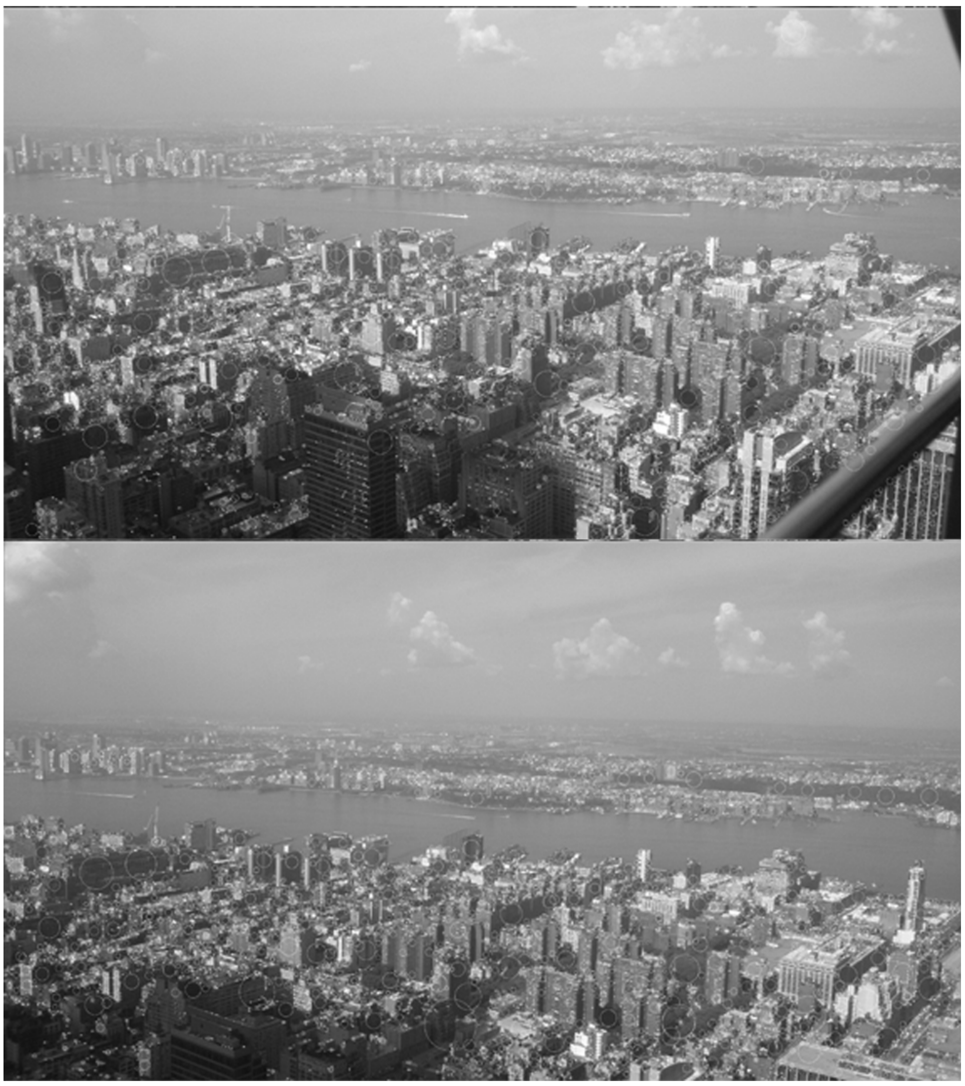
Figure 6.11: Features in image set NY

6.6.3 CPU tests

Metric	Matches	Runtime	Speedup
Euclidean distance	122	32 ms	1x
Arc length	109	29 ms	1.1x
Dot product	122	26 ms	1.2x
ANN k-d tree	123	60 ms	0.5x
Cascade	126	10 ms	3x

Table 6.3: CPU test on Ace

Table 6.3 on page 60 shows that there are no major differences between the methods regarding both runtime and the number of matches. The Cascade hashing has the most noticeable speedup; however,it has more false matches compared to the other methods, and it is not necessarily the best solution for images with low resolution. Our arc length and dot product solution is slightly faster than the original Euclidean distance, and the matches they provide are satisfying, making them more or less equal. There is no good reason to use one over the other, and if the image data consists of low-resolution images, the standard solution will work fine.

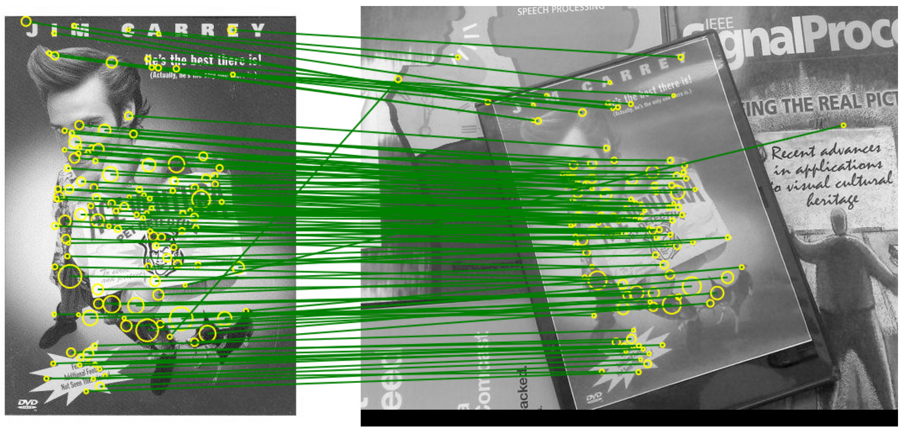

Figure 6.12: Matches from Euclidean distance

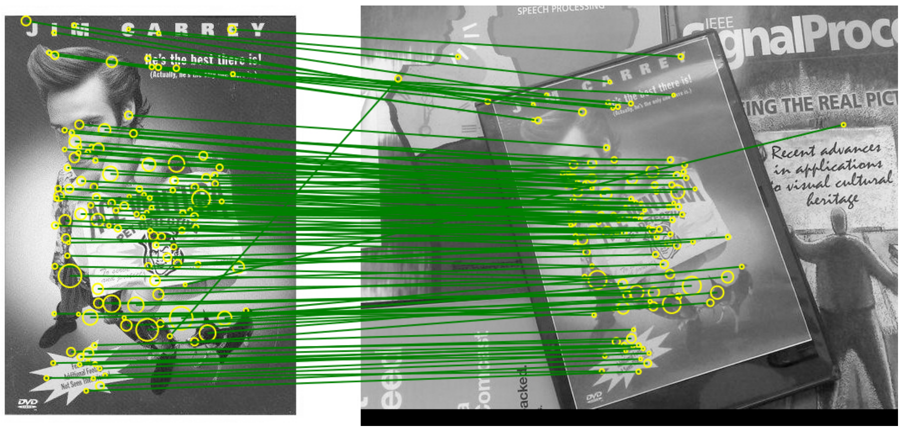

Figure 6.13: Matches from dot product

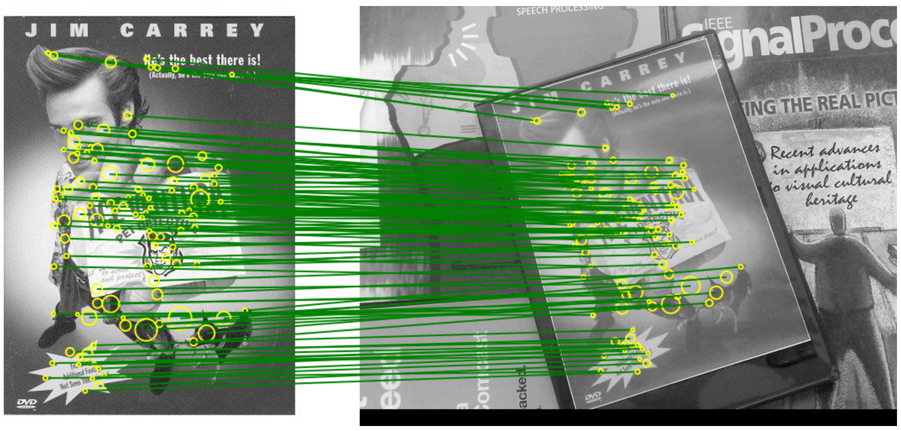

Figure 6.14: Matches from arc length

Metric	Matches	Runtime	Speedup
Euclidean distance	540	3363 ms	1x
Arc length	286	1899 ms	1.8x
Dot product	540	760 ms	4.5x
ANN k-d tree	559	234 ms	14x
Cascade	590	83 ms	40x

Table 6.4: CPU test on Col

The Colosseum test shown in table 6.4 on page 62 provides some more interesting data. All the methods perform faster than Euclidean distance, but there is more disparity in the matches provided. Again the Cascade hashing is by far the most efficient method with potentially 40 times speedup, but the number of matches provided differ from the exhaustive searches. The k-d tree solution is faster than the original implementation with up to 14 times speedup. The dot product solution which has the same matches as the original implementation is 4.5 times faster, and are a good alternative if the quality of the matches matters. The arc length solution has significantly fewer matches, however, as we discussed earlier, the arc length solution needs tuning of the distance threshold to get all the matches, and we did not do this on the test sets as this would not change the performance noticeably.

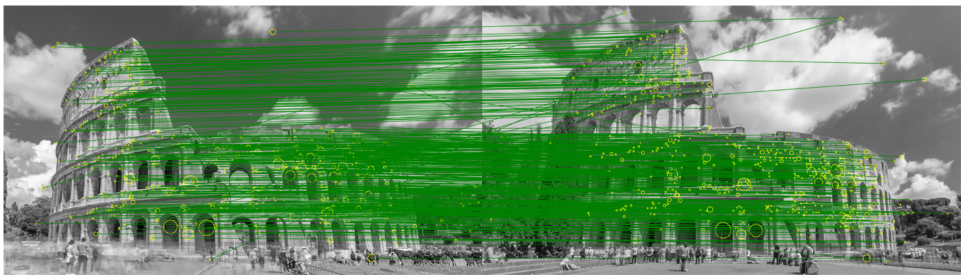

Figure 6.15: Colosseum euclidean distance

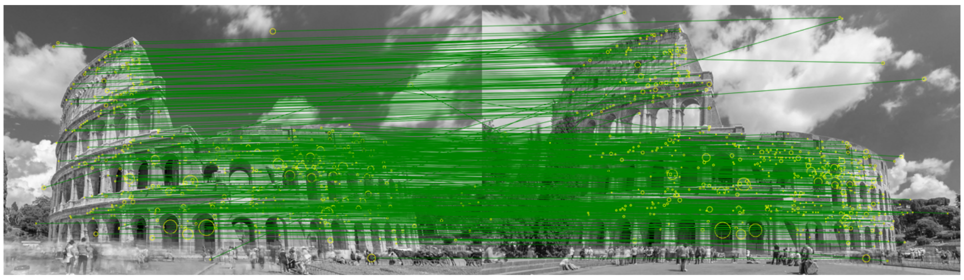
Figure 6.16: Colosseum dot

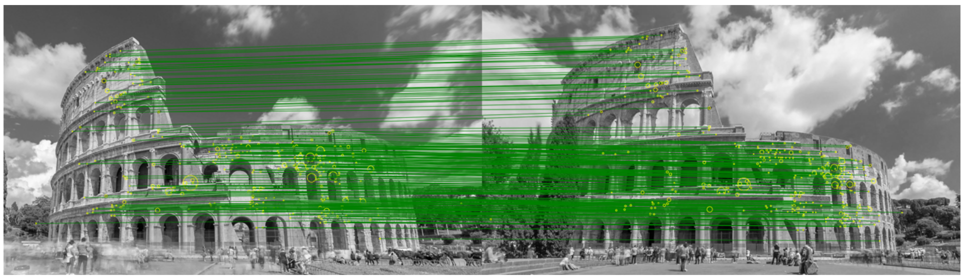
Figure 6.17: Colosseum arc

Metric	Matches	Runtime	Speedup
Euclidean distance	4166	14980 ms	1x
Arc length	3544	8431 ms	1.8x
Dot product	4168	3338 ms	4.5x
ANN k-d tree	4159	481 ms	31x
Cascade	3772	183ms	82x

Table 6.5: CPU test on NY

The final test on the CPU implementation with the New York image set show in table 6.5 on page 63 describes a similar pattern as on the Colosseum test. The speedup for our dot product and arc length solutions are the same as before while the approximate searches provide an even greater speedup. Thus, as the number of image points increases, the efficiency of the exhaustive search will deteriorate in comparison to the approximate searches.

