Comparing LSH and brute-force on GPU for SIFT feature point matching

Scott Mathias Richards

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2022

Comparing LSH and brute-force on GPU for SIFT feature point matching

Scott Mathias Richards

Comparing LSH and brute-force on GPU for SIFT feature point matching

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Scale-invariant feature transform (SIFT) is a computer vision algorithm which is able to detect the same features in differing pictures despite 2D rotation, image scaling, translation, and to some degree 3D rotation. SIFT has been shown to be accurate, fast and reliable. Matching of SIFT feature points is still a very compute-heavy task, with no obvious solution fast enough for real-time applications when dealing with high definition images. We implement two algorithms to match SIFT points on GPU with the GPU programming interface CUDA, and the libraries cuBLAS and Thrust, and explore using CUDA's half-precision (16 bit) for the values in the SIFT vectors. The first implementation is of Locality-sensitive hashing (LSH). LSH is an approximate nearest neighbor (ANN) algorithm, which reduces the search space by hashing values which are close in proximity into the same buckets, at a higher probability than values that are further apart. We show that on GPU for SIFT feature point matching at 0.8 recall, this implementation can get a 1.23X speed up when looking for the nearest neighbor (NN) and a 10.48X speed up when looking for any of the 100NNs compared to a naive CUDA brute-force when matching on the ANN_SIFT1M data set. The second implementation is a brute-force using CUDA's cuBLAS library where we use the dot product, achieving a 86X speed up over the naive CUDA brute-force on the ANN_SIFT1M data set. On an RTX 3060 12 GB it can match 10^4 query vectors and 10^6 reference vectors in 177ms, and can match SIFT points in real time at 45k query points and 45k reference points at 27 fps. We also show that using `halfs` as compared to `floats` in CUDA gives a speed up of up to 5X, while only reducing recall by 0.0056 on the ANN_SIFT1M data set when used in a brute-force.

Contents

List of Figures

List of Tables

Preface

Chapter 1

Introduction

Typically, mobile phone Augmented Reality (AR) applications just render virtual objects into space at the intended position. If anyone walks in front of that intended position, these applications will render the scene as if the person is behind the object. For a mobile phone that may be acceptable, but it is very irritating when you watch an AR scene through a head-mounted display (HMD). Knowing the depth of the objects would be a good starting point to solving this problem. To get the depth of objects in a image using two cameras (stereo vision) and Scale-invariant feature transform (SIFT) [11] feature points has been shown to be a viable solution [14]. SIFT is a computer vision algorithm which is able to detect the same features in differing pictures despite 2D rotation, image scaling, translation, and to some degree 3D rotation [11]. SIFT features from two cameras could be used to identify the same feature points in two given frames. Given the x and y cords of these points and the location of camera one in relation to camera two, one is able to calculate the Euclidean distance between the cameras and the object. Knowing the depth, it should be possible to occlude objects when necessary. To do this in real time, being able to perform the computations fast is an absolute necessity. SIFT has been shown to be accurate, fast and reliable. Furthermore, a GPU-accelerated implementation of SIFT has been shown to be able to locate and extract key points in real time [6]. However the matching of SIFT feature points is still a very compute-heavy task, with no obvious solution fast enough for real time applications when dealing with high definition images.

1.1 Purpose of thesis

In this thesis we explore two solutions for matching SIFT feature points on a graphics processing unit (GPU). General computation on GPUs have gained a lot of traction lately, one interface to do this is Compute Unified Device Architecture (CUDA) which is what we use in this thesis. The first solution for matching SIFT feature points we look at is the approximate nearest neighbour (ANN) algorithm Locality-sensitive hashing (LSH) [8]. LSH works by hashing close points into buckets, reducing the search space of the brute-force to points which land in the same buckets, i.e have hash

values which collide. The second approach for matching we look at, is a brute-force using the dot product with matrix - matrix multiplications [5]. The biggest motivation here being the cuBLAS library, which has very fast matrix - matrix algorithms. GPUs excel at tasks which require work which can be done in parallel. Both LSH and the brute-force with the dot product have parts which can be done well in parallel making them suitable for GPU. CUDA offers 2 libraries we use, cuBLAS and Thrust. cuBLAS is CUDA's Basic Linear Algebra Subroutine (BLAS) implementation and is very optimized. Thrust is a CUDA library designed for productivity, it is easy to use in the sense that the user does not need to have any understanding of the underlying architecture of the GPU or CUDA. Thrust offers much of what C++ STL offers like sort, reduce and the ability to use vectors in much the same way we use them in C++. CUDA on newer architectures also offers using 16 bit floating points referred to as half-floats or just `halfs`. Because CUDA offers two-way Single instruction multiple data (SIMD) for half-precision, using `halfs` should give a at least a 2X speed up over using `floats`, in theory. However this is at the cost of some accuracy as we go from 32 bit floating points to 16 bit floating points. If using `halfs` is viable for our two implementations is also something we look at.

1.2 Research questions

The research questions we aim to answer are:

- Can locality sensitive hashing be efficiently implemented with CUDA to match SIFT vectors with good recall, and how will such an implementation compare to a brute-force CUDA implementation in terms of speed?
- Can Thrust and cuBLAS be used when implementing LSH, and how will using these CUDA libraries affect performance and complexity?
- CUDA is optimized for half-precision (16 bit), is using `halfs` instead of `floats` when dealing with SIFT vectors viable, and how does using `halfs` when matching SIFT vectors affect recall and performance?
- Can an efficient brute-force for SIFT vectors be implemented with the help of cuBLAS in CUDA, and will such an implementation be able to perform SIFT vector matching in real time?

1.3 Research methodology

According to the ACM Task Force on the Core of Computer Science in the paper "Computing as a discipline" [3] and Amnon H. Eden in "Three Paradigms of Computer Science" [4], computer science in general has 3 paradigms. Both Eden and ACM describe three more or less identical paradigms, as follows:

- The rationalist paradigm Eden [4] or theory ACM [3]. This paradigm seeks a priori knowledge of objects or systems by deductive reasoning. This paradigm is rooted in mathematics and can be defined to have the following 4 steps [3]:

  1. characterize objects of study (definition);
  2. hypothesize possible relationships among them (theorem);
  3. determine whether the relationships are true (proof);
  4. interpret results.

  This is an iterative process where if inconsistencies are found, revision is necessary.

- The technocratic paradigm Eden [4] or design ACM [3]. This paradigm seeks probable, a posteriori knowledge about programs or systems through implementation/prototyping and testing. This paradigm is rooted in engineering and can be defined to have the following 4 steps [3]:

  1. state requirements;
  2. state specifications;
  3. design and implement the system;
  4. test the system.

  This is an iterative process where if the tests results do not satisfy the requirements, the implementation must be revised.

- The scientific paradigm Eden [4] or abstraction ACM [3]. This paradigm is based on the scientific method where we use both a priori and a posteriori knowledge to say something about a model/system. This paradigm can be defined to have the following 4 steps [3]:

  1. form a hypothesis;
  2. construct a model and make a prediction;
  3. design an experiment and collect data;
  4. analyze results.

  This is an iterative process where if the results of the experiment do not confirm the hypothesis/prediction, the hypothesis/prediction must be revised.

Choosing a research methodology

To decide on a research methodology, we need to understand what metrics we need to answer the research questions. The questions revolve around two things, firstly efficiency in terms of time to complete (as in the time needed by one of our implementations to match all query and reference SIFT vectors), and secondly recall (as in the number of relevant matches

we find). More precisely, we want to see how fast our implementations can get good recall (we define good recall as over 0.8) when matching SIFT vectors. Measuring time to complete and recall is fairly straightforward. For time measurement we can use the CPU system time, and for recall a set of true values found with a brute-force will suffice. We also need some sort of benchmark since judging the efficiency of any approach is hard without drawing a comparison. As a benchmark we use a naive CUDA brute-force 2NN implementation.

While ACM notes that all 3 paradigms are intrinsically intertwined, the paradigm most suitable for our problem given the metrics we use is The technocratic paradigm or design. We would not be able to know how well our implementations could meet our requirements with a priori knowledge alone (The rationalist paradigm/theory), nor does it seem fitting to form a hypothesis which predicts how well our implementation could meet our requirements (The scientific paradigm/abstraction). Rather we want to implement a prototype, test to see how well, said prototype meets our requirements, then make changes to our implementations according to the test results.

Our approach

More formally our approach is:

- Write a design which works as an outline or base for our implementations.
- When implementing we use this design as a base, but as we test, we change the implementation according to the test results if we find that this gives better results.

This is an iterative process where as we continuously test we want to both keep good recall and minimize time used by the implementations on different test data.

1.4 Structure of thesis

This paper has 5 main chapters. They are Background, Design, Implementation, Evaluation and test results and lastly Conclusion and future work. In chapter 2 (Background) we introduce the background and related topics of our thesis. In chapter 3 (Design), we look at how we can implement LSH on GPU, a brute-force using the dot product and cuBLAS on GPU, and we discuss using `halfs` instead of `floats` for SIFT vectors. In chapter 4 (Implementation) we go over how we implemented our prototypes, both LSH and the brute-force with cuBLAS. In Evaluation and test results which is (chapter 5) we show the test results and compare our solutions. In chapter 6 (Conclusion and future work), we sum up the answers to our research questions, reflect on the thesis as a whole, and lastly talk about some possible future work.