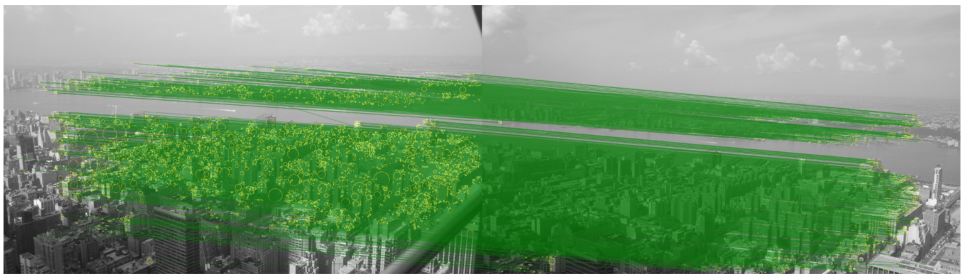
Figure 6.18: brute NY

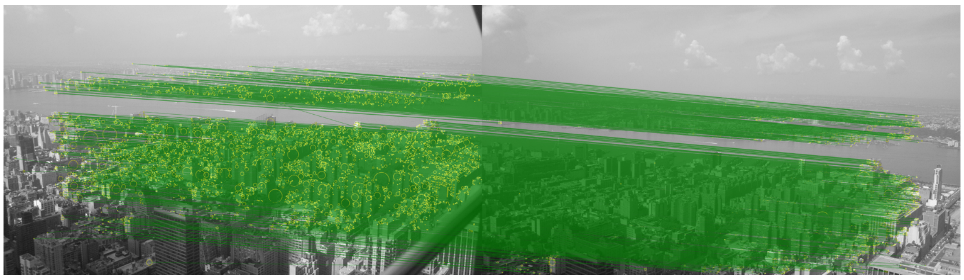

Figure 6.19: dot NY

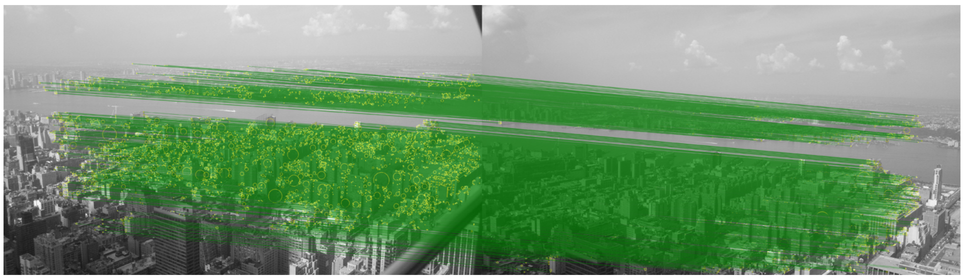

Figure 6.20: arc NY

	Ace Ventura	Colosseum	New York
Arc length	1.1x	1.8x	1.8x
Dot product	1.2x	4.5x	4.5x
ANN k-d tree	0.5x	14x	31x
Cascade	3x	40x	82x

Table 6.6: CPU speedup on the different image sets

The table 6.6 on page 64 shows how well the different solutions do on the different image sets. As the resolution on the images increases, the exhaustive searches improves the runtime to a point before stagnating around full HD resolution. The approximate searches continue to improve even after the resolution surpasses 1920x1080 and yields a much better speedup when it comes to images with a significant resolution.

6.6.4 GPU tests

Metric	Matches	Runtime	Speedup
Euclidean distance	752	3.627 ms	1x
Dot product	752	3.606 ms	1x
ANN k-d tree*	755	118 ms	0.03x

Table 6.7: GPU test on Ace
Done with NVIDIA Visual Profiler
*ANN k-d tree done on CPU

Metric	Matches	Runtime	Speedup
Euclidean distance	5550	519 ms	1x
Dot product	5550	502 ms	1x
ANN k-d tree*	5970	1997 ms	0.3x

Table 6.8: GPU test on Col
Done with NVIDIA Visual Profiler
*ANN k-d tree done on CPU

Metric	Matches	Runtime	Speedup
Euclidean distance	24999	3173 ms	1x
Dot product	24999	2022 ms	1.6x
ANN k-d tree*	25668	4864 ms	0.7x

Table 6.9: GPU test on NY
Done with NVIDIA Visual Profiler
*ANN k-d tree done on CPU

The GPU tests on table 6.7, table 6.8 and table 6.9 show us that the dot product is performing better than the Euclidean distance, but there is no evident speedup on the smaller images. It does have some speedup on the New York image set, and since it has the same matches as the Euclidean distance, it can replace it on any image set without any drawbacks.
The tests shows that both the exhaustive searches compared to the ANN k-d tree is significantly faster on both the Ace image set and the Colosseum image set. On the New York image set, the difference is not as substantial, but both GPU versions of the exhaustive search is still better. In any case, the GPU solution using the dot product is the best option since it is significantly faster and 100% correct.

Thread count	Runtime	Occupancy
32t	2107 ms	49.6
64t	2022 ms	99.9
128t	2092 ms	99.9
256t	2166 ms	99.9

Table 6.10: Optimizing the dot products number of threads per block GPU test on NY
Done with NVIDIA Visual Profiler

One of the things we did to optimise the dot product was to implement it with 32, 64, 128 and 256 threads, to see what would give us optimal performance. We did this because the profiling results pointed to an achieved occupancy of 49.6 in our 32 thread implementation which we thought we would be able to improve by increasing the number of warps per thread block. Table 6.10 on page 65 shows the number of threads, execution time and occupancy for the dot product on image set NY. The 64 thread version proves itself as the winner with the lowest running time and 100% occupancy.

6.6.5 Conclusion

The CPU tests give us an indication that the arc length implementation performs slightly better in comparison to the Euclidean distance metric with regards to the runtime. The dot product is faster than both these implementations, and it is a lot faster for the larger images. The Approximate searches are quite similar for smaller images, but they are much faster on the larger inputs. The precision of the matches are not as good since the number of matches varies from the exhaustive search, the k-d tree solution is not far from the original implementation, while the cascade hasher has a high percentage of false matches and are not a viable option. If the input consists of standard HD images and the guaranty for correct matches is needed, the dot product solution should be an excellent alternative to the approximate searches as it performs well enough and supplies the same matches as the Euclidean distance does.

The GPU tests show that if there are input images with low resolution, there is no significant difference if we are using the Euclidean distance metric compared to the dot product. Larger images show us that there is a difference and that the dot product will enhance the program. There is evident that the size of the images or more precisely the number of features has a big say on the algorithms on a CPU. If there is a moderate number of features, usually because the images has a low resolution, the exhaustive searches often has a adjacent runtime to the approximate searches. As the number of features increases, the gap between the exhaustive searches and the approximate searches grows and the speedup they give will often weigh heavier than the drawbacks they give regarding the precision. On the GPU, our dot product solution is faster than the other solutions both compared to the CPU and the GPU solutions. It is also 100% correct, thus the best solution regardless of the data set in this setting.

Chapter 7

Approximate nearest neighbour searches

7.1 The need for speed

We managed to improve the exhaustive Nearest Neighbour search, and it should be a viable solution in many scenarios. However, there is still a need for even faster solutions, and there is a limit to how far we could take the exhaustive search. So our next focus would be to change the search method itself, and try to make a more effective way of doing it, and still be able to provide an acceptable number of matches. There are several algorithms for computing an Approximate Nearest Neighbour [6, 8, 16, 34, 61], however, the fastest algorithms sacrifice much precision to get a fast running time. We will attempt to develop an algorithm which surpasses these algorithms in speed an precision. The optimal solution would give a 100% number of matches, but we are prepared to lose some precision if it could improve the overall runtime drastically.

7.2 Tree structure of transposed descriptors

Our proposed algorithm for an Approximate Nearest Neighbor search makes use of transposed descriptors to quickly discard different descriptors through a coarse comparison of the most significant bits. The algorithm takes advantage of the 128 dimensions of the SIFT descriptor for fast matching. The flow of this approximate search is illustrated in figure 7.1 on page 68.

7.2.1 Memory layout

The SIFT descriptor is laid out in memory either as 128 consecutive single precision floating point numbers(32 bit) as the most precise representation, or 128 consecutive unsigned chars in the less precise but more memory efficient representation. All dimensions have equal weight. Let us take the descriptor consisting of floating point numbers as an example. This descriptor consists of 512 consecutive bytes. If we instead visualise this as a two dimensional 128 * 4-byte matrix where the floating point numbers are laid out column-wise in memory, we would be able to iterate through a descriptor in such a way where we first see the bytes which have the strongest impact on the size of the number first. We continue this abstraction and visualise our descriptor as a 128 * 32-bit matrix where every float is laid out column-wise with one bit in each depth, and we can iterate through the descriptor looking at 128 bits at a time which will be the most significant bits, then the second most significant bits, etcetera.

This method will function in the same way for the unsigned char representation of the SIFT descriptor, the only difference being that our descriptor would be a smaller 128 * 8-bit matrix instead. With this abstraction, we will be able to detect fast whether two descriptors are a potential match for each other and otherwise discard without making a thorough comparison. It also gives us the ability to sort the descriptors in a purposeful matter, as the descriptor is represented as a large integer. This sorting will allow us to preprocess our data, creating bins of descriptors in which a descriptor is either highly likely or highly unlikely to find its match. We will use this to drastically cut down the number of descriptors we need to search to find the nearest neighbour.

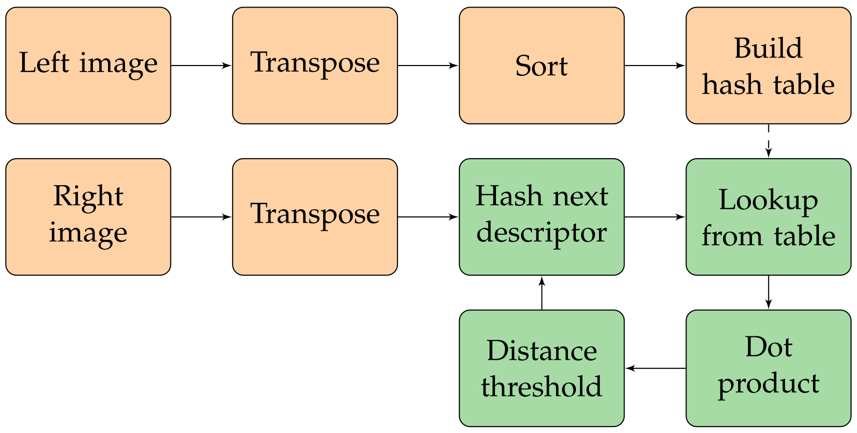

Figure 7.1: Flowchart of approximate search by transposing memory layout

7.2.2 Bitwise transpose

To convert our descriptor into a transposed bit matrix sorted by bit significance, we can use bitwise operations. What is needed is a bitwise transposition of the data. We have implemented several algorithms that perform this bitwise transpose targeting CPU and GPU architectures as well as the two different descriptor representations. However, the general concept of the bitwise transpose stays the same. We will start by going over the serial bitwise transpose of the 128x32 bit matrix on the CPU. As we do the transposition of the 128x8 bit matrix by 16 8x8 transpositions also performed for the 128x32 bit matrix, both transposition procedures will be clear from the explanation. We will later discuss solutions optimized for the GPU in section 7.2.15 on page 75.

7.2.3 Detailed explanation of fast binary transpose of 128 * 32 bit matrix

The way we choose to do this is through four 32x32 bit transpositions, and then a final function to organize the data in such a way that we get one 128 * 32 bit matrix instead of four 32 * 32. The naive way to transposing a bit matrix would be to mask and shift one bit at the time, and store the current shift value in a temporary swap variable. This approach would be very slow, as it does not take into account any knowledge about the matrix structure. What would be a better solution would be to first swap large 16*16 blocks, then 8*8 blocks, and so on dividing the size by two each time until we hit one [67]. Organizing the 32 * 32 bit blocks with each other is simply a manner of swapping specific bytes in the pattern within each 32 * 32 bit sequence, this is shown in table 7.2 on page 72. As no operation needs to swap data between the four different 32x32 blocks, the same steps will run individually and if the architecture supports it, in parallel. In the first step, we look at our 32x32 bit matrix as four separate 16x16 bit matrices, and transform them in the following manner.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

Each number represents one 16x16 bit matrix. 1 is the left upper part, 2 the right upper part, 3 the left bottom part and 4 the right bottom part. As shown in the illustration, the right upper part is swapped with the left bottom part. As our 32x32 matrix consists of 32 four byte values, this swap can be accomplished by swapping the last two bytes in the first 16 integers(0 - 15) with the first two bytes in the last 16 integers(16 - 31). In the second step, we view our 32x32 bit matrix as 16 independent 8x8 bit matrices, and transform them in the following manner.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 1 & 5 & 3 & 7 \\ 2 & 6 & 4 & 8 \\ 9 & 13 & 11 & 15 \\ 10 & 14 & 12 & 16 \end{bmatrix}$$

Similar to the previous illustration, each number represents one 8x8 bit matrix corresponding to each correct placement within the 32x32 bit matrix. This transformation can be accomplished by swapping bits 0x00FF00FF of the first eight integers(0 - 7) with bits 0xFF00FF00 of the next eight integers(8 - 15). The same procedure is done for bits 0x00FF00FF of integers(16 - 23) with bits 0xFF00FF00 of the next eight(24 - 31). What this accomplishes is to swap bits zero to seven, also known as the eight least significant bits, with bits eight to fifteen. In the third step we again divide the size of our representation by two, so we get 64 4x4 bit matrices. We transform them in the following manner.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 9 & 3 & 11 & 5 & 13 & 7 & 15 \\ 2 & 10 & 4 & 12 & 6 & 14 & 8 & 16 \\ 17 & 25 & 19 & 27 & 21 & 29 & 23 & 31 \\ 18 & 26 & 20 & 28 & 22 & 30 & 24 & 32 \\ 33 & 41 & 35 & 43 & 37 & 45 & 39 & 47 \\ 34 & 42 & 36 & 44 & 38 & 46 & 40 & 48 \\ 49 & 57 & 51 & 59 & 53 & 61 & 55 & 63 \\ 50 & 58 & 52 & 60 & 54 & 62 & 56 & 64 \end{bmatrix}$$

Four 4x4 bit matrices represent each of the previous 8x8 bit matrices, and each 4x4 matrix corresponds to its correct placement within the 32x32 bit matrix. We transform them following the same pattern as previous examples. The difference is that we again have reduced the size of our swapping intervals. We swap bits 0x0F0F0F0F of the first four integers(0-3) with bits 0xF0F0F0F0 of the next four integers(4 - 7). The same swaps are done consecutively for all remaining integers(8 - 11, 16 - 19, 24 - 27, are swapped with 12 - 15, 20 - 23, 28 - 31).

As the size of the illustration matrix doubles in vertical and horizontal size we will not draw out the remaining two steps, but the procedure is essentially the same. In the fourth step we divide our representation in two yet again, four 2x2 bit matrices corresponds to one 4x4 bit matrix, giving us 256 2x2 bit matrices representing the complete 32x32 bit matrix. Here we swap bits 0x33333333 of the first two integers(0 - 1) with bits 0xCCCCCCCC of the next two integers(2-3). This is done through all 32 integers in the same pattern as the previous scales. In the Last bit swapping step we swap 1x1 bit matrices, which of course is only one and one bit giving us our final representation of consecutive 1024 bits, representing the 32x32 bit matrix. The swaps still follows the same pattern, and now we swap bits 0x55555555 of every even integer with bits 0xAAAAAAAA of every odd integer. As it is easier to visualize the swaps taking place in binary, a conversion of the hexadecimal values to binary is provided in table 7.1 on page 71.

The last operation performed is the swapping of integers in the 128x32 bit matrix to get the most significant byte of the second third and forth 32x32 bit matrix before the second most significant byte of the first matrix. The second most significant bits of the second third and fourth-bit matrix before the third most significant of the first-bit matrix etcetera. This operation is very similar to a normal transposition. In the illustration below each number represents a 32x8 bit matrix(eight four-byte integers). A figure that illustrates the necessary swaps is seen in figure 7.2 on page 72.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 12 \\ 4 & 8 & 15 & 16 \end{bmatrix}$$

$0x0000FFFF$		0000	0000	0000	0000	1111	1111	1111	1111
$0xFFFF0000$		1111	1111	1111	1111	0000	0000	0000	0000
$0x00FF00FF$		0000	0000	1111	1111	0000	0000	1111	1111
$0xFF00FF00$		1111	1111	0000	0000	1111	1111	0000	0000
$0x0F0F0F0F$		0000	1111	0000	1111	0000	1111	0000	1111
$0xF0F0F0F0$	$\Rightarrow$	1111	0000	1111	0000	1111	0000	1111	0000
$0x33333333$		0011	0011	0011	0011	0011	0011	0011	0011
$0xCCCCCCCC$		1100	1100	1100	1100	1100	1100	1100	1100
$0x55555555$		0101	0101	0101	0101	0101	0101	0101	0101
$0xAAAAAAAA$		1010	1010	1010	1010	1010	1010	1010	1010

Table 7.1: Hexadecimal swap values shown in binary

7.2.4 Detailed explanation of fast binary transpose of 128x8 bit matrix

As previously mentioned, the transposition operation of the 128x8 bit matrix follows the same pattern as the 128x32 bit matrix. The core difference is that it only runs a subset of the operations. We split the descriptor consisting of 128 bytes into 16 eight byte groups, and transpose each individually with the same steps as performed for the 8x8 bit level. The last operation we perform is another bit swapping routine. This routine puts the bit level groups after each other in memory, instead of having the transposed 8x8 bit groups after each other.

7.2.5 Sorting descriptors

After successfully transposing our descriptor into a bit matrix sorted by bit significance, we can further use this representation to our advantage by sorting the descriptors. This sort will give a meaningful representation as each of the descriptors are essentially large 512/128 byte integers after the transformation. As an integer of this size obviously can't fit inside a register, we need to find an appropriate iterative method of comparing two descriptors. An efficient way to do this is to loop through the descriptor 4 bytes at a time looking for the first occurrence of uneven values. If we do not find any, then the descriptors are exact copies.

7.2.6 Data structure

The data structure we store our descriptors in is a lookup table with several layers. A hash table of structures containing hash tables of structures is a good option. To create this structure all 128 bits from a specific bit level will be used as keys. Depending on the descriptor representation used, (vector of floating point numbers or unsigned chars) we will start with the 3rd

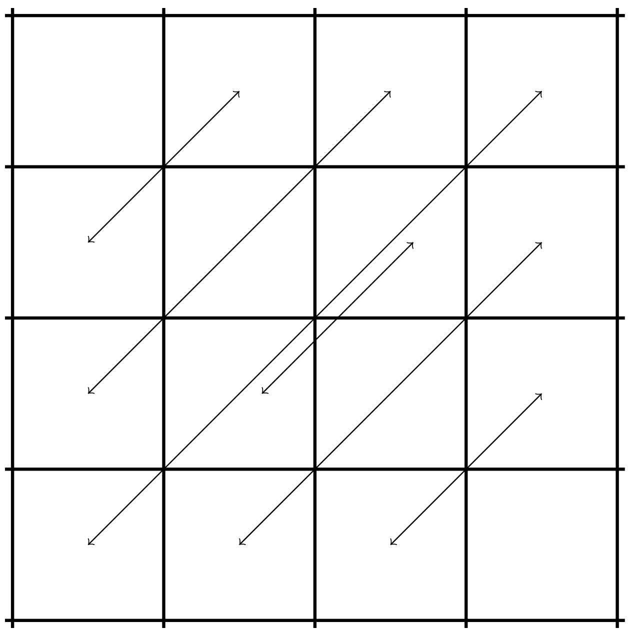

Figure 7.2: Swap table for transforming four 32x32 bit matrix to one 128x32 bit matrix sorted by bit significance. On the GPU, the same pattern is performed on 8x8 bit blocks for completing a 32x32 transpose in parallel.

most significant bits(floating point) or the most significant bits(unsigned char). The reason we can ignore the two first bit levels when dealing with the floating point representation is due to the nature of the SIFT descriptor explained in section 2.2.5 on page 11. As all elements in the SIFT descriptor is in the interval $[0, \frac{1}{2}]$, the sign bit will never be set. Further, as no value is greater than or equal to two, the first exponent bit will be zero as well.

7.2.7 Hash functions

When picking the hash functions for our descriptors, we make a few considerations. Firstly, they should be independent and produce uniformly distributed hashes. Secondly, it is important that the hash function execute fast. Standard hash functions such as SHA1 which might be considered a good option. However, it is not because of the speed consideration. Good choices are often non-cryptographic hashes. A good example is the djb2 hash function [44]. This hash function is a string hash function which produces very well distributed hashes with few instructions on most key sets.

Hash table

The data structure we create will contain similarities to a tree structure, but it can contain multiple root nodes as it is a tree of hash tables. The hash key maps to a certain range in the sorted descriptor list. If we encounter a duplicate key in a map insertion, we increase the range of the provided key. We call these keys and their corresponding range of descriptors pools. The range given by a key can further be reduced by having multiple key levels, meaning that each larger pool can have multiple smaller pools within(tree branches), reducing the amount of possible descriptor matches the further down the structure we get. The number of key levels recorded can either be provided as a dynamic variable provided by the user, be pre-set, or be dynamically set from a size limit of the innermost pools. A illustration of a simple hash table is provided in figure 7.3 on page 73. Each entry in the hash table will have a structure as value. This structure will contain two indexes representing begin and end, as well as a new hash table in which the next bit level will function as the key.

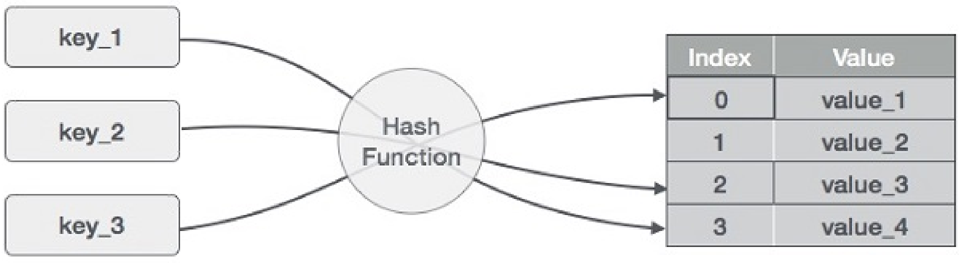

Figure 7.3: Illustration of keys passed to a hash function which are used to compute entry indexes to a hash table

7.2.8 Discover matching range

To match descriptors in this structure the first step needed is to prepare the query descriptors to transpose them with the same method we used on the database set. Further, we want to find the smallest possible range in the sorted descriptor array to minimise comparisons without giving up precision. We find this range in the following manner: For each descriptor, the bit levels are used as keys in the structure, to see if an exact match of that bit level exists. If we find an exact match in the previous level, we will progressively check for exact matches in bit levels until no such exact match exists, or we are above a threshold in level depth or descriptor range size. As two descriptors are needed to compute the distance threshold, the minimum size of a pool is two descriptors. For searches where no match in any bit level occurs, the matching range is set equal to the entire set.

7.2.9 Descriptor comparison

When a perfect match to a new level no longer can be found, we compare the descriptor to every descriptor inside this pool. The comparison operation uses the Euclidean distance, arc length or the dot product for measuring similarity.

7.2.10 Descriptor comparison using Hamming distance

To speed up this process, we can take advantage of the fact that we have already transposed and sorted our data, giving us the opportunity to do this comparison using hamming distance in the next level where they are not equal. The Hamming distance between the descriptors is the number of set bits after an XOR operation. Counting the bits: several options, an easy fail safe way would be to use a gnu built-in function _builtin_popcount. Given sufficient amount of memory, it would also be possible to count up the bits in O(1) using a large lookup table. The drawback with using hamming distance in comparison to the Euclidean distance is that it is not a precise measurement. Further, it will significantly reduce the number of correct matches as well as introduce several false matches.

7.2.11 Distance threshold

The final step used for determining if two descriptors matches, is the distance threshold computation. If the dot product comparison or the Hamming distance comparison method was used to find the two nearest neighbours, we need to recalculate the Euclidean distance or arc length between these two, so we can apply the distance threshold.

7.2.12 Complexity

This algorithm has a worst-case running time of $O(n^2)$ and a best case of $O(n \log n)$ bound by the sorting algorithm. The worst case arise when no sector reduction is found, and we need to examine all descriptors N times which is then the same as the exhaustive search.

7.2.13 Tree structure of transposed descriptors conclusions

This algorithm would allow us to do a rough comparison between two 128 dimensional descriptors with very few operations, allowing us to determine that two descriptors are dissimilar quickly. The cost of the transpose algorithm, the sorting algorithm and the building of the hash table will determine if this algorithm improves performance. The grouping of data must also be thorough. As the grouping of data is linked to the number of descriptors, the effectiveness of this algorithm should improve in correlation to the number of descriptors.