Chapter 2

Background and related work

In this chapter we introduce the background and related work for this thesis. This includes Scale-invariant feature transform (SIFT) (section 2.1), the k-nearest neighbors (kNN) problem and some kNN algorithms (section 2.2), The curse of dimensionality (section 2.3), approximate nearest neighbors (ANN) (section 2.4), Fast Library for Approximate Nearest Neighbors (FLANN) (section 2.5), Locality-sensitive hashing (LSH) and some LSH variants (section 2.6), and lastly we introduce the basics of CUDA (section 2.7) and the two CUDA libraries cuBLAS (section 2.8) and Thrust (section 2.9).

2.1 Scale-invariant feature transform (SIFT)

Scale-invariant feature transform (SIFT) is a feature detection algorithm. The features found by SIFT are shown to be invariant to 2D rotation, image scaling and translation. In addition, they are to some degree invariant to 3D rotation. The algorithm was introduced by David G. Lowe in the paper "Distinctive Image Features from Scale-Invariant Keypoints" [11]. The SIFT algorithm consists of two main stages: key point detection and feature point extraction. This summary of how SIFT works is based on David G. Lowe's 2004 paper "Distinctive Image Features from Scale-Invariant Keypoints" [11] and "Anatomy of the SIFT Method" [20] by Ives Rey-Otero and Mauricio Delbracio.

2.1.1 SIFT algorithm

The SIFT algorithm starts by upscaling the input image; this is usually done with bilinear interpolation. The upscaling factor is normally set to 2, which gives us a new image with twice the width and height of the original image. This is somewhat optional where the trade-off for not upscaling is speed at the cost of accuracy. Gaussian blur is applied to the upscaled image, usually by a factor of $\sigma = 1.6$. We repeat the blur, giving us a number of more and more blurred images, we then take the third to last of this row and downscale it and repeat the process. We continue this process until

we can no longer downscale the image, as illustrated in figure 2.1. Each downward row of downscaled images is called an octave.

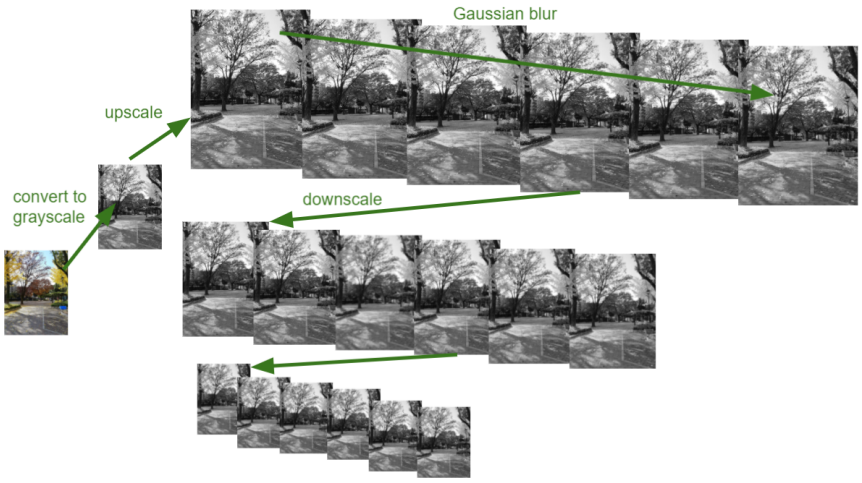

Figure 2.1: Illustration of SIFT steps: Initial upscaling, add blurring and downscaling

We are now left with a scale space that simulates the different scales of observation. We then think of our scale space as a continuous three-dimensional space. Given the x, y coordinate of the pixel and σ, we then calculate the Difference of Gaussians between the adjacent pictures, leaving us with a new set of images. On this new set, we find the extrema by comparing a pixel with its 26 neighbours. All the extrema we find are potential key points. As the extrema we have found have discrete coordinates. We try to refine this by approximating the Taylor expansion of the scale-space function for each extremum. During the process, we are sometimes not able to refine the location in the number of iterations we want to, in which case we either discard or keep the extrema as is.

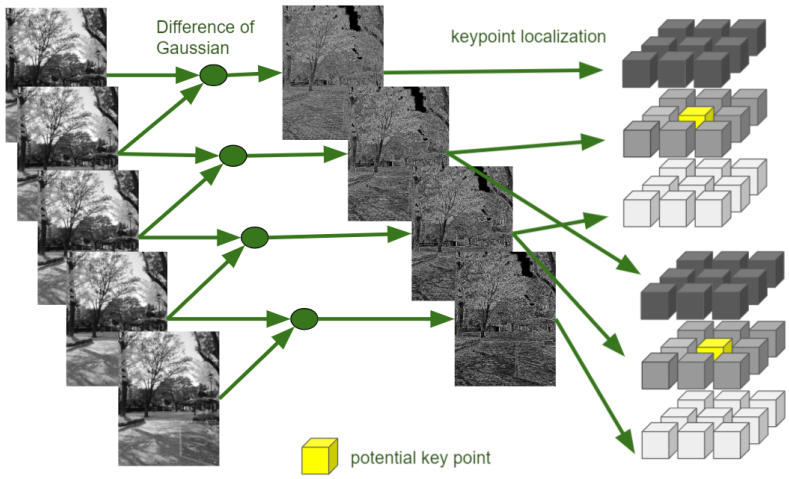

Figure 2.2: Illustration of SIFT steps: Find DoG, check for extrema

We then try to identify edges, as they are not ideal key points. To do this, we find the edge extrema by comparing the principal curvatures of the projection onto the picture plane created by the scale-space function. We can now discard more low-contrast points as we have a better understanding of the position. We then assign a reference orientation to the remaining key points. This is done by approximating the gradients of the pixels in a square-shaped patch around the keypoint. The 360° are divided into 36 bins, each representing 10°. Before the size of the gradient is added, it is multiplied by the Gaussian weight. We are now left with a histogram, which will be smoothed with box filtering before the extremum is identified and selected if the threshold is met. Then a better reference orientation is calculated using quadratic interpolation on the extrema and the two neighboring bins. Key points with more than one dominating orientation may be turned into new keypoints, one for each dominating orientation (at most 3 orientations).

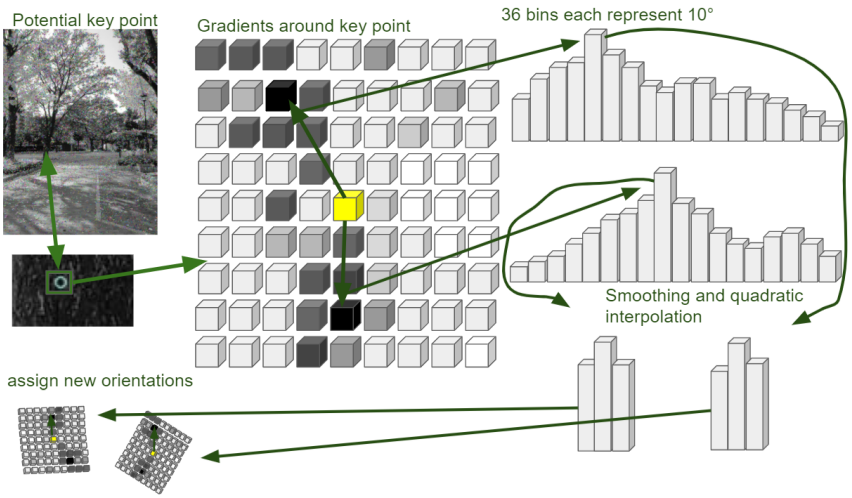

Figure 2.3: Illustration of SIFT steps: Find dominant gradient/s, assign reference orientation

Lastly, we compute 16 more histograms where each histogram has 8 bins. This time, the histograms are computed from a circular patch around the keypoint. Before we do so, we rotate the coordinate system to match the orientation. Each of the new histograms corresponds to a point near the center of the newly made coordinate system. The bins are normalized and we are now left with our key point descriptor, which is represented with 128 float values.

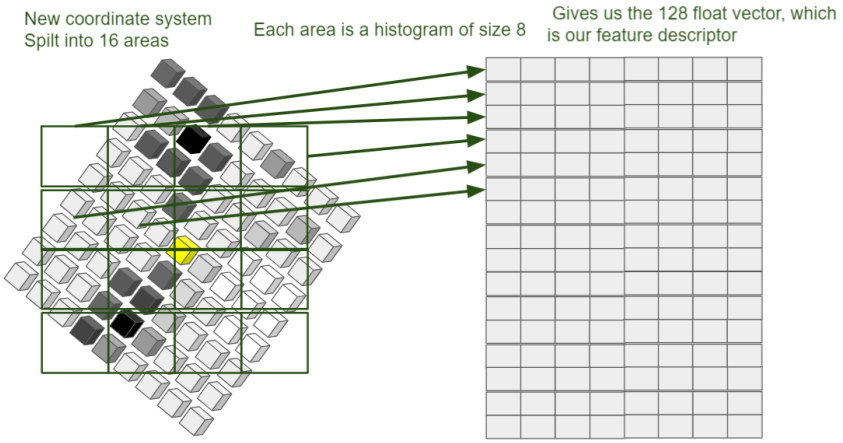

Figure 2.4: Illustration of SIFT steps: 16 regions around a keypoint are used to create 16 histograms with 8 bins each

2.1.2 SIFT descriptors

This descriptor is shown to be largely invariant to 2D rotation, image scaling and translation. In addition, 3D rotation works up to some degree [11]. The invariance to 2D rotation is caused by computing the key points relative to an orientation reference. It is Illumination-invariant because the vectors are normalized, saving only the relation between the gradients. It is invariant to scaling because the key points are extracted at different blur and scales. It is to some degree invariant to rotation in 3D because of the way in which the gradient weights are distributed over the histograms for the 16 cells. The gradient's weight is determined by its distance from the keypoint itself, following a Gaussian distribution. This weight is then distributed over up to 4 cells, where the gradient's distance from the cells' centers is used for a linear distribution of the weight between the cells.