7.2.14 CPU Implementation

We implemented this within the OpenMVG framework. The input data to our matcher is two compressed SIFT descriptor sets represented as two unsigned char vectors. The transposition has two main steps, and is implemented as we explained within section 7.2.2 on page 68. We create a hash table with byte sequences as keys and a structure for each entry containing an interval, as well as another hash table to distinguish further the keys mapped to the same hash entry. This way, we store the two first bit levels as keys. When matching, if we hit a level two pool containing more than two descriptors, we compare them with either hamming distance or the dot product. If the dot product is used, we find the two best matches before computing the Euclidean distance and the distance threshold between them. If we use hamming distance and there is no clear winner, we store all the descriptors with equal hamming distance to a list on which we later compute the Euclidean distance to find the best and next best match.

7.2.15 GPU Implementation

Transposition kernel

The first step to the algorithm is the transposition of the descriptor vectors. As we do not want to corrupt or original descriptors, we allocate two new buffers in GPU memory. We allocate these buffers to the same size as the descriptor vectors. Our first attempt at this transposition was a naive one, as it simply tried to parallelise the bit transpose without any consideration to finding a better algorithm more fitting the massively parallel architecture of the GPU. We call the GPU kernel with one block and 32 or 64 threads per descriptor depending on which of our two suggested implementations we use. We include the algorithm with worse performance here. This is to show that an algorithm that has a low instruction count on the CPU will not be faster than an algorithm with a higher instruction count if this algorithm utilises the GPU resource better as the parallel computation will hide the extra instruction count. The 64 threads version of the algorithm runs in about $\frac{2}{3}$ of the original algorithm.

We create one shared array of 128 floats, and the threads write the descriptor from global memory to the new shared array. Each thread writes four floating point numbers. After the write we synchronise our threads with a barrier, ensuring all threads have completed the previous step before we use the data. Four threads call the main transposition routine, each responsible for a 32x32 section. Again the threads are synchronised with a barrier. The final step is to combine the separate 32x32 transpositions to get the 128x32 transposed descriptor. As all threads write to separate floating point values, there is no need for synchronisation. This step is done from the shared memory buffer and back to the global memory before the kernel completes. The main Problems with this implementation is that we need two synchronisations, as well as having most of our threads idle during the transposition. We solve these problems by using an alternative algorithm which uses 64 threads through the entire process and runs without the need for synchronisation.

First each thread reads in an 8x8 block bytes to local memory. This block is not consecutive in memory, as we want to eliminate the need to read or write data from other 32x32 blocks than what we initially assigned the threads. Thus, the blocks consist of 8 bytes each having a 16-byte interval between them. All bytes in the 8x8 matrix will be in a separate bit level when the transposition is complete. This way 16 threads will be cooperating on each 32x32 transposition. When the 8x8 bit transposition routine completes, we finish by transposing the 8x8 blocks between each other. This transposition writes from the local memory to the global destination memory. Throughout this entire process, no synchronisation is needed as two threads never write or read from the same memory address.

Sorting transposed descriptors with thrust

As there is only need to sort one of the descriptor sets, we pick the largest one. Although this will make the sort itself slightly more time consuming, it will later benefit the algorithm runtime to build the matching structure with the larger descriptor set. We create a thrust device vector of integer type with length equal to the descriptor count of the set. It is initialised with thrust sequence to index the descriptors array position. This sequence vector is created to perform an indirect lookup sort on the descriptors. The indirect lookup comparison compares one byte at the time to avoid the possible NaN value of a float or integer comparison. Comparison of NaN values is an illegal operation. IEEE floating-point standard [33] single precision (32-bit) NaN would be:

$$s111 \quad 1111 \quad 1xxx \quad xxxx \quad xxxx \quad xxxx \quad xxxx \quad xxxx$$

Where s is the sign. If the x sequence is all zeros then the floating point number would be infinity, otherwise it would be a NaN value. The comparison function compares bytes iteratively as long as they are equal. As thrust sort does not accept equal values, if all 512 bytes of the descriptors are equal, we compare the indexes of the indexing array to determine sort order.

Building the GPU Hash Table

For efficient matching, a matching structure is created based on the transposed descriptors. A hash table is a suitable choice. We pass the descriptor count together with the number of hash entries to an initialisation function, which allocates the required memory. We chose the number of hash entries carefully. Since our hash table will have no further elements added after its first build, we do not need to worry about the load factor, i.e. the percent full point where the table will increase its size and redistribute elements. For maximum efficiency, the hash entry count should be set equal to the descriptor count. However, as there will likely be several duplicates, $\frac{descriptor\ count}{2}$ will be more than sufficient in most cases.

To add keys to our hash table in parallel, we cannot hash two keys to the same value without synchronisation. If this happens, several things could go wrong. One of the key-value pairs could disappear, or we could get a key-value pair in which the key does not correspond to the value as they are from two different descriptors. To solve this, we create one mutual exclusion lock for each hash entry. To insert a new key-value pair to the table, the lock corresponding to the hash value is first locked before we insert the new entry. If one or more entries already exists at this location, we compare our key to the keys of all the other entries, as we do not want to insert it if it has a duplicate. If a duplicate exists, we update its value. We store multiple entries as a linked list, each entry having a pointer to the next. An interesting problem arises if more than one thread within a warp (an assembly of 32 threads executing together in lockstep) tries to acquire the same lock at the same time. Sanders and kandrot [59] suggests to create a loop iterating over the 32 threads in a warp, and an if statement inside the loop, letting only the thread_Idx % loop_iteration continue to the locking portion of the code.

When comparing this GPU implementation to the CPU implementation, we see that their performance is quite similar. The GPU performs poorly at parallelising access to data structures which need synchronisation. Thus, building a hash table would not be a profitable operation to perform on the GPU. However, as it significantly reduces the cost of data retrieval, it is still a good choice. Additionally, our data already resides on the GPU, and thus, it is counterproductive to perform this operation on the CPU as the memory copy from device to host is costly.

Get section from hash table

To retrieve a section to search for each of our descriptors we first create two integer arrays of the same length as the descriptor count to hold the begin and end position for each descriptor. Each thread is given an index based on the specific block and thread index, and we retrieve the specific section by hashing the descriptor keys and comparing the keys with the different keys in the linked list in the acquired pool. If we find a match, we set the begin and end index for the corresponding position in our two integer arrays. If the key does not match any hash entry, we set the section equal to the entire descriptor set.

Sort descriptors based on indirect lookup table

In our initial implementation we did not perform this operation and used the previously computed indirect lookup table to locate the descriptors in the interval. However, as we experienced a slow execution time, we speculated that this was due to our non-linear memory reads. Therefore we created a kernel which sorts the descriptors based on the indirect lookup table. The kernel copies descriptors from the original descriptor memory to the previously used transposed descriptor memory, thus synchronisation is not needed. Strided memory access can hurt performance, so the kernel reads from and writes to global memory with 128 threads per block. Each thread reads a single floating point value from the blocks designated descriptor using its thread ID to locate the correct position. Further, it writes the floating point value to the corresponding position in the new descriptor location found through the indirect lookup table. This way both memory reads and writes are coalesced. Sorting the descriptors reduced the algorithms running time by 20%.

Compute hamming distance in section

This function measures the hamming distance between the descriptors in a section to find the best match. Bits are compared in iteratively lower levels until we find the two best matches. When implementing this concept, we realised that hamming distance between floating point vectors is imprecise to a much higher degree than with the unsigned char representation, and as a consequence, the number of correct matches reduces to a point where the matcher is no longer useful. Therefore, we did not optimise the kernel further. The concept still showed great promise concerning running time, completing in $\frac{1}{2}$ of the dot product match time, with a kernel running one thread per block. Running one thread per block is inefficient as 31 out of 32 threads in a warp will be idle, so the kernel had considerable room for improvement.

Compute dot product in section

The main difference between this kernel and the previously explained dot product kernel is that this kernel only computes the dot product against the descriptors within the given section retrieved in the get section kernel. We initially used the global begin and end index arrays in the loop, however, after writing these to local variables before the loop and using these local variables in the loop reduced the algorithms running time by 5%.

GPU implementation Conclusions

When we compared this ANN search to the exhaustive NN search, we saw that the exhaustive dot product search is faster. We found this strange as we had hypothesised that our ANN algorithm would perform similarly to the hamming distance comparison, and closer to the other approximate searches than to the exhaustive search. After profiling the algorithm with the NVIDIA Visual Profiler, we saw that the main bottleneck was not the preprocessing steps as we had initially thought; it was responsible for no more than 5% of the running time on our largest image set. Computing the dot product of a section spent the remaining 95% of execution time.

This realisation led us to several experiments. The first experiment was to preprocess the original descriptor memory instead of allocating new buffers. After acquiring a search interval, the descriptors would be transposed back to their original state before the dot product was computed. The performance gained by this was minimal at best. Further, we tried sorting the original memory instead of using the indirect lookup array as we believed that the non-linear memory access could be at fault for the slow performance. This assumption was correct, reducing the algorithms running time by 20%. Further, we removed all repeated global reads by creating local copies, which reduced the running time by an additional 5%. However, when compared to the exhaustive searches on our largest image set it only performed slightly better than the Euclidean distance and worse than the dot product which completes in $\frac{2}{3}$ of the time. Therefore, it is still faster to run the dot product against all descriptors in comparison to the dot product of a reduced range. We are not sure what the cause of this is, as the profiling results show very similar values if we ignore the running time.

7.3 Ideas that did not work

Not all of our ideas had the desired effect we were looking for. Some of them did not give us a speedup, and some did not give us the correct matches.

7.3.1 Smart exhaustive search

We created an Approximate Nearest Neighbour search based on the exhaustive search algorithm. This algorithm would retain information about matched descriptors, removing them from the database set such that the number of descriptors to compare against would be reduced for every match found. We implemented this using a distance threshold between best and second best match. If the ratio was sufficiently large, we would put the last element in the database set in the position of the match we found. This would ensure its removal and at the same time shrink the database set. The main problem with this algorithm was that the more descriptors we removed, the chance for the other descriptors finding their correct second best match was reduced. This resulted in many false positive matches being accepted. The speed increase was also minimal, as most descriptor sets on average are several times larger than the number of matches. Therefore we only saw a small runtime reduction of around five to ten percent. After running our tests, it became clear that this algorithm sacrificed too much precision for the small speedup we achieved, thus we concluded this ANN algorithm to be inadequate.

7.3.2 Binary tree from transposed descriptor

Our initial reason to use transposed descriptors came from the idea of building a binary search tree for the Nearest Neighbour search. We thought this would prove an interesting Nearest Neighbour algorithm as a binary tree would nicely reflect the structural relationships in the data and provide efficient searching considering that tree search is an operation of logarithmic running time. This tree would have a depth of 128 as the

SIFT descriptor has 128 elements. We would initially build this tree with all the descriptors from one descriptor set. When matching descriptors in the tree, we would search the tree one node at a time, going to the left or right branch depending on the bit at the equivalent position in the transposed query descriptor. This concept is illustrated in figure 7.4 on page 80.

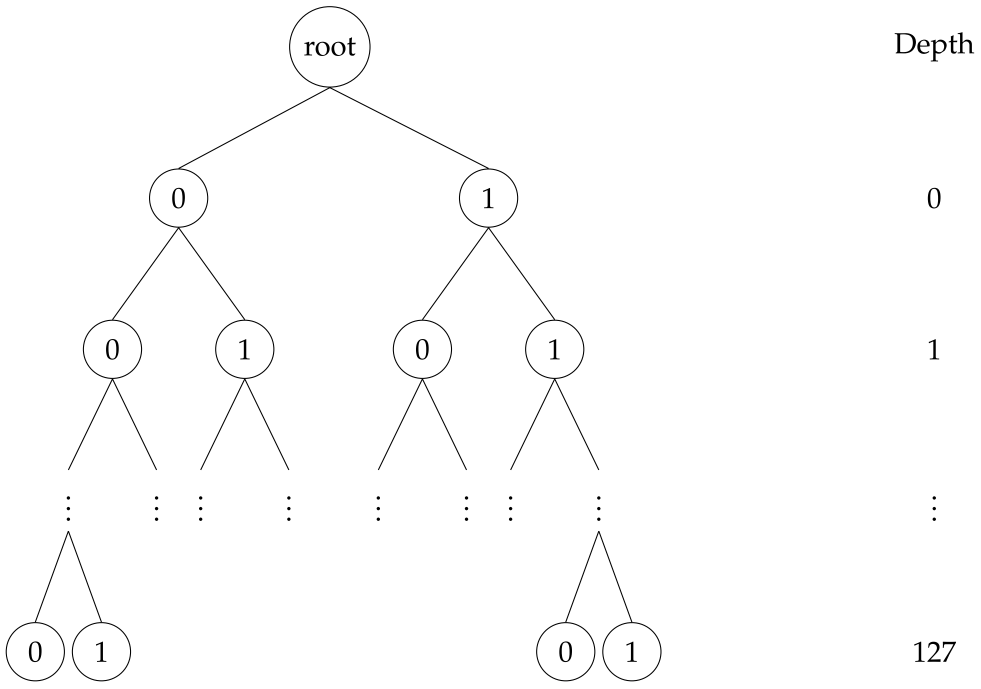

Figure 7.4: binary tree

Implementation

After searching the tree and reaching a leaf node, this node could either contain a pointer to a new binary tree consisting of the next bit level, or a begin and end index to a sorted descriptor array. For traversing the new binary tree, the descriptor count should be over a previously set threshold that balances the extra work of searching a new tree to the number of descriptors possibly removed from the set. As we need two descriptors for the distance threshold, a pool containing only a single entry would add the descriptors in the pools on its right and left side to its set.

Problem

The main problem with this algorithm is that the search of the tree made the elements near the start of the descriptor matter more than the elements further back. As all elements of the SIFT descriptors have equal weight, this would lead to many false negatives as an early branch in the wrong direction could disqualify a close match. As a result of this, we had to scrap this algorithm as it would not give robust results concerning the equal weight of each element in the bit sequence. However, as this method would work for complete matches within a bit level, we continued to develop a modified version of this idea.

7.3.3 Hamming distance

The main problem with using hamming distance as comparison mechanism on the transposed descriptors is that the best match in one bit-level not necessarily leads to the nearest neighbour. To illustrate this, we give a small example. Let us say we have a database with the following bit sequences.

Binary	Decimal
1111	15
1110	14
1011	11

If we then would like to match a new binary sequence 1100 (12 decimal) against this database using bit-level hamming distance the problem with this technique becomes apparent. When looking at the decimal numbers, it is obvious that 1011 (11 decimal) is the nearest neighbour of 1100 (12 decimal). However, using our proposed hamming distance for comparison, this bit sequence would be excluded as it is not the best match after the second comparison, where 1111 (15 decimal) and 1110 (14 decimal) both are a match, while 1011 (11 decimal) is not. As a result of this, the Nearest Neighbour implementation using bit-level hamming distance as comparison operator on transposed descriptors cannot guarantee correctness, and thus it will be an approximate Nearest Neighbour search. This explains the poor match results discussed in section 7.2.15 on page 78. Further, it raises the question around the precision loss when matching a complete bit-level as we do in our proposed ANN matcher. Experiments show that the precision loss is minimal as several levels after each other are complete matches.

7.3.4 Bloom filter

A Bloom filter is a fast and memory efficient data structure which can check if a specific element is part of a set. While The Bloom filter can determine if an element is not present in a set, it can not guarantee that an element is part of the set due to its overlapping nature. As such, it is a probabilistic data structure, because it can return a false positive answer. The standard Bloom filter implementation is a large array of size N/8 initialised to zero. Thus, considering each bit as a separate bin, the filter has N entries. A Bloom filter also has M keys. The size of M will determine the probability for the Bloom filter to return a false positive answer. Therefore the number of hash functions M is a trade-off between speed and accuracy. The Bloom filter accuracy can also be increased or decreased by changing its size, as a larger filter will have less overlaps. The procedure for adding an element to the Bloom filter is quite simple. For each hash function we have in the Bloom filter, we set the bit at the given hash position. Whether we have previously set the specified bit is not of concern. Thus, it is easy to see that an element not inserted in the Bloom filter still could have all its bits set from other elements. A illustration of a Bloom filter is provided in figure 7.5 on page 82.

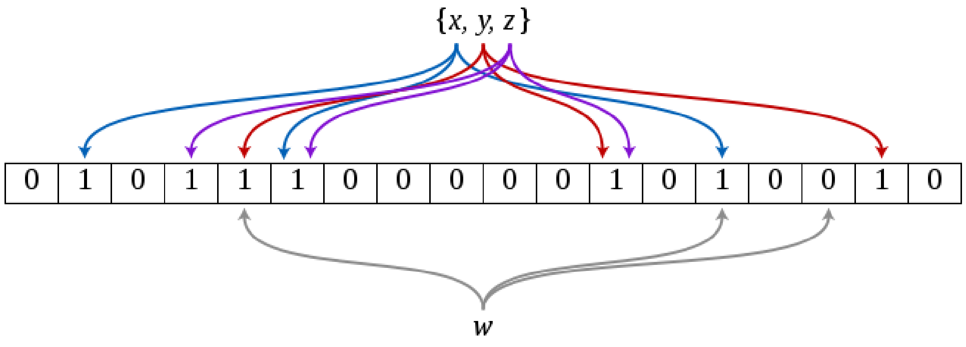

Figure 7.5: Element x, y and z inserted in a Bloom filter. The query by element w will yield a negative return as not all bits corresponding to w is set in the Bloom filter. Three hash functions is used.

Idea

Our idea was to improve efficiency by creating a Bloom filter which would function as a fast check to see if the element we are looking for does not have a potential match. If the quick check yields a positive response, we query the lookup table. As with the lookup table, we use all 128 bits from a specific bit level as keys. Depending in the descriptor representation used, (vector of floating point numbers or unsigned chars) we will start with the 3rd most significant bits(floating point) or the most significant bits(unsigned char). The Bloom filter would be very fast to set up, especially on the GPU by using bytes instead of bits for parallel inserts. If two threads set two different bits inside a byte at the same time, it would lead to a potential race condition. However, this would not be a problem if bytes were used instead of bits as all the slots we write in would always contain one. Writing one to a byte several times might be unnecessary, but as the alternative would be to perform a synchronisation mechanism between the threads, this would not matter.

Bloom filter conclusions

The reason we originally thought this would work was due to the poor performance of our hash table. The main problem was that several keys hashed to similar values in the hash table. Because of this, we store the values as a linked list and check against the lookup table to compare the incoming descriptor against all keys in the list until we are at the end of the list or we find a match. However, several factors made the Bloom filter obsolete. Firstly, overlapping keys became much less of a problem after improving the hash function used. Secondly, the non-existent keys would often not hash to the same value as other keys, and it would hash to an empty bin, quickly determining that no suitable match exist. Therefore, the hash table is both faster and more precise, as the hash table only needs one hash function, while the Bloom filter would use at least two hash functions to limit false positive results.

7.4 Results

This study documented examples of two variations of an Approximate Nearest Neighbour compensation strategy used to reduce the runtime of the Nearest Neighbour descriptor matching process. This section describes and discusses overall running time results, as well as results compared to matching correctness. We will compare our results against each other as well as the two state of the art ANN algorithms, k-d tree [53] and cascade hashing [16]. For the benefit of being able to compare the Approximate Nearest Neighbour solutions to the Nearest Neighbour solutions as well, we will use the same image test sets as can be seen in section 6.6 on page 54 and the benchmark will be the standard exhaustive search algorithm using the Euclidean distance metric.

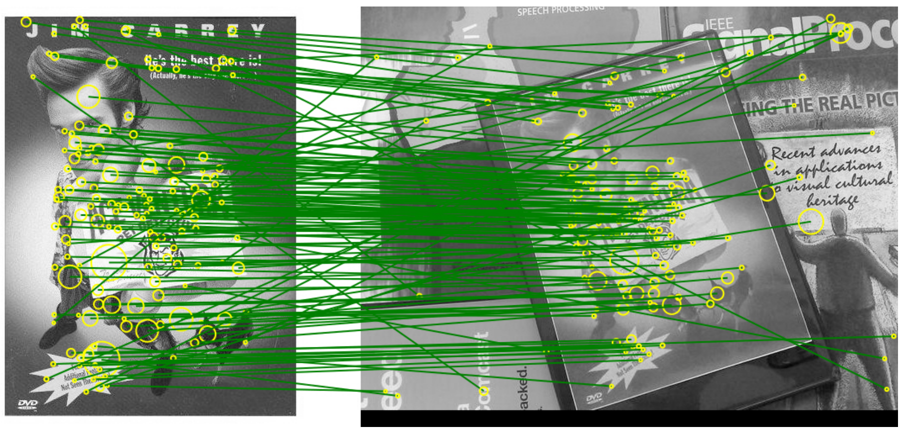

Figure 7.6: ace trans hamming

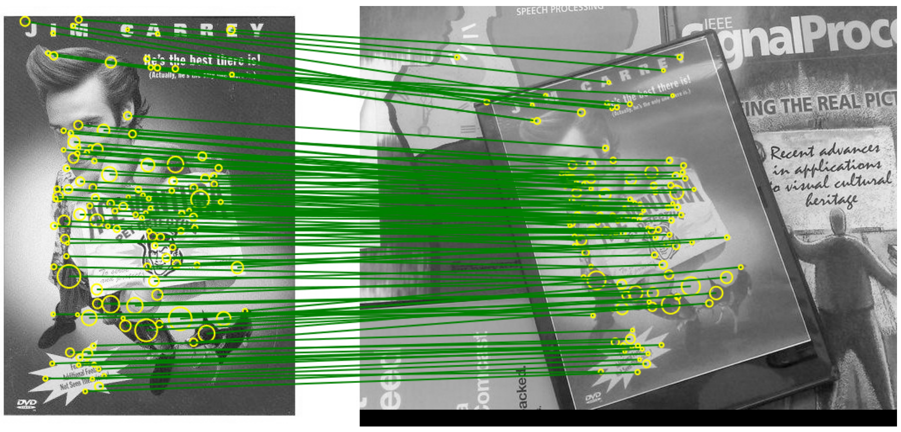

Figure 7.7: ace trans dot

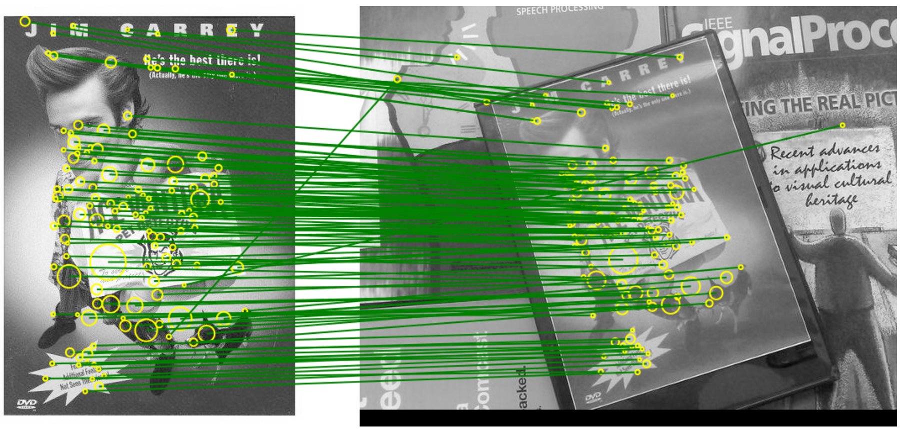

Figure 7.8: ace k-d tree

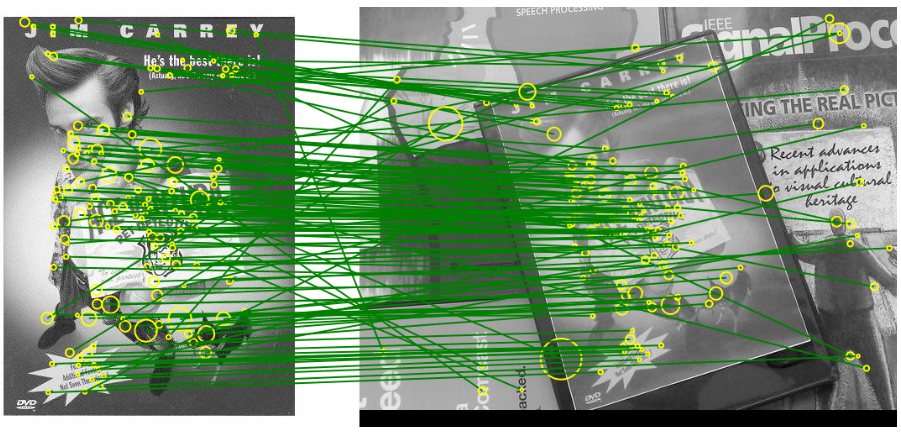

Figure 7.9: ace cascade hashing

Algorithm	Matches	Runtime	Speedup
Exhaustive search	122	32 ms	1x
Transpose hamming	146	26 ms	1.2x
Transpose dot product	121	10 ms	3x
ANN k-d tree	123	60 ms	0.5x
Cascade hashing	126	10 ms	3x

Table 7.2: CPU test on Ace

The tests on the low resolution image set Ace shows us that there is small differences between the methods. All of the solutions has a relative similar number of matches, but both our hamming distance proposal and the Cascade hashing solution has to many incorrect matches. This is probably because they both rely on hamming distance, which does not take into consideration the bit significance which can be a big problem on the relatively large dimension of 128 the features in our tests have. The runtime of our Transpose dot product solution is encouraging, and with its precise matches, it seems like the best option for low resolution images at this stage.