SIFT vectors are usually stored as 128 `float` long vectors. As the SIFT descriptors have a norm of 1, each `float` value is in the range $[0, 0.5]$. However they are often represented as 128 element long arrays of `unsigned chars`. This is done by multiplying every value in the vector by 512. This can done as we are not interested in the values alone, but rather the relation between the values between the descriptors. This relation will not change as long as we perform the same actions on all vectors.

2.1.3 SIFT feature-point matching

After the SIFT algorithm is done, we have set of 128-dimensional vectors, we now usually want to match our vectors with another set of SIFT vectors. By matching, we mean finding the points that are the closest using Euclidean distance. When matching SIFT vectors to reduce wrong matches, we often check if the nearest neighbor is sufficiently close compared to the second nearest neighbor. Because of this we need to find the 2 nearest

neighbours. This is a classic k-Nearest Neighbor (kNN) problem (or to be more specific in SIFT's case, it's a 2-Nearest Neighbor (2NN) problem).

2.1.4 SIFT distance ratio threshold

As mentioned, when matching SIFT feature-points we have a somewhat optional step where we see if the closest neighbour is close enough, compared to the second closest neighbour. If this is the case we considered the closest neighbour a match. This is done by looking at the distance ratio, where if the distance ratio is above some threshold we count it as a match and if not we don't. Assuming d_1 is the distance from the query point to the closest neighbour, d_2 is the distance from the query point to the second closest neighbour, and the distance ratio is d_r then we can get the distance ratio by:

$$d_r = \frac{d_1}{d_2}$$

According to Lowe [11] rejecting all matches where the distance ratio is above 0.8 will lead to removing 90% of all false matches, while only removing 5% of true matches.

2.2 k-Nearest Neighbor (kNN)

The k-Nearest Neighbor (kNN) problem can be defined as follows.
If we have a metric space S in a D-dimensional l_P space, and a data base A where $A \subseteq S$. Given a query point q where q is a vector in the D-dimensional space i.e $q = (q_1, ..., q_D)$ and $q \in S$. The k-nearest neighbors algorithm $kNN(q)$ will return the k nearest points to our query point q for some norm L_p from the database A.

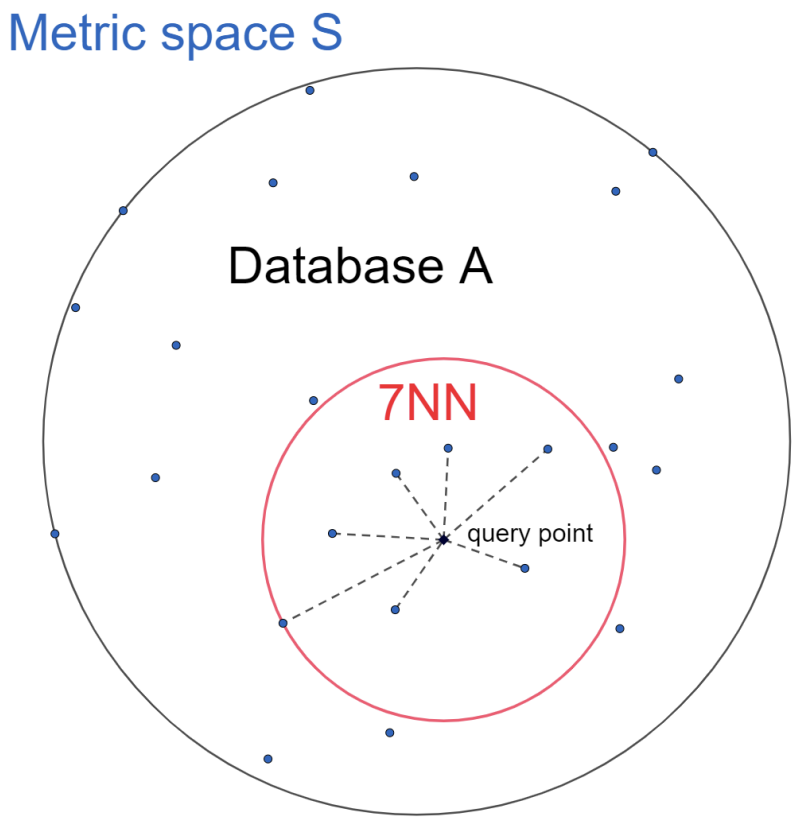

Figure 2.5: Illustration of the kNN problem in a R^2 space with k = 7 using Euclidean distance

A naive brute-force approach for $kNN(q, k, A)$ could look like this:

1. Initialize a dist array of same size as data set A.
2. Loop through all elements in A while filling up the dist array with the distance between q and A[i].
3. Initialize the output array which is an array of length k.
4. Iterate though the dist array k times, for each iteration choose the smallest element, add the index to the output array and remove it from the dist array.
5. Return the output array

The complexity of this naive approach would be $O(nd + nk)$. This is because for step 2 we need to iterate thorough n elements, then for each iteration we need to calculate the distance d, giving us $O(nd)$. For step 4 we need to iterate over all n elements k times giving us $O(nk)$. Add step 2 and 4 and we get $O(nd + nk)$. A still naive but slightly better approach

could achieve $O(nd)$ by using for example quick select at step 4. Quick select can find the k smallest element/s in $O(n)$ time, which would give us $O(nd + n)$ which we write as $O(nd)$.

2.2.1 K-dimensional trees

There are more efficient approaches for kNN in lower dimensions, one such approach is the k-dimensional tree. The k-dimensional tree was first introduced by Jon Louis Bentley in the paper "Multidimensional binary search trees used for associative searching"" [1].

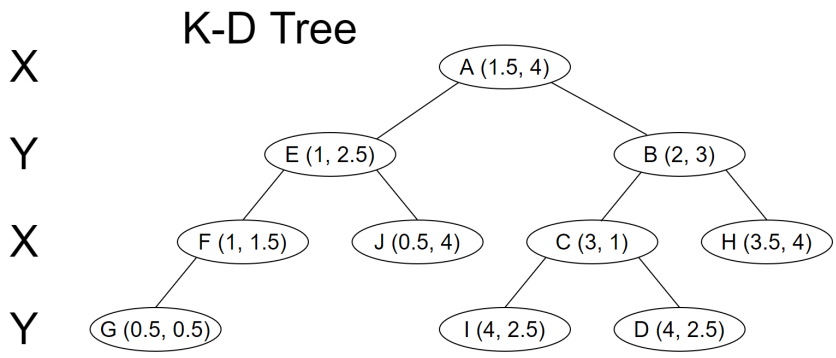

Figure 2.6: Illustration a K-D tree in R^2 space where the nodes are inserted in alphabetical

A k-dimensional tree is a binary tree data structure used to store k-dimensional data points. Each non-leaf node can be seen as a hyperplane splitting the data.

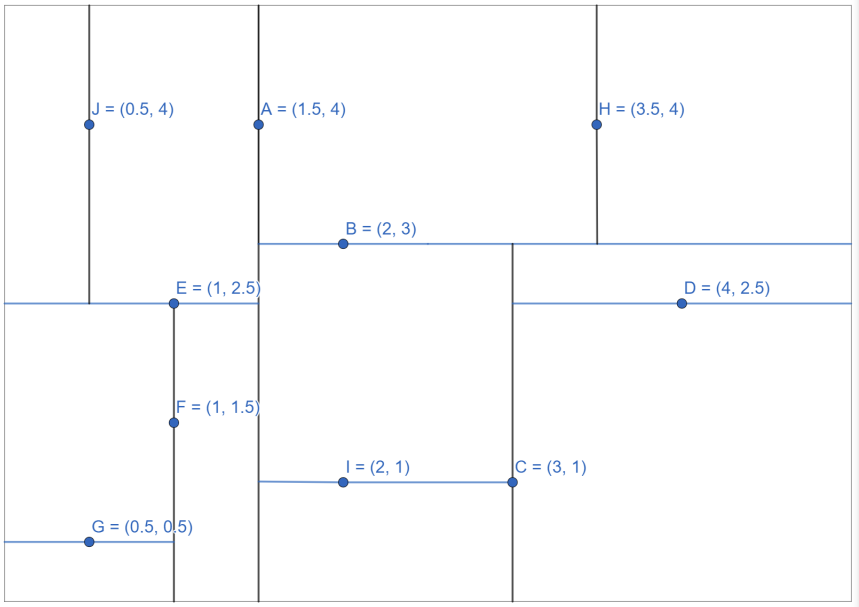

Figure 2.7: Illustration of how the k-d tree illustrated in 2.6 partitions R^2 space

When inserting a data point, you iterate between the dimensions at the different levels of the tree. Whether the node goes to the left or the right is only dependent on the value in the dimension for the level of the tree you are at. When doing a nearest neighbour search on a k-dimensional tree, the algorithm goes as follows:

1. Start at the root node, do as when you insert a node, compare the first dimension with the root node, go left or right depending on the value. In the next level of the tree you compare the next dimension. Continue this until you reach a leaf-node.

2. Check if the distance to the leaf node is better than the current best distance.

3. We then unwrap the recursion and do the following things at each node

   (a) Check if the current node is closer than current best. If closer set new current best.