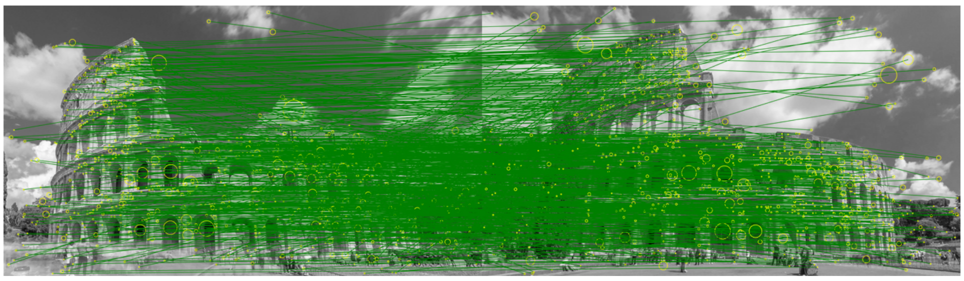
Figure 7.10: col trans hamming

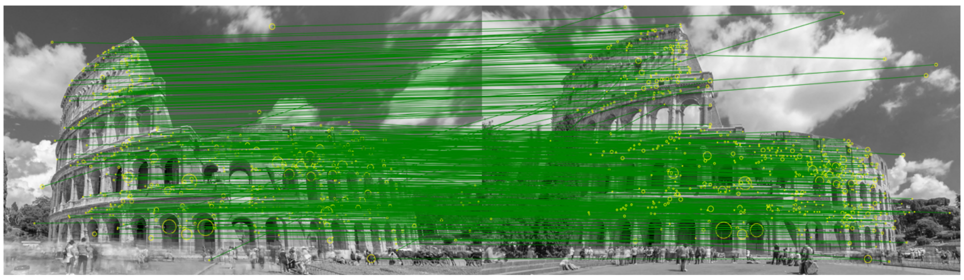
Figure 7.11: col trans dot

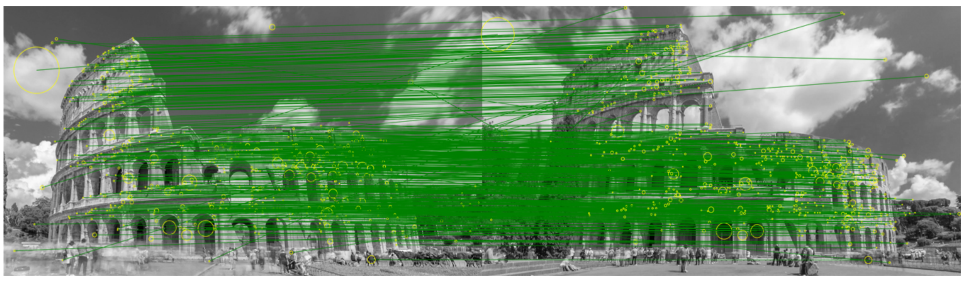

Figure 7.12: col k-d tree

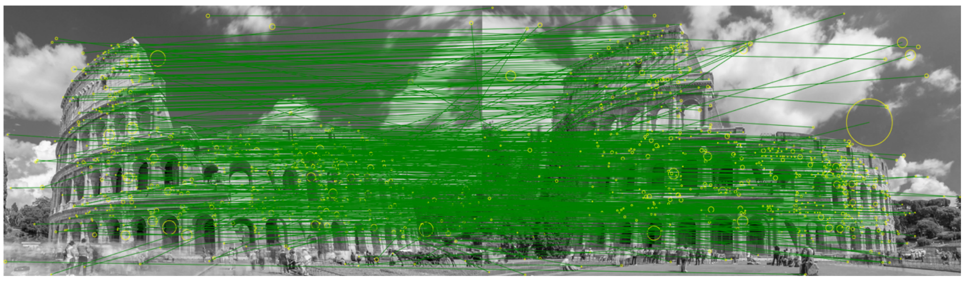

Figure 7.13: col cascade hashing

Algorithm	Matches	Runtime	Speedup
Exhaustive search	540	3363 ms	1x
Transpose hamming	891	1310 ms	2.5x
Transpose dot product	541	970 ms	3.5x
ANN k-d tree	559	234 ms	14x
Cascade hashing	590	83 ms	40x

Table 7.3: CPU test on Col

On the HD image set of Colosseum we see that our solutions are faster than the exhaustive search, but they are falling behind the other approximate searches. The matches from the transpose hamming distance solution is not as good as they should, while our transpose dot product still produces matches close to the ones the exhaustive search provides. The k-d tree solution is so much faster than our solutions, and with similar matches as the exhaustive search it is clearly a better option on this data set. Again the Cascade hashing is the absolute fastest solution, however, the matches are not as good as the k-d tree solution.

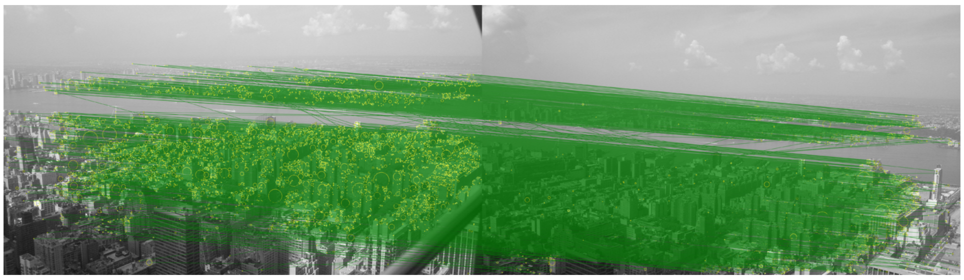
Figure 7.14: trans ham NY

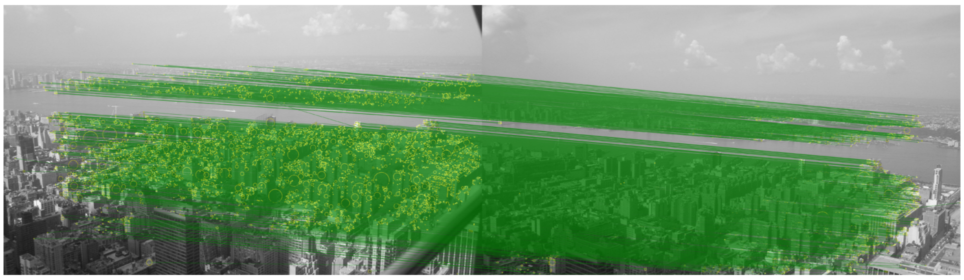
Figure 7.15: trans dot NY

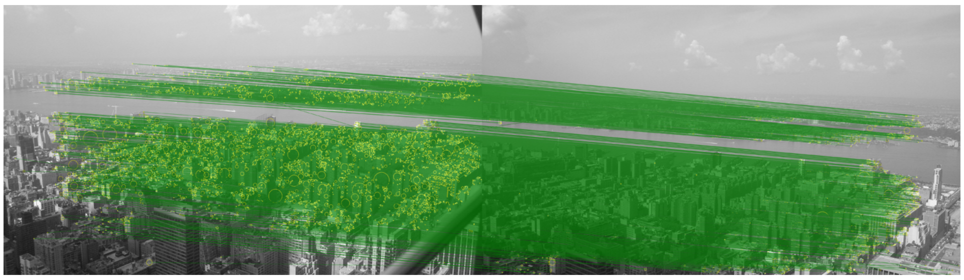
Figure 7.16: k-d tree NY

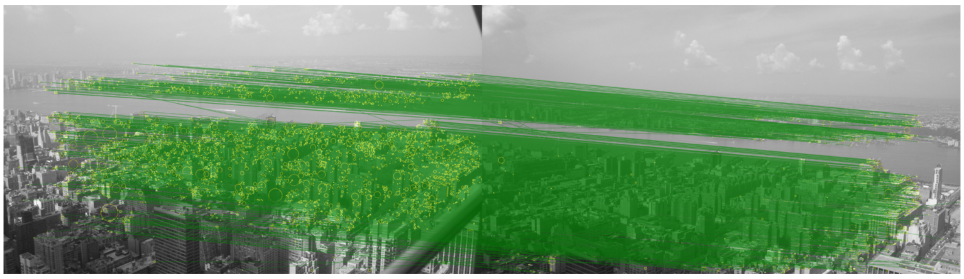
Figure 7.17: cascade hashing NY

Algorithm	Matches	Runtime	Speedup
Exhaustive search	4166	14980 ms	1x
Transpose hamming	4357	6442 ms	2x
Transpose dot product	4153	4493 ms	3x
ANN k-d tree	4159	481 ms	31x
Cascade hashing	3772	183ms	82x

Table 7.4: CPU test on NY

On the New York image set, we see that the difference between the solutions is even greater. Both our solutions have to an extent the same speedup as on the Colosseum tests, while both the k-d tree solution and the cascade hashing are running away from our solutions regarding speedup. There is not so easy to compare the matches on this data set, but both the k-d tree solution and our dot product has a similar number of matches compared to the exhaustive search. The cascade hashing and our hamming distance solution has a more dissimilar number of matches, which would suggest that the matches are not as good as the other solutions.

Algorithm	Matches	Runtime	Speedup
Exhaustive search	752	3.627 ms	1x
Transpose hamming	220	20.517 ms	0.2x
Transpose dot product	720	21.055 ms	0.2x
ANN k-d tree*	755	118 ms	0.03x

Table 7.5: GPU test on Ace
Done with NVIDIA Visual Profiler
*ANN k-d tree done on CPU

Algorithm	Matches	Runtime	Speedup
Exhaustive search	5550	519 ms	1x
Transpose hamming	5594	305 ms	1.7x
Transpose dot product	5815	739 ms	0.7x
ANN k-d tree*	5970	1997 ms	0.3x

Table 7.6: GPU test on Col
Done with NVIDIA Visual Profiler
*ANN k-d tree done on CPU

Algorithm	Matches	Runtime	Speedup
Exhaustive search	24999	3173 ms	1x
Transpose hamming	4915	1082 ms	2.9x
Transpose dot product	19432	3255 ms	1x
ANN k-d tree*	25668	4864 ms	0.7x

Table 7.7: GPU test on NY
Done with NVIDIA Visual Profiler
*ANN k-d tree done on CPU

The GPU tests show that our ANN solutions did not bring anything to the table. On the low-resolution images, our algorithm is ten times slower than the exhaustive search and not a viable option. However, as the number of features increases on the HD images, the solution is improving compared to the exhaustive search giving it almost the same runtime on the largest image set of New York. This improvement shows that our solution is best suited for large inputs. However, our ANN is still not good enough, as the exhaustive search is faster. The hamming distance does also provide way too few matches on two of the tests, which suggest that this method does not work as intended. One thing to note is that the ANN k-d tree and our ANN solutions scales well, and does considerably better on the larger image sets. Even though these algorithms are slower for all our tests, it is likely that they could provide a speedup on larger descriptor sets. When comparing our ANN dot product to the k-d tree, we see that our ANN is faster. However, the run-time gap is less for our largest image set (NY) than for our HD image set (Col) which suggests that the k-d tree algorithm scales better with increased descriptor set size.

7.5 Conclusion

Our approximate searches did not yield the results we wanted. They do find an acceptable number of correct matches at least for the transpose dot product version, and they do provide some speedup on the CPU over the Euclidean distance exhaustive search. However, they do not compete well enough with the already existing solutions available. The GPU tests show that the ANN algorithms perform decently, but the exhaustive search is a better alternative since it is faster and more precise, due to the predictable nature of the exhaustive search which is suitable for parallelisation.

Since the exhaustive search is the best solution on these tests, we can also conclude that our dot product solution still is the best option as it offers a significant speedup on the original GPU L2 implementation.If the GPU option is unavailable, there would be a discussion between the ANN k-d tree and dot product solution, and whether the diminishing matches the approximate search have will weigh more than the speedup accomplished.

The main problem with the approximate searches is the quality of the matches. Most of the solutions can be tweaked to achieve a higher percentage of correct matches, but this will be at the cost of the overall runtime. The already existing ANN k-d tree solution in OpenMVG is an excellent example of this, as it is very effective when providing up to 80-90% correct matches. When the correctness is above this percentage, it will quickly deteriorate, and as it reaches 100%, it will either be as slow as the exhaustive search or even slower [53].

Chapter 8

Conclusions

8.1 Problems and solutions

Many things can go wrong when manipulating parts of such a large program OpenMVG is. If one small domino brick of the program falls, all other parts will usually follow. On the other hand, there is almost worse if the program runs "fine" even though the part we change has some faults, as this is harder to detect. So one of the most important things to consider is to verify the results often to avoid complications later on. Since this is a large and complex program, it is also difficult to find the flow of it, and know where to look if something is wrong as there are many dependencies each part of the program has.

8.1.1 Getting to know the code

One of our first issues was to understand how some of the variables and data were structured, and how to access them. There was a lot of trying and failing, and we would change small lines in the code incrementally to learn how things worked. This process made us more familiar with the code itself, and it made things a lot easier over time.

8.1.2 Finding errors

Another issue is to detect errors and to pinpoint where they are. The compiler will in many cases do this for us, and report problems in the code, but there are also errors that will slip past it. It is often faults that are not necessarily errors, but issues that provide wrong variables and calculations resulting in negative results. The problem with this is that even though there is only one small error, it will often give the impression that there are many of them as all the parts depend on each other.
In cases like these, we would start at the earliest point possible in the code, and verify each result from there on. After a while we learned that it was easier to do this before any errors occurred, so we did a lot of unnecessary checks, but in the long term, it helped us a lot. We also kept external methods to a minimum, writing most of the solutions to get a more in-depth understanding of the code.

8.1.3 GPU issues

Programming on a GPU presents a whole other range of difficulties compared to CPU programming. Memory has to be allocated between the CPU and the GPU, and there are challenges regarding shared memory. Concurrent programming is all about doing as much as possible at the same instance, dividing larger problems into multiple smaller and let an independent thread handle each of them.
One of the issues regarding parallel computing is that there is no fixed ranking between the different threads, all of them will execute their code as soon as they are allowed, and there is no way of telling which order they are executing. Random ordering makes it harder to find errors, as the code may or may not execute correctly depending on the order of the threads. One of the most common issues we encountered were race conditions, and this occurred when multiple threads read or wrote to the same memory which led to one thread reading a variable too early or too soon depending on the other threads behaviour. There are many different solutions to avoid issues like this, from atomic operations to locks, allowing only one thread at a time accessing a specific memory location. However, these types of solutions will often slow down the program as there is no longer concurrency, as threads have to wait to access different sections of the memory. It is far better to avoid threads using the same memory locations, to make them as independent as possible. Letting each thread do all its calculations before merging them at the end is the best practice.

Thrust sort equality issues

As we were trying to implement the thrust sort function, we stumbled on a problem that at first seemed very complicated, but as it later turned out were quite simple. Our purpose was to sort our transposed descriptors with thrust, but the sort function had some strange effects on our data. Each descriptor has an index, and we sorted the indexes based on the value of the transposed descriptors. After the sort, some of the indexes were missing, and duplicates replaced them. This behaviour was not what we expected, and there were not any similar cases that other had experienced as we explored this issue on thrust forums.
In the end, we learned that the thrust sort function could not handle equality. Thus, it did not know what to do when some of the data it was sorting had the same value, and the behaviour in these cases was unspecified. The solution to this was to change the sort parameters and create a new parameter for equality thus giving the sort function a standard behaviour each time the parameters were equal.

8.1.4 Measuring the results

The last issue is regarding the measurement of our work. A program today could work fine on the computer the programmer uses, but that may not be the case when using it on different hardware. We did do a timed test on our machine, but there is no way of guaranteeing the results we got. Our approach was to merge both physical tests and theoretical calculations. We do not claim that the test results are 100% accurate, but they do tell us something about the performance of our implementations.

8.2 Work approach

Since the beginning of this project to the end, it became clear that there was room for much improvement. At the same time, it revealed to be a tough process, and there were several mountains to climb before reaching the end. The learning curve was steep, and there was lots of new theory to look into before we were able to change anything. Our approach to it was to do everything in small iterations and try to verify every result on the way. By this method, we were able to gradually improve the program step-by-step continuously exploring new ideas and changes on the way. We focused on one problem at a time and did not explore other parts until we either were satisfied with our code or if the conclusion was that the idea was a dead end. One thing that helped a lot was to make most of the code separately from the program, and verifying it before implementing it.

8.3 Our implementations

In the end, we provided three different solutions to improve the Nearest Neighbour search. Two of them changed the metric to do fewer operations so that the exhaustive search would perform better. The last solution was more complex and based on changing the memory layout for the data set and performing a different search algorithm to find the potential matches.

8.3.1 Arc length

The arc length would explore the possibility of enhancing the matching process. Our initial thought was that this could have the same runtime as the Euclidean distance, but after looking at the calculation needed, we feared that this would have a negative impact on the runtime. However, after we took into consideration that we were calculating the length of normalised vectors, we managed to speed up the algorithm, by replacing these calculations with a constant. The use of a constant led to improving the complexity from $O(6n)$ to $O(2n)$ making it potentially more efficient than the Euclidean distance.

So what have we learned? The arc length calculation allows us to provide the distance threshold as an arc length on the sphere which gives the same precision as the Euclidean distance algorithm with faster execution time. Additionally, It is an interesting method to describe how close to each other the descriptors are allowed to be as the distance threshold now can be provided as an angle. It gives equal performance to the Euclidean distance algorithm for lower inputs and has some speedup for larger inputs. This method needs a review of the distance threshold to secure a larger amount of matches.

Since the solution is faster, it is a viable alternative to the Euclidean distance. There is also a potential for this method to be even faster by replacing the inverse cosine calculation with a lookup table.

8.3.2 Dot product

In the dot product solution, we wanted to improve the runtime without diminishing the precision of the matches. We knew that this implementation would have fewer calculations than the Euclidean distance, but we were not sure how the quality of the matches would be. After doing some tests, we saw that both the number of matches and their quality were the same, and the runtime already on the small test set were twice as fast as the original implementation. When escalating the size of data, the overall runtime revealed to be progressively better, showing that the solution scales well on large data sets.

Having found a solution that worked, we decided to implement it on the GPU in the PopSIFT framework as well. Since the GPU implementation of the Euclidean distance algorithm already was a significant improvement on the original implementation, there was a question if our solution would manage to improve the runtime at all. However, after running a few tests, we saw that our solution did have a positive impact on the runtime. The difference was not as considerable as on the CPU implementation where we got a speedup of 4.5x, but it was enough to see the benefits of our solution. The improvements we got with a speedup of around 1.6x reflected the expected enhancements the mathematical formula suggested. The reason we got fewer improvements on the GPU in comparison to the CPU was that the CPU Euclidean distance implementation in OpenMVG did not take advantage of the unsigned char representation and used additional floating point variables.

Our dot product calculation solution improves the program both on the CPU and the GPU, and since the speedup is present for every input size, the solution could easily replace the original implementation without any drawbacks on both platforms. Since the final calculation used for the distance threshold is such a small part of the solution, we can choose between the Euclidean distance and the arc length at this point without losing performance. That means that using the dot product could work for all programs matching normalised vectors, especially if there is a need for exact results which approximate searches not necessarily can provide.

8.3.3 Tree structure of transposed descriptors

The last algorithm we explored was the tree structure of transposed descriptors. After testing on several different image sets, we concluded that this algorithm produces a moderate improvement in execution time compared to the Euclidean distance exhaustive search when there are enough descriptors to group them. On data sets with less than 1000 descriptors, it is likely that the pool size will be substantial, thus reducing the potential efficiency of the algorithm. Still, the execution time is not slow, as some grouping of the data is likely to occur. On datasets with size above 10000 descriptors, the algorithm starts to improve its efficiency as more grouping occurs, reducing the search area to a small subset of the database for most descriptors. The algorithm continues to grow in efficiency with larger descriptor numbers due to descriptor equality in increasingly lower key levels.

When compared to the k-d tree and cascade hashing algorithms, it is clear that this is an inferior algorithm regarding execution time. The main problem is that the SIFT descriptors have a similar bit structure to each other, as they are in the interval $[0, \frac{1}{2}]$. Therefore, specific groups are likely to contain large chunks of the descriptors, unless the descriptor set is large enough to split further. As we compute the best and second best match within a group through an exhaustive search, this slows the algorithm down considerably.

When it comes to quality of matches, this algorithm is an Approximate Nearest Neighbour because an incomplete bit level key potentially can be better than the complete bit level match. However, as we only sort on full bit level matches, our matching results are more precise than the ANN solutions we used as a comparison. In conclusion, we are of the opinion that this algorithm is not fast enough to justify the loss of precision.

8.4 Future work

There is still more to explore regarding the exhaustive search algorithm. We feel that there are more potential improvements regarding the calculations. For instance, the exhaustive search behaves the same each iteration and is still the main bottleneck in the matching part. There are different approaches to improve this further, but we did not find the time to explore these. On the CPU side, there is the possibility to parallelise using multiple cores. Another option would be to write parts of the code in assembly to remove all unnecessary operations and get an even greater speed on the calculations. The main problem with these solutions is that they would make the algorithm architecture dependent and not generic.

8.4.1 Implementing dot product in other methods

Since we proved that the dot product was capable of improving the exhaustive search without losing precision, there is also a high possibility of doing so on other methods used. The first thing to look at would be the solutions that do many calculations with Euclidean distance and change it to dot product to enhance the performance. Another solution will be to see if there is any solution that could be changed to do more calculations and then use dot product on these, as a way to improve the precision of the method. In either case, it would be exciting to see the benefits it could yield.

8.4.2 Transposed descriptors ANN

We still think that our approximate search based on transposing the descriptors and building a tree structure could be beneficial. This solution was much more complex than our other methods, and many parts probably could be solved differently.

8.5 Conclusion

We managed to implement a new metric in dot product calculation instead of using Euclidean distance, thus making exhaustive search more attractive to use in place of the Approximate Nearest Neighbour searches. We also managed to implement arc length calculation as an alternative, although not as efficient as our dot product solution proved to be, it has the potential to be even more effective with the implementation of an inverse cosine table. Our approximate searches did work to some extent; however, they did not generate the desired results we were after.

With our new metric, the exhaustive search performs 4.5 times faster on standard HD images and is performing almost as fast as the ANN k-d tree solution on the CPU. On the GPU, the dot product is the best option available, and it is also the best solution compared to the approximate searches on the CPU. It is as precise as the Euclidean distance, so there are no obvious reasons not to use this method instead.

Depending on the scenario there will still be a discussion whether to use exhaustive searches or approximate searches, and there is still a big gap between them when the size of the images grows larger. There are some reasons why the exhaustive search is the ideal solution. The main reason is that it will always find the correct matches regardless of the number of features. The runtime is easier to predict as it will always explore all possible matches, in the same manner, each iteration. Its rigid design does also make it ideal for parallel programming. The drawback of the exhaustive search is its sensitivity against large images because of its exponential complexity. The runtime will drastically increase for high-resolution images with a large number of features.

The approximate searches have many advantages regarding time complexity. However, none of them can guarantee the correctness of the matches provided. Some of the solutions are more sensitive to dimensionality, especially the ones that rely on storing the features in tree structures. Since many of the approximate searches preprocess the data, they will often need a large number of features to be effective. This preprocessing means that if there is no consistency in the input images regarding the resolution, the benefits of the approximate searches can quickly diminish. The solutions that have speedup on all the images will in these cases be the obvious choice. However, most of these have such a low percentage of correct matches to be considered in the first place, as the cascade hashing results in our tests indicated.

We do not have a comprehensive rule to determine whether a NN-only, ANN-only, or hybrid approach will work best, but understanding the differences between NN and ANN allows us to select the appropriate algorithm on a case-by-case basis.

Bibliography

[1] Abraham Adrian Albert. *Solid analytic geometry*. Courier Dover Publications, 2016.

[2] Padmanabhan Anandan. 'A computational framework and an algorithm for the measurement of visual motion'. In: *International Journal of Computer Vision* 2.3 (1989), pp. 283–310.