   (b) Check if there could be any closer points than the current best on the opposite branch of the route we originally took. This is done by checking if a hypersphere around our search point with radius of current best is intersecting the hyperplane splitting the branch.

   (c) If this is not the case, we can discard that branch and all its nodes as we know none of them can be better than our current best. (If we were to use this algorithm for *k*NN we would have to check with the radius of our *k*-nearest neighbour instead)

(d) If it is intersecting, there may be a point closer than the current best so we have to repeat the algorithm for the branch.

4. When we return to the root node, we are done and should have the NN.

The average Big O complexity of a k-dimensional tree search is $O(\log nd)$ where n is the number of points in the k-dimensional tree and d is the time it takes to calculate the distance to one point in n. The worst case will be the same as brute force $O(nd)$.

2.2.2 *k*NN on GPU

As there are a lot of calculations that can take place independently, the *k*NN problem is highly parallelizable and would thus benefit greatly from a GPU implementation. One such implementation using CUDA is proposed by authors Vincent Garcia, Eric Debreuve, Frank Nielsen and Michel Barlaud in the paper "k-nearest neighbor search: fast gpu-based implementations and application to high-dimensional feature matching" [5] and works as follows:

1. Use two kernels, the first kernel calculates the distance matrix from Q query points to the R reference points. This is done completely in parallel as the distances between points are independent. Each thread calculates the distance between a given query point q and reference point r.

2. The second kernel sorts the distance matrix. One thread is run for each query point q, this thread sorts the distance matrix corresponding to q

This GPU implementation was shown to give up to a 25X speed-up compared to FLANN [13] a highly optimized C++ ANN implementation on high-dimensional SIFT feature matching [5].

Another brute force GPU implementation of the *k*NN problem using matrix operations with the CUDA library cuBLAS is also proposed [5]. cuBLAS is a highly optimized linear algebra vector/matrix library (see section 2.8). To take advantage of the cuBLAS library, the calculation of the squared Euclidean distance p^2 from point x to y is done:

$$p^2(x,y) = (x-y)^T(x-y) = ||x||^2 + ||y||^2 - 2x^Ty$$

where the T is the transpose. If we rewrite it for two matrices, R which is a $d \times m$ matrix and Q which is a $d \times n$ matrix we get the equation:

$$p^2(N_R, N_Q) = N_R + N_Q - 2R^TQ$$

here N stands for the norm, and the resulting matrix $p^2(N_R, N_Q)$ would be a $m \times n$ matrix. The GPU brute force algorithm which uses 8 kernels with cuBLAS and CUDA, goes as follows:

1. Kernel 1 calculates N_R using CUDA (coalesced read/write)
2. Kernel 2 calculates N_Q using CUDA (coalesced read/write)
3. Kernel 3 uses cuBLAS to calculate $-2R^TQ$ we call the resulting matrix A
4. kernel 4 adds all elements in N_R to A we call the new matrix B. this is done with CUDA threads and shared memory.
5. kernel 5 sorts the B matrix in parallel with n threads. giving us the matrix C
6. kernel 6 adds the j^{th} element of N_Q to the k first elements of the j^{th} column of the matrix C. this is done using CUDA (coalesced read/write). We call the new matrix D
7. Kernel 7 computes the square root of the first k elements in D, this gives us the k smallest distances, this is done using CUDA (coalesced read/write). We call the new matrix E
8. The last kernel extracts the $k \times n$-submatirx from E. This is the matrix of the distances for each query point.

This method is shown to be up to 62X faster than FLANN [13] when testing with SIFT feature matching [5].

2.3 The Curse of dimensionality

For lower dimensions the *k*NN problem can be solved efficiently using methods like k-dimensional trees, however when the dimensions increase the efficiency of such approaches quickly degrade. This is due to something often referred to as "The Curse of dimensionality". The curse of dimensionality is a phenomena which occurs with the increase of dimensions in data. The degrade in approaches such as k-d trees are caused because the relevance of one dimension for the Euclidan distance decreases with the increase of dimensions [2]. Therefore when the dimensionality increases k-d trees and k-d tree like approaches will often have to check all possible points to find the NN. This gives us equal or worse performance than we would get from a naive brute-force. Therefore in higher dimensions finding the approximate nearest neighbour (ANN) and not the nearest neighbour is often necessary to out-perform brute force.

2.4 Approximate nearest neighbour

As mentioned in section(2.3) most sophisticated *k*NN solutions will with the increase of dimensions degrade to naive brute force, therefore we often need to use approaches which find the approximate nearest neighbour (ANN) instead of the nearest neighbour (NN) to outperform a naive brute force implementation. There exists many approximate nearest neighbour

algorithms with various trade-offs, some trade accuracy for speed while others are very task/data dependent.

2.5 Fast Library for Approximate Nearest Neighbors (FLANN)

FLANN or "Fast Library for Approximate Nearest Neighbors" is a known C++ library for ANN that uses a variant of k-dimensional trees. It was introduced in the paper "Fast approximate nearest neighbors with automatic algorithm configuration" [13] by Marius Muja And David G. Lowe. The paper introduces the 2 main ANN algorithms used in the library while also showing results from different experiments. Which of the 2 algorithms are best suited for a given query is dependent on the input and on a few parameters decided by the user. In the paper it is shown that the two algorithms that give the best result depending on the input and desired accuracy are hierarchical k-means tree or multiple randomized kd-trees. In this section we introduce the 2 main algorithms in the paper.

2.5.1 Randomized kd-trees

Pure kd-trees are efficient in low dimensions, but the performance quickly degrades in higher dimensions. Therefore finding the approximate nearest neighbour is necessary to out-perform brute force. One of the two algorithms Muja and Lowe chose to use is randomized kd-trees. A randomized kd-tree will at random choose the dimensions used for splitting out of a set of the dimensions that give the biggest variance. Muja and Lowe state that using the 5 dimensions that give the biggest variance is sufficient and that there is little to no point in using more [13]. The precision is decided by how many leaf nodes you want to check. This is not decided by the user but rather the user will specify a desired accuracy and the algorithm will train on the data to see how many leaf nodes it needs to visit to achieve said accuracy.

2.5.2 K-means trees

The second algorithm Muja and Lowe use is hierarchical k-means trees. This algorithm works by splitting the data into K distinct clusters using k-means clustering. This is then repeated recursively on each cluster until the number of points in each is smaller than K. To traverse the k-means tree, the algorithm uses a best bin first approach where it first traverses the tree once, adding all unexplored branches into a priority queue. It then selects the branch with the center closest to the query point and repeats the same process from said branch. The accuracy is decided in the same way as with the randomized kd trees where the user supplies the accuracy and the algorithm trains on data to see how many nodes need to be traversed to achieve it.

2.5.3 How FLANN chooses ANN algorithm

Which of the two algorithms is the most efficient is highly dependent on a few different factors: memory usage, build time (if it is relevant) and accuracy. Also, the parameters for each individual algorithm is fine tuned by the library. This is an optimization problem which (just as with choosing the number of leaf nodes to traverse for a certain accuracy) will be done by training on the data set. How much of the data set you want to train on is up to the user, but it is stated that training on 1/10 of the training set is sufficient to get close to optimal parameters [13]. However, this will vary somewhat on the data sets.

2.5.4 Notes on Randomized kd-trees

The experiment results for multiple randomized kd-trees shows that by increasing the number of trees, we can increase the performance, however when we reach over 20 trees, there is no benefit from increasing the number anymore. Also the more trees used the more memory is used, therefore even though using around 20 trees is shown to give the best accuracy, in some cases when memory is important, the optimization algorithm may choose a lower number.

2.5.5 Notes on k-means trees

The biggest problem with the hierarchical k-mean trees is the build time. However, the build time can be reduced drastically by setting a max number of iterations for k-means rather than running to convergence. The experiments show that by running 7 iterations, we still get 90% of the performance of the nearest neighbor algorithm, however the build time is reduced to 10%.

2.5.6 Test results

The paper also shows test results for tests with SIFT feature points. It shows that for SIFT, k-means trees seem to work best on a 100k data set, but when the set goes up to 31M, rand trees seem to be a bit better. It also shows that the larger the set of features points, the bigger the speedup over brute-force is. It also shows that the performance is better on a set that has true matches vs one where there are no true matches. The paper clearly shows that there is a speed-up to be achieved by implementing an ANN algorithm over linear search on CPU, and that this is true even when we want high accuracy.

2.6 Locality-sensitive hashing

Locality-sensitive hashing (LSH) was first introduced by Piotr Indyk and Rajeev Motwani in the paper "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality" [8]. LSH is an ANN algorithm

which uses hashing. Hashing is a great way to lookup items in O(1) time, however this is really only possible if we are looking for exact matches. In SIFT feature matching and other kNN problems this is not the case, rather we have to find the closest match in an often extremely large set of points. LSH (Locality-Sensitive Hashing) solves this by hashing points in the same proximity into buckets. Buckets will in this context mean a hash value, and all points mapped to the same bucket will be points who cause a collision after being hashed.

2.6.1 Locality-sensitive hashing intuitively

This explanation of LSH is inspired by the paper "Locality-Sensitive Hashing for Finding Nearest Neighbors" [21] by Malcolm Slaney and Michael Casey. A very intuitive way of thinking about how LSH works is by picturing a 3D sphere with points inside of it. When you project the sphere onto a 2D surface some points inside the sphere will appear close in proximity. If you rotate the sphere randomly and project the sphere onto a 2D surface again, chances are "true" close neighbours will appear close to each other again, while "false" close neighbours will not. This is true for any dimension projected into a lower dimension. There will obviously be a lot of false neighbours for a big set of points. However, if you repeat the process multiple times and keep a list of the results, chances grow with the number of times you project into the lower dimension that you end up with a list of "true" close neighbours. You then search this list for the nearest neighbour instead of the whole data set. This is the essence of how LSH works.

2.6.2 LSH hash functions

For any hash function to be of use in LSH, it must have the property that it is more probable to put close neighbours into the same buckets rather than not so close neighbours. This is often referred to as $(r_1, r_2, p_1, p_2) - sensitive$ [8]. Given some metric space S with some metric for distance d, two distances r_1 and r_2 where $r_1 < r_2$, and two probabilities p_1 and p_2 where $p_1 > p_2$. For a hash family H to be $(r_1, r_2, p_1, p_2) - sensitive$ in S for d it must fulfill: For any $x, y \in S$

- if $d(x, y) \leq r_1$ then $Pr_H[h(x) = h(y)] \geq p_1$
- if $d(x, y) \geq r_2$ then $Pr_H[h(x) = h(y)] \leq p_2$

Meaning that, the probability of 2 close points $d(x, y) \leq r_1$ being hashed into the same bucket has to be higher than the probability of 2 faraway points $d(x, y) \geq r_2$ being put in the same bucket. There are many $(r_1, r_2, p_1, p_2) - sensitive$ hash-families, the by far most used for hash function for LSH is the dot product. Intuitively the dot product is $(r_1, r_2, p_1, p_2) - sensitive$ because a bigger part of the Gaussian distribution for a random vector will lead to two close points vs two not so close points ending up in the same proximity after the dot product. This is true

because of the linearity of the dot product, meaning the difference between two points $||h(x) - h(y)||$ will have a distribution whose magnitude is proportional to $||x - y||$ which means that when we use the dot product as hash $(p_1 > p_2)$ [21] .

2.6.3 LSH accuracy

When using the dot product as hash function in high-dimensional space, p_1 (the probability that the hash function h hashes 2 close point to the same value) will in all probability not be that much higher than p_2 (the probability that the hash function h hashes 2 not so close point to the same value). To make this difference bigger, LSH is often run for k rounds with different random vectors for the dot product. This changes the chance of ending up with true neighbours vs the chance of ending up with false neighbours from (p_1/p_2) to $(p_1/p_2)^k$. One downside to this is that the true closest neighbour now has a p_1^k chance of ending up in the same bucket. This is because it now has to end up in the same bucket k times. To increase the chances and make up for an unlucky dot-product, we can run the algorithm independently multiple times. Doing this lets us adjust what accuracy we want.

2.6.4 Short list search

When the points are divided into buckets, the search space is greatly decreased. This is because you only search for the closest neighbour within the bucket your query point ends up in. This can be done by brute-force or any other kNN algorithm and is often referred to as a "short list search". Also this is usually the main bottleneck of the LSH algorithm.

2.6.5 Formal LSH definition

Basic LSH in Euclidean space can be defined as: Given a metric space S in a D-dimensional Euclidean space, and a data base A where $A \subseteq S$. Given a query point q where q is a vector in the D-dimensional space $q = (q_1, ..., q_D)$ and $q \in S$. Then we can show our LSH hashing function as:

$$H(q) = \langle h_1(q), h_2(q), ..., h_m(q) \rangle$$

The hash function can hash any given value in S into some M-dimensional hash space. Elements that end up in the same cell (hash value/bucket) or close cells are potential closest neighbors. Each hash function $h_i()$ corresponds to a dimension in the space. And the most common function for $h_i()$ is:

$$h_i(q) = \lfloor \frac{x_i \cdot q + b_i}{w} \rfloor$$

Here x_i is a random D-dimensional vector where each value is drawn from a Gaussian distribution e.g $N(0, 1)$. b_i is drawn from the uniform distribution $U(0, W)$. W is width of the quantization bin, $\lfloor \cdot \rfloor$ is the floor operator and $\cdot$ is the dot product.

2.6.6 LSH variants and improvements

Many improvements and variants of the LSH algorithm have been proposed. Using lattices especially the E8 lattice as hash function is proposed as an improvement over the dot product (random projections to 1 dimension) for Euclidean spaces as shown in the paper "Query-Adaptative Locality Sensitive Hashing" [10]. Another improvement proposed is Multi-Probe LSH.

Multi-Probe LSH

Multi-Probe LSH was introduced in the paper "Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search" [12] by Qin Lv, William Josephson, Zhe Wang, Moses Charikar and Kai Li. Multi-probe LSH works by also searching buckets which are considered "close" up to +1 -1 away where the metric for closeness depends on the hash space. If we use hamming space, the neighbours will be the bit values which are one bit flip away, i.e has a hamming distance of 1. Searching buckets with a distance of 1 away from the query point, can lead to reducing the number of hash tables needed to achieve some recall by 14X - 18X. Meaning we can greatly reduce the needed memory.

2.7 CUDA

General computation on GPUs has gained a lot of traction lately, and Compute Unified Device Architecture (CUDA) is an interface used for this purpose. GPUs excel at tasks which can be done well in parallel. In this section we will give a brief overview of the main aspects of CUDA programming. The information is based on NVIDIA's documentation. especially "CUDA C++ Best Practices Guide - Design Guide" [16] and "CUDA C++ Programming Guide - Design Guide" [17].

2.7.1 CUDA memory hierarchy

The most important thing to keep in mind when using CUDA is the memory hierarchy, since this is where most of the performance can be gained or lost. In CUDA we refer to the GPU as the "device" and the CPU as the "host". Both the host and device usually have their own memory spaces, although on some laptops and devices this can be shared. This means that if our host/device do not share the same memory space, and we want the device to work on something that is in host memory, it has to be explicitly transferred from host to device. However, this can be hidden from the user by using `CudaMallocManaged` in which case the compiler will transfer the data for us.

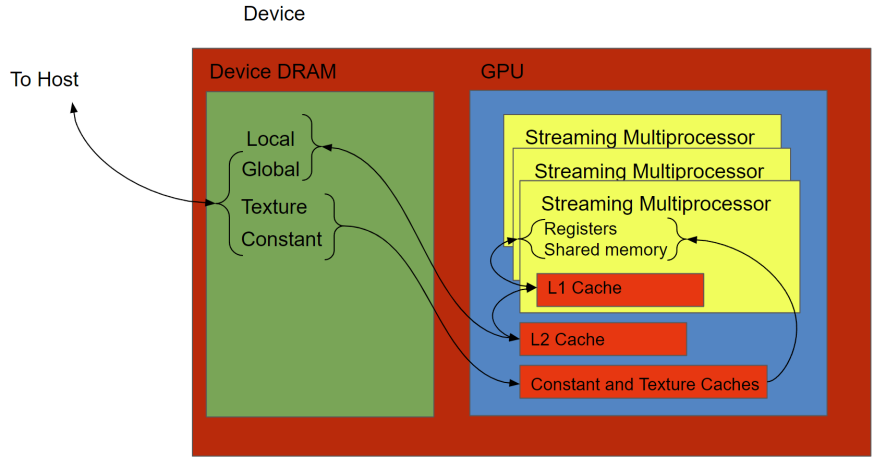

Figure 2.8: Illustration of CUDA's memory hierarchy

The device memory hierarchy is as follows:

- **Global memory** is the device's DRAM and the largest memory space. All data from the host has to be made available in global memory (one exception to this is the use of zero copy) if it is to be used in a device thread. The contents of global memory can be copied back and forth from host to device. Every thread on the device in any running kernel can read/write to global memory. The best case scenario for any kernel is that it only has to read any individual chunk of memory it needs from global memory once. Global memory on newer devices is by default cached in the L1 and L2 cache, however this can be changed.

- **Register memory** is the smallest but also the memory with the lowest latency a thread has access to. Each thread has their own part of the register memory, which only they can read/write to. When all registers are used the thread will write to local memory.

- **Shared memory** is the on chip memory of the GPU. This memory is significantly smaller but as it is on chip it has both higher bandwidth and lower latency compared to global memory. The same shared memory is accessible by all threads on the same Streaming Multiprocessor. The main purposes of shared memory is reducing redundant access to global memory and sharing values with other threads within the same block. The lifetime of shared memory is the same as the block which uses it.

- **Local memory**. If a thread uses up all its available registers it will use local memory. Local memory is in reality only cached global memory, which means that in the worst case scenario a thread will have to read from global memory leading to a significant loss of performance.

- **Constant memory** is global memory but cached on chip in the constant cache. This is read only memory. The cache algorithm is suited for reads which are close together or the same. On some devices L1 and constant cache are the same.
- **Texture memory** is global memory but cached on chip in the texture cache. The cache algorithm is suited for texture like objects where we often read addresses close in horizontal or vertical directions.

2.7.2 CUDA thread hierarchy

The CUDA thread hierarchy is spilt into three dimensions: threads, blocks and grids as seen in figure 2.9.