[3] Sunil Arya et al. 'An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions'. In: *J. ACM* 45.6 (Nov. 1998), pp. 891–923. ISSN: 0004-5411. DOI: 10.1145/293347.293348. URL: http://doi.acm.org/10.1145/293347.293348.

[4] Vassileios Balntas et al. 'HPatches: A benchmark and evaluation of handcrafted and learned local descriptors'. In: *CVPR*. 2017.

[5] Herbert Bay et al. 'Speeded-up robust features (SURF)'. In: *Computer vision and image understanding* 110.3 (2008), pp. 346–359.

[6] Jeffrey S Beis and David G Lowe. 'Shape indexing using approximate nearest-neighbour search in high-dimensional spaces'. In: *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*. IEEE. 1997, pp. 1000–1006.

[7] Nathan Bell and Jared Hoberock. 'Thrust: A productivity-oriented library for CUDA'. In: *GPU computing gems Jade edition*. Elsevier, 2011, pp. 359–371.

[8] Jon Louis Bentley. 'Multidimensional Binary Search Trees Used for Associative Searching'. In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: http://doi.acm.org/10.1145/361002.361007.

[9] Manuele Bicego et al. 'On the use of SIFT features for face authentication'. In: *Computer Vision and Pattern Recognition Workshop, 2006. CVPRW'06. Conference on*. IEEE. 2006, pp. 35–35.

[10] Mårten Björkman, Niklas Bergström and Danica Kragic. 'Detecting, segmenting and tracking unknown objects using multi-label MRF inference'. In: *Computer Vision and Image Understanding* 118 (2014), pp. 111–127. ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2013.10.007. URL: http://www.sciencedirect.com/science/article/pii/S107731421300194X.

[11] Georgia Tech - Aaron Bobick. *RANSAC - General algorithm*. [Online; visited 15-March-2017]. URL: https://classroom.udacity.com/courses/ud810/lessons/3189558841/concepts/31679389260923#.

[12] Gary Bradski and Adrian Kaehler. 'OpenCV'. In: *Dr. Dobb's journal of software tools* 3 (2000).

[13] Matthew Brown and David G Lowe. 'Automatic panoramic image stitching using invariant features'. In: *International journal of computer vision* 74.1 (2007), pp. 59–73.

[14] Matthew Brown and David G Lowe. 'Invariant features from interest point groups.' In: *BMVC*. Vol. 4. 2002.

[15] Matthew Brown, Richard Szeliski and Simon Winder. 'Multi-Image Matching using Multi-Scale Oriented Patches'. In: ().

[16] Jian Cheng et al. 'Fast and Accurate Image Matching with Cascade Hashing for 3D Reconstruction'. In: (June 2014).

[17] Wikipedia contributors. *Nearest neighbor search — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-February-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Nearest_neighbor_search&oldid=824307601.

[18] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series. Elsevier Science, 2012. ISBN: 9780124159884. URL: https://books.google.no/books?id=EX2LNkSqViUC.

[19] Geoffrey Egnal. 'Mutual information as a stereo correspondence measure'. In: *Technical Reports (CIS)* (2000).

[20] R. Farber. *CUDA Application Design and Development*. Applications of GPU computing series CUDA application design and development. Elsevier Science, 2011. ISBN: 9780123884329. URL: https://books.google.no/books?id=Y-XmJO2uwvMC.

[21] Saeid Fazli, Hamed Moradi Pour and Hamed Bouzari. 'Particle filter based object tracking with sift and color feature'. In: *Machine Vision, 2009. ICMV'09. Second International Conference on*. IEEE. 2009, pp. 89–93.

[22] Martin A. Fischler and Robert C. Bolles. 'Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography'. In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: http://doi.acm.org/10.1145/358669.358692.

[23] Yasutaka Furukawa. 'Multi-view 3D reconstruction techniques in computer vision'. In: *CVIM* (2011).

[24] Alfréd Haar. 'Zur Theorie der orthogonalen Funktionensysteme'. In: *Mathematische Annalen, doi:10.1007/BF01456326*. 1910, pp. 331–371.

[25] Mark Harris. 'Optimizing cuda'. In: *SC07: High Performance Computing With CUDA* (2007).

[26] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518, 2004.

[27] Richard I Hartley. 'In defense of the eight-point algorithm'. In: *IEEE Transactions on pattern analysis and machine intelligence* 19.6 (1997), pp. 580–593.

[28] Richard Hartley, Rajiv Gupta and Tom Chang. 'Stereo from uncalibrated cameras'. In: *Computer Vision and Pattern Recognition, 1992. Proceedings CVPR'92., 1992 IEEE Computer Society Conference on*. IEEE. 1992, pp. 761–764.

[29] Heiko Hirschmuller. 'Stereo processing by semiglobal matching and mutual information'. In: *IEEE Transactions on pattern analysis and machine intelligence* 30.2 (2008), pp. 328–341.

[30] Berthold KP Horn and Brian G Schunck. 'Determining optical flow'. In: *Artificial intelligence* 17.1-3 (1981), pp. 185–203.

[31] Gang Hua, Matthew Brown and Simon Winder. 'Discriminant embedding for local image descriptors'. In: *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*. IEEE. 2007, pp. 1–8.

[32] Zhen Hua, Yewei Li and Jinjiang Li. 'Image stitch algorithm based on SIFT and MVSC'. In: *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on*. Vol. 6. IEEE. 2010, pp. 2628–2632.

[33] 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[34] 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*. ACM. 1998, pp. 604–613.

[35] V. Kindratenko. *Numerical Computations with GPUs*. Springer International Publishing, 2014. ISBN: 9783319065489. URL: https://books.google.no/books?id=CbH0AwAAQBAJ.

[36] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2012. ISBN: 9780123914187. URL: https://books.google.no/books?id=E0Uaag8qicUC.

[37] Chuck L Lawson et al. 'Basic linear algebra subprograms for Fortran usage'. In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[38] Stefan Leutenegger, Margarita Chli and Roland Y Siegwart. 'BRISK: Binary robust invariant scalable keypoints'. In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2548–2555.

[39] H Christopher Longuet-Higgins. ‘A computer algorithm for reconstructing a scene from two projections’. In: *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms, MA Fischler and O. Firschein, eds* (1987), pp. 61–62.

[40] David G Lowe. ‘Distinctive image features from scale-invariant keypoints’. In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[41] Bruce D Lucas, Takeo Kanade et al. ‘An iterative image registration technique with an application to stereo vision’. In: (1981).

[42] Jun Luo et al. ‘Person-specific SIFT features for face recognition’. In: *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*. Vol. 2. IEEE. 2007, pp. II–593.

[43] K. Markantonakis and K. Mayes. *Secure Smart Embedded Devices, Platforms and Applications*. Springer New York, 2013. ISBN: 9781461479154. URL: https://books.google.no/books?id=%5C_Ta3BAAAQBAJ.

[44] Bruce J. McKenzie, Rodney Harries and Timothy Bell. ‘Selecting a hashing algorithm’. In: *Software: Practice and Experience* 20.2 (1990), pp. 209–224.

[45] Ajmal S Mian, Mohammed Bennamoun and Robyn Owens. ‘Keypoint detection and local feature matching for textured 3D face recognition’. In: *International Journal of Computer Vision* 79.1 (2008), pp. 1–12.

[46] Ajmal Mian, Mohammed Bennamoun and Robyn Owens. ‘An efficient multimodal 2D-3D hybrid approach to automatic face recognition’. In: *IEEE transactions on pattern analysis and machine intelligence* 29.11 (2007).

[47] Krystian Mikolajczyk and Cordelia Schmid. ‘A performance evaluation of local descriptors’. In: *IEEE transactions on pattern analysis and machine intelligence* 27.10 (2005), pp. 1615–1630.

[48] Krystian Mikolajczyk and Cordelia Schmid. ‘Indexing based on scale invariant interest points’. In: *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*. Vol. 1. IEEE. 2001, pp. 525–531.

[49] Krystian Mikolajczyk and Cordelia Schmid. ‘Scale & affine invariant interest point detectors’. In: *International journal of computer vision* 60.1 (2004), pp. 63–86.

[50] Krystian Mikolajczyk et al. ‘A comparison of affine region detectors’. In: *International journal of computer vision* 65.1-2 (2005), pp. 43–72.

[51] Jean-Michel Morel and Guoshen Yu. ‘ASIFT: A new framework for fully affine invariant image comparison’. In: *SIAM Journal on Imaging Sciences* 2.2 (2009), pp. 438–469.

[52] Pierre Moulon et al. 'OpenMVG: Open Multiple View Geometry'. In: *Reproducible Research in Pattern Recognition*. Ed. by Bertrand Kerautret, Miguel Colom and Pascal Monasse. Cham: Springer International Publishing, 2017, pp. 60–74. ISBN: 978-3-319-56414-2.

[53] Marius Muja and David G. Lowe. 'Scalable Nearest Neighbor Algorithms for High Dimensional Data'. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 36 (2014).

[54] John Nickolls et al. 'Scalable parallel programming with CUDA'. In: *ACM SIGGRAPH 2008 classes*. ACM. 2008, p. 16.

[55] S.Y. Nof. *Springer Handbook of Automation*. Springer Handbook of Automation. Springer Berlin Heidelberg, 2009. ISBN: 9783540788317. URL: https://books.google.no/books?id=2v%5C_91vSCIK0C.

[56] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer Berlin Heidelberg, 2013. ISBN: 9783642378010. URL: https://books.google.no/books?id=UbpAAAAAQBAJ.

[57] Ethan Rublee et al. 'ORB: An efficient alternative to SIFT or SURF'. In: *Computer Vision (ICCV), 2011 IEEE international conference on*. IEEE. 2011, pp. 2564–2571.

[58] Boris Ruf, Effrosyni Kokiopoulou and Marcin Detyniecki. 'Mobile museum guide based on fast SIFT recognition'. In: *International Workshop on Adaptive Multimedia Retrieval*. Springer. 2008, pp. 170–183.

[59] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN: 0131387685, 9780131387683.

[60] Daniel Scharstein and Richard Szeliski. 'A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms'. In: *Int. J. Comput. Vision* 47.1-3 (Apr. 2002), pp. 7–42. ISSN: 0920-5691. DOI: 10.1023/A:1014573219977. URL: http://dx.doi.org/10.1023/A:1014573219977.

[61] Chanop Silpa-Anan and Richard Hartley. 'Optimised KD-trees for fast image descriptor matching'. In: *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. IEEE. 2008, pp. 1–8.

[62] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[63] Philip HS Torr and Andrew Zisserman. 'MLESAC: A new robust estimator with application to estimating image geometry'. In: *Computer Vision and Image Understanding* 78.1 (2000), pp. 138–156.

[64] Bill Triggs et al. 'Bundle adjustment—a modern synthesis'. In: *International workshop on vision algorithms*. Springer. 1999, pp. 298–372.

[65] Andrea Vedaldi and Brian Fulkerson. 'VLFeat: An open and portable library of computer vision algorithms'. In: *Proceedings of the 18th ACM international conference on Multimedia*. ACM. 2010, pp. 1469–1472.

[66] Jinjun Wang et al. 'Locality-constrained linear coding for image classification'. In: *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE. 2010, pp. 3360–3367.

[67] Henry S. Warren. *Hacker's Delight*. 2nd. Addison-Wesley Professional, 2012. ISBN: 0321842685, 9780321842688.

[68] Patrick Wieschollek et al. 'Efficient large-scale approximate nearest neighbor search on the gpu'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2027–2035.

[69] D.S. Wilks. *Statistical Methods in the Atmospheric Sciences*. International Geophysics. Elsevier Science, 2011. ISBN: 9780123850232. URL: https://books.google.no/books?id=fxPiH9Ef9VoC.

[70] Jianchao Yang et al. 'Linear spatial pyramid matching using sparse coding for image classification'. In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE. 2009, pp. 1794–1801.

[71] Alper Yilmaz, Omar Javed and Mubarak Shah. 'Object tracking: A survey'. In: *Acm computing surveys (CSUR)* 38.4 (2006), p. 13.

[72] Guoshen Yu and Jean-Michel Morel. 'ASIFT: An algorithm for fully affine invariant comparison'. In: *Image Processing On Line* 1 (2011), pp. 11–38.

[73] Huiyu Zhou, Yuan Yuan and Chunmei Shi. 'Object tracking using SIFT features and mean shift'. In: *Computer vision and image understanding* 113.3 (2009), pp. 345–352.