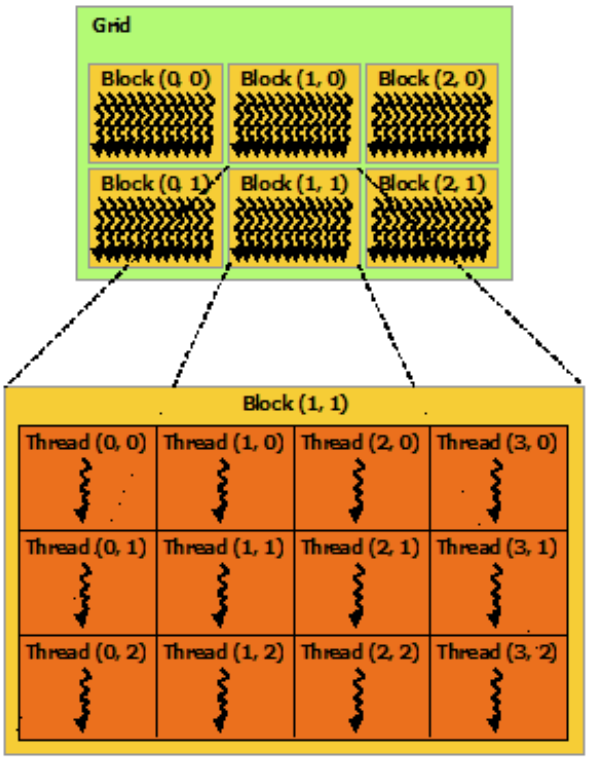

Figure 2.9: Illustration of CUDA's thread hierarchy, figure from [16]

- **Threads and warps**: CUDA threads can be thought of as normal threads in that they can (theoretically) work as normal threads with their own unique paths of execution and data. However, a CUDA thread is a part of a warp. Warps are groups of 32 (on current NVIDIA GPUs) threads which share the same instruction pointer and execute in lock step. This is referred to as single instruction multiple thread (SIMT). The warp or instruction pointer will have to go through every possible path any of the 32 threads take, which is also what every thread has to do where threads that do not belong to said path will

be idle. When threads within the same warp take different paths it is referred to as thread divergence, see section (2.7.4).

- **Block**: Blocks are groups of warps/threads. All warps/threads in the same block share the same shared memory. Also all threads in a block are scheduled on the same Streaming Multiprocessors.
- **Grid**: Grids are all the blocks which belong to the same kernel.

2.7.3 Synchronization

As we saw in section (2.7.2) CUDA's thread hierarchy is three-dimensional. We also have different levels of barriers/synchronizations.

- **Warp wide:** Synchronization is achieved with the `syncwarp` function or any warp-level primitive. What threads within the warp you want to synchronize can be defined by a mask which is given as input to `syncwarp`.
- **Block wide:** Synchronization is achieved with the `syncthreads` function. It is also possible to only synchronize a subgroup of the threads/warps in the block. This can be done with Cooperative Group.
- **Grid wide:** Synchronization can be achieved by the use of CUDA Cooperative Groups. If we start, and try to synchronize more blocks than we have SMs we may end up in a deadlock.
- **Host/Device:** the device and host can be synchronized with `cudadevicesynchronize`. Host and device are also synchronized at various CUDA calls such as `cudaMemCpy`.

2.7.4 Thread divergence

Thread divergence is an important thing to keep in mind while writing kernels. Thread divergence occurs when there is branching within a warp. Branching means one or more threads will take a different path of execution at for example an if statement. This leads to a drop in performance as the threads not in the current branch will be idle. To maximize efficiency of the SIMT architecture we want every thread in the same warp to stay in the same execution branch. However this is not always possible and you will often end up with some degree of thread divergence, nevertheless the compiler is very good at optimizing and simple divergence is some times dealt with. Thread divergence can be calculated by the number of different paths the threads within a warp take. The worst case scenario will be when every thread takes a different path, if we are working with 32 threads per warp, this will lead to a slowdown of 32X.

2.7.5 CUDA kernel

Kernel is the term we use for functions we run on the GPU. They are defined by adding `__global__` to a normal C++ function. When launching a kernel you can specify the number of blocks, threads and the size of shared memory. The blocks in the kernel will be scheduled over the different multiprocessors on the GPU.

2.7.6 Data transfer host - device

The bandwidth between device and host is very low (if they do not use the same DRAM), because of this one wants to minimize data transfer between device and host. Thus, at times it can be faster to do tasks which are not suited for the GPU on the GPU, rather than moving the data back to the host only to have to move it back to the device again. The host and device are asynchronous, this gives us some different approaches for copying/reading data. One approach is simply copying all the data we need from host to device while the GPU is idle. In CUDA you can do this with `cudamemcpy`, and in some cases this approach is necessary, but performance wise it means the GPU will be idle for however long it takes to copy the data. Another approach would be to do both copying and computing concurrently. One way of doing this would be starting a kernel with data we already have on the device, and then start copying data with `cudamemcpy`. This could give us some overlap, but how much overlap will vary. To maximize concurrency between copying data and the GPU working on data there are two things we need to be aware of: streams and pinned memory. A stream is a context which can call CUDA, therefore multiple streams give us the freedom to schedule multiple kernels, copies, etc. at the same time. Pinned memory in this context is unpageble host memory. In CUDA we can allocate pinned memory with `cudaHostMalloc`, this is necessary to make sure that `cudaMemcpyAsync` is actually asynchronous. `cudaMemcpyAsync` lets you specify the stream you want to perform the copy with. This gives the ability to copy data between device and host, while also running kernels on the GPU as illustrated in figure 2.10.

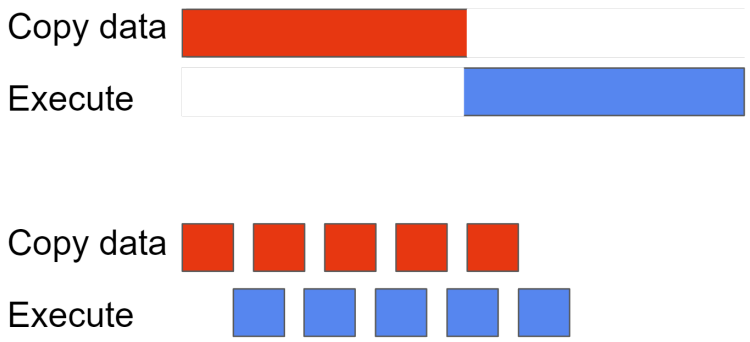

Figure 2.10: Illustration of Sequential and Concurrent execution

Zero copy (reading directly from host memory in a kernel as threads) is also possible, which lets us achieve concurrent execution as long as we have run enough threads to hide the latency of the reads from host. Unified Memory between device and host is also possible, but this only serves to improve the readability of the code, and as far as we know is does not contribute to better performance. However if your device and host use the same DRAM this will most likely lead to better performance and let us avoid having to store the same data in 2 different locations. Manged memory can be allocated with `cudaMallocManged`.

2.7.7 Data transfer global memory - warps/threads

Coalesced access to global memory is one of if not the most important thing to keep in mind when using CUDA. While the compiler will generally be able to optimize the reads/writes, there are some things which should always be done, if possible. The most efficient possible reads from global can be achieved with coalesced reads. What this means is that if threads in the same warp access sequential memory addresses, we can fetch the data with one memory transaction depending on the size which each thread wants to read. On newer devices this generally works as longs as the threads in the same warp access the same 32 or 128 byte memory bank, where one big read will be made including the bytes no one requested. Coalesced writes work in much the same way, where as long as the threads write to sequential memory addresses the writes will be done in fewer transactions.

2.7.8 Data transfer shared memory - warps/threads

For shared memory there is no performance loss if threads in the same warp do not access sequential memory addresses. As with all other memory, two threads writing to the same address at the same time leads to unspecified behavior. When reading from shared memory there is

one thing to be aware of namely bank conflicts. Shared memory is split into multiple banks (memory modules), and each bank can be accessed concurrently. This lets us read/write to and from multiple banks at the same time leading to a speed up equal to the number of banks we read/write to. However, if multiple threads want to access the same banks the requests are serialized. One exception to this however is when multiple threads in the same warp want to access the same banks, in which case each bank broadcast is coalesced into a multi cast.

From a software perspective there are two types of shared memory available, static and dynamic. To use static shared memory the size needs to be known at compile-time, while dynamic lets you decide at run-time.

2.7.9 Data transfer thread - thread with Warp-level Primitives

The fastest way to transfer data between two threads in CUDA is with the warp-level primitives. Warp level primitives work within the warp and lets us transfer data between threads in the same warp. To do this all the threads which want to partake in the exchange have to be synchronized. This is usually done by giving the primitive a mask which specify which threads will partake.

2.7.10 CUDA atomics

CUDA atomics are functions offered by CUDA which allow us to read, write, increment values, change values which exist in in a critical section without having to worry about race conditions. This works for both shared and global memory. However using atomics can be costly if there are many threads constantly trying to write to the same values, it should therefore be avoided when possible.

2.7.11 Double, single and half-precision

In CUDA there are three main levels of precision, double (64 bit), single (32 bit) and half (16 bit). CUDA is optimized for single (32bit) operations, however on newer GPU architectures NVIDIA introduced a 2 way Single instruction, multiple data (SIMD) for half precision [7]. This speeds up the use of half-precision to 2X compared to single if used correctly. This is because it gives us the ability to perform two 16 bit operations at the cost of one 32 bit operation. It also leads to better performance because of the reduced size leading to less data having to to be read. However this is only suitable if the precision of the values are not that important as there is a loss in accuracy.

Using half-precision correctly for max performance

To achieve two-way SIMD when using `half` (16-bit floats) in CUDA, both the `half`s have to be written to the same 32-bit register. The easiest way of doing this is by by using the `__half2` type which can be used by including

`cuda_fp16.h`. To do two-way SIMD with `__half2` we use the arithmetic and comparison functions from the CUDA math API [18].

2.7.12 Occupancy

Occupancy is a metric for how many warps we are running compared to the theoretical max number of warps we can run over all the multiprocessors. This is often a good metric to look at when optimizing. It shows to what degree we are actually utilizing the GPU, and if we are solving the problem in a way that does a good job dividing the work over all the multiprocessors. Furthermore, when a warp/thread executes some arithmetic instructions or reads from global memory, it will have to wait for the results before continuing. The only way to hide this latency is by scheduling more warps/threads. However, max or high occupancy does not necessarily correlate to good performance. Usually there is a threshold above which, increasing occupancy no longer affects performance. There are also cases where the problem is simply better suited for kernels that cannot achieve high occupancy, for example when one block needs all the shared memory.

Conditions which determine what occupancy we can reach

Conditions which determine how many blocks and threads we can run on a multiprocessor:

- Max number of threads the hardware allows
- Max number of blocks the hardware allows
- Shared memory available
- Registers available

A multiprocessor obviously has a finite amount of resources, so for any block to run on the multiprocessor it needs the required resources available before it can run. For example, if each block in the kernel uses 60% of the total shared memory on the multiprocessor it is running on, we would only be able to run one such block at a time, however if it only used 50% we could possibly run two.

Concurrent Streams

Streams lets us schedule multiple kernels at once, which gives us more options for occupancy. We no longer necessarily need to make sure the kernels get good occupancy alone, but we can now write kernels which utilize the GPU well when run concurrently. For example if each block in the first kernel uses 60% of the total shared memory on the multiprocessor it is running on, and the second uses 40%.

2.8 The cuBLAS library

cuBLAS is CUDAs BLAS (Basic Linear Algebra Subroutine) implementation. BLAS is a specification on how to perform linear algebra operations. This cuBLAS introduction is based on NVIDIAS documentation "cuBLAS Library - User Guide" [15]. BLAS has three levels where each level deals with its own type of problems:

- **Level 1** deals with vector and vector-vector operations. Some examples are the dot product of two vectors, and the Euclidean norm of a vector. In cuBLAS they are respectively referred to as dot and nrm2.
- **Level 2** deals with matrix and matrix-vector operations. One such operation would be the vector - matrix multiplication. vector - matrix multiplication is in cuBLAS and in most BLAS libraries referred to as GEMV (General matrix-vector multiply).
- **Level 3** deals with matrix - matrix operations. One such operations is the matrix - matrix multiplication. This is referred to as GEMM (General matrix-matrix multiply) and is one of the more widely used BLAS operations (especially with the rise of machine learning). This it true to the extent to where NVIDIA introduced a new type of core optimized for this problem called the tensor core.

cuBLAS is highly optimized, fairly easy to use and is supported on most NVIDIA GPU architectures. Therefore if we need to do some BLAS operation we are in most cases better of using cuBLAS versus trying to implement our own kernels.

2.8.1 Using cuBLAS

The cuBLAS library is called from the host, and as with other CUDA kernels it can be called from different streams. A cuBLAS call from host is not asynchronous, meaning the CPU will not continue its execution before the call is finished. However, it can be asynchronous if you specify a stream. To use cuBLAS the first thing you need to do is initialize a cuBLAS context, referred to as a handle. Handles can and should be reused as there is an overhead when initializing one. One thing to note however, is that while you need different handles for different CPU threads, different streams can share the same handle. Probably the most notable thing when using cuBLAS is that it operates in column-major while C and C++ works in row-major. This leads to some complexity. A row-major matrix in column-major will be read as a transpose of a row-major matrix (and vice versa for a column-major matrix in row-major). By understanding this, we can deal with the problem either by doing the opposite in terms of transposing, or by changing the order of the matrices. When we are only dealing with vectors reading in column-major vs row-major does not make a difference. When using cuBLAS the input for the calls has to be in either the device memory or in unpageble host memory allocated with `cudaMallocHost`.

2.8.2 cuBLAS GEMM example

Here we go through an example of how we can perform the dot product with cuBLAS GEMM on two row-major matrices. GEMM is defined as $C = \alpha op(A)op(B) + \beta C$ here op is a functions which either transposes, conjugates or does nothing to the matrix. C is the resulting matrix, while A and B are the input matrices. α and β are two scalars giving us the ability to scale the matrices. If all we want to do is compute the multiply of the matrices we set $\alpha = 1$ and $\beta = 0$. The GEMM function for `floats` in cuBLAS is called `cublasSgemm` and takes the following input:

```
cublasSgemm(cublasHandle_t   handle,
                    cublasOperation_t   transa,
                    cublasOperation_t   transb,
                    int   m,   int   n,   int   k,
                    const   float   *alpha,
                    const   float   *A,   int   lda,
                    const   float   *B,   int   ldb,
                    const   float   *beta,
                    float   *C,   int   ldc)
```

A matrix in CUDA and C/C++ in general will be stored in a continuous array. Given a matrix with 3 vectors of length 3 it could be stored as an array of length 9 where we can write it as $A[9] = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9\}$. If we were to write this array as a matrix it would look like this:

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{pmatrix}$$

In C++ we work in row-major, meaning that if we were to read the first vector we would read in the horizontal direction i.e $A[0], A[1], A[2]$ which would be $\{a_1, a_2, a_3\}$. cuBLAS on the other hand works in column-major, meaning it would read in the vertical direction giving us $A[0], A[3], A[6]$ which is $\{a_1, a_4, a_7\}$. Assuming we also have a matrix B of the dimensions 4×3 4 rows 3 columns stored as a 12 element long array $B[12]$. As a row-major matrix we could write it as:

$$\begin{pmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \\ b_{10} & b_{11} & b_{12} \end{pmatrix}$$

For our example we want every row-major vector in A to be dotted with every row-major vector in B. This would give us the 3×4 matrix C. $C[0 \rightarrow 3]$ would be the dot product between the first row vector in A and every row vector in B, $C[4 \rightarrow 7]$ would be would be the dot product between the second row vector in A and every row vector in B, and $C[8 \rightarrow 11]$ would be the dot product between the third row vector in A and all the row vectors in B. Assuming that cuBLAS GEMM worked in row-major i.e. multiplies every row vector in the first matrix with every column vector in

the second one, then we would want to transpose the second matrix giving us $A * B^T = C$. However as cuBLAS works in column-major it would read this as $A^T * B$ which is every column vector multiplied with every column vector which would not be the C matrix we want. To fix this we could rewrite our function to $A^T * B$ which cuBLAS would then read as $A * B^T$, however while this would give us the correct values it would give us the output in column-major. To circumvent this we change the order of the matrices giving us $B^T * A$, cuBLAS would read this as $B * A^T = C^T$. C^T will when read in row-major be C which is the 4×3 matrix we want. In cuBLAS we would write this as:

```
   float alpha = 1.0f;
float beta = 0.0f;
int m = 4;
int n = 3;
int k = 3;
int ldA = 3;
int ldB = 4;
int ldC = 3;
cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N,
          m, n, k, &alpha, B, ldB, A, ldA, &b, A, ldC);
```

2.8.3 Best performance for GEMM when using tensor cores

When using cuBLAS GEMM we want to utilize our GPUs tensor cores if it has any. To get the best performance possible when using tensor cores NVIDIA provides a few conditions which must be met:

1. `m % 8 == 0`
2. `k % 8 == 0`
3. `op_B == CUBLAS_OP_N || n % 8 == 0`
4. `intptr_t(A) % 16 == 0`
5. `intptr_t(B) % 16 == 0`
6. `intptr_t(C) % 16 == 0`
7. `intptr_t(A+lda) % 16 == 0`
8. `intptr_t(B+ldb) % 16 == 0`
9. `intptr_t(C+ldc) % 16 == 0`

As long as we use a CUDA function to allocate our arrays A, B and C we will not have to worry about condition 4, 5 or 6. This is because of how CUDA allocates memory. The rest of the conditions however are things we should keep in mind, and when plausible for example use padding to make sure that they are met.

2.9 The Thrust library

Thrust is a CUDA library designed for productivity. This short introduction to Thrust is based on NVIDIAs documentation "Thrust Quick Start Guide" [19]. It is easy to use in the sense that the user does not need to have any understanding of the underlying architecture of the GPU or CUDA. Blocks, warps, threads are all decided by Thrust removing much of the complexity which comes with programming in CUDA. It offers much of what C++ STL offers. Some examples are sort, reduce and the ability to use vectors in much the same way we use them in C++. We can also change these algorithms for our own custom values and classes in much the same way we would in C++. Thrust is called from the host, and we can use both device and host pointers as long as we specify it in the thrust calls. Thrust vectors let us use dynamic allocation, and while this comes with a loss in performance it also significantly simplifies the implementation of many algorithms.

Chapter 3

Design

In this chapter we go over the design of the brute-force SIFT feature point matching algorithm which uses the dot product and cuBLAS (section 3.1), and the design for LSH on GPU (section 3.2). We also talk about using `halfs` instead of `floats` to represent the values in the SIFT vectors (section 3.3). The designs we present act as a baseline for how we plan to implement the algorithms, and reflections about what problems we may face.

3.1 Design of the brute-force SIFT feature point matching algorithm which uses the dot product and cuBLAS

A kNN cuBLAS implementation is introduced in the paper "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching"[5] which we introduced in section 2.2. In our design and implementation we have used this paper as inspiration, however as we do not need to solve the kNN problem but the 2NN problem, and our input is SIFT vectors, our solution is much simpler. For the implementation see section (4.1)

3.1.1 kNN with cuBLAS

Our goal is an algorithm that with the help of cuBLAS calculates the 2 nearest neighbors (2NN), and checks if the 1 nearest neighbor is a valid SIFT match. For distance we use Euclidean. If we assume that X and Y are 2 vectors in a D-dimensional Euclidean space, e.g. $X = (x_1, x_2, ..., x_D)$ and $Y = (y_1, y_2, ..., y_D)$. Then we know that to calculate the distance between X and Y we can use the formula.

$$d(X, Y) = \sqrt{(x_1 - y_2)^2 + (x_2 - y_2)^2 + ... + (x_D - y_D)^2}$$

As we only want to find the 2NN and are not really interested in the exact distance, we can drop the square root as it does not change the 2NN. This gives us $d(X, Y)^2 = (x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_D - y_D)^2$. Moreover, as we want to use cuBLAS as much as possible we want to rewrite our

equation to be more suited for matrix operations. We can do this by seeing that [5]:

$$d(X,Y)^2 = (X-Y)^2(X-Y) = ||X||^2 + ||Y||^2 - 2X^TY$$

Here $||\cdot||$ represents the Euclidean norm, and X^T is the transpose of X. To rewrite for sets of vectors, we think of Q and R as two sets of vectors, e.g. $Q = \{v_1, .., v_i\}$ and $R = \{w_1, .., w_j\}$. $v_1, ..., v_i$ and $w_1, .., w_j$ are vectors in a D-dimensional Euclidean space. Dist is the vector of squared distances between any vector in Q to any vector in R. We can then write Dist as:

$$Dist = ||Q||^2 + ||R||^2 - 2Q^TR$$

3.1.2 cuBLAS for SIFT vector matching

Here an important thing to note is that we are dealing with SIFT vectors, not arbitrary data. SIFT vectors are normalized, in our case to 1. This means that we already know both $||Q||^2$ and $||R||^2$ as this it the distance from the zero vector to the vector. For matrices of size $D \times i$ like Q, the norm $||\cdot||$ is defined as $||Q|| = Q^TQ$, which is an i-dimensional vector. The definition means that each row v_n in Q^T is multiplied with the same column v_n in Q, which is the same as $v_n \cdot v_n = ||v_n||^2$, which we know is 1. We can therefore reduce our problem even more giving us the equation:

$$Dist = 2 - 2Q^TR$$

$-2Q^TR$ is a matrix multiplication and can be done extremely well with one call to the cuBLAS library. We only need the 2NN, meaning we can simplify what we do into 2 steps:

1. Calculate $-2Q^TR$
2. Find the two smallest values for each query vector

While step 1 is perfect for cuBLAS, step 2 is as far as we know best suited to a self written CUDA kernel. Thrust could also be used, but as this is a fairly straightforward reduction problem we believe a self written kernel would be best.

3.2 Locality-sensitive hashing for SIFT on GPU Design

LSH has many versions but the shell or outline will for the most part remain the same. To test as many different versions of LSH as possible with minimal coding, we will try to design our algorithm on the GPU in a way that allows us to change different parts without too much work. Our goal is to implement LSH in a way which runs efficiently on a GPU. Therefore the first step would be to see how and where we can parallelize the algorithm. Basic LSH for SIFT can have 3 main steps:

1. Make hash values for query and reference SIFT vectors by hashing

2. Order/index the data by the hash value

3. Do a short list 2NN search where we compare every query point with every reference point which has the same hash value

These 3 main steps will also most likely have to be done multiple times independently to achieve the desired accuracy. Doing independent runs of step 1 to 3 in parallel will be the out-most point where we can achieve parallelism. To do this we can use C++ threads and concurrent streams. Also each of the 3 steps can be parallelized.

3.2.1 Hash function and hashing in parallel on GPU

Using the dot product as the hash function seems the most reasonable, this is mostly due to the fact that it means we can use cuBLAS. This somewhat simplifies the process, and it also guarantees good performance. cuBLAS lets us perform multiple dot products in parallel for all the SIFT points using a matrix - matrix multiplication.

Using the dot product results to make hash values

To make our hash value we first need to decide what our hash values should be stored as. As we want to be efficient using a 32 bit integer as hash value seems the best choice. This means we will have up to 2^{32} buckets we can divide the data in. Notably using all 2^{32} buckets will most likely not be necessary. There are many ways of setting the 32 bits with the dot products, but the most basic one will be seeing if the value is over or under 0.

Here is a quick example of how we create a hash value for one SIFT vector using cuBLAS and 1 - 32 random vectors:

1. First we dot the SIFT feature point with 1 - 32 random vectors (when we take the dot product we are projecting the SIFT feature point onto a 1 dimensional line i.e to some number)

2. We look at the value from each of the 1 - 32 dot products and assign a bit according to if the value is over or under 0

Some iteration of this could be using more than one bit for each dot product. For example we could use 2 bits meaning we would want every dot product to be mapped to the range 0 to 3 or some range where we get a good split over 4 values.

Random vectors

Every value in a SIFT vector is between 0 and 0.5 also the norm of the vector will be 1. Meaning the data is spread over a hyperplane in distance 1 from the zero vector. To split SIFT vectors with random vectors, if we were to split on 0 as described above, we would have to make sure that some of the values in the random vectors are negative. Preferably every value has a 50% chance of being either negative or positive. We could for example put every value to 1 or -1.

Kernels for Hashing

We will need at least 2 kernels to create hash values for each query and reference SIFT vector. The cuBLAS call, and a self written CUDA kernel. The CUDA kernel will read in the dot products and set the bits for each SIFT vectors hash value according to the hash scheme and dot product value.

3.2.2 Order/sort by hash value

When every query point and reference point has a hash value we have to somehow sort or link the points in a way which we can easily and efficiently perform a 2NN search. The two main approaches for this we came up with were:

- Make an index array and sort the indexes by hash value
- Create an empty array, equal in length to the range of the hash values, insert into said array by using the hash value as index and using a linked list when collision occurs.

Making an index array and sorting it by the hash values seems the most plausible. The biggest problem with the empty array linked list approach is that it means we will have to limit the number of buckets we can use as an array of size 2^{32} alone will use more memory than we have available on the GPU. There could be better solutions for doing this (hashing schemes etc.), even so this will be complicated and hard to optimize for memory accesses.

Sorting index array by hash value

This can be done with the Thrust library. Implementing our own sorting algorithm seems very complicated, and will probably not be worth the effort. We will have to implement a sorting class to use with thrust.

3.2.3 2NN short list search

While we wanted to use the same approach as described in 3.1 this will most likely not be a good approach as our points will be spread in memory, meaning using cuBLAS will require moving data. As this is supposed to be a short list search, simply using shared memory to reduce redundant reads will probably be faster, which is why this is what we will try. However as it will be very hard to implement a short list search kernel which gets high occupancy (because we do not know the sizes of the different lists we have to match), we will use concurrent streams. This means using an CUDA atomic will most likely be necessary as many warps may try to write to the same address when setting the index and value for the 2NNs.

3.2.4 Using C++ threads and concurrent streams

One of the bigger problems we will probably face when trying to implement LSH on GPU is low occupancy. The problem arises from how unpredictable the algorithm quickly becomes. The main problem being the short list search. The problem is simply that we have no way of knowing how "short" each list will be, and as we will schedule each short list search as a block, and CUDA will not start the next kernel in the same stream as long as all blocks are not done. This means we may end up with one block blocking the next steps only utilizing a small part of the GPU. To counter this we can use concurrent streams and maybe C++ threads. Also using the CPU for certain parts of the algorithm, may be necessary, in which case streams are necessary to hide the overhead of our GPU waiting for the CPU.

3.3 SIFT vectors in CUDA as `halfs`

As mentioned in section 2.1.2 we know that we can represent SIFT values as `unsigned chars`, however as CUDA is not optimized for single byte operations this would not be the best choice. Half-precision or 2 byte `floats` on the other hand is something which CUDA is optimized for. As we already know we can reduce our `float` SIFT vectors to `unsigned char` vectors, reducing to `halfs` should also work. This should be more accurate than using `unsigned chars`. The motivation for using half-precision in CUDA is that we can get a guaranteed speed up of 2X. This is because as long as 2 `halfs` are in the same 32-bit structure, we can perform the same action on both for the cost of performing it on one 32-bit `float`. On top of that we will be able to reduce the transfer time from global GPU memory and the threads as we do not need to read as many bytes as before.

Chapter 4

Implementation

In this chapter we present the implementations of brute-force SIFT vector matching using cuBLAS (see section 4.1), LSH on GPU (see section 4.2), and a naive CUDA brute-force (see section 4.4).

4.1 Implementation of brute-force SIFT vector matching using cuBLAS

Overview of how we implemented the 2 nearest neighbour brute-force for SIFT vectors using cuBLAS and `halfs`. For design see section 3.1. For results and reflections see 5.4.

4.1.1 Assumptions

We assume that the query SIFT vectors `Q`, references SIFT vectors `R` and the output array `matches` where we write the results all fit in device memory at the same time. We also assume that the SIFT vectors are normalized as this is how we define the problem in the design part.

4.1.2 `half` vs `float`

As mentioned in section 3.3 we know that using `halfs` gives the best performance in terms of speed when using CUDA. However if the loss of precision is justifiable or not, we can only known through testing. When testing on real data (see 5.4.1) we see that there is not much of a difference in the recall between using `floats` vs using `halfs`. However, there is much to be gained in terms of run time (see 5.4.6). Based on this. the version we really focus on in this implementation is the version using `halfs`. We did also implement a version using `floats`, however we did not optimize to the same degree.

4.1.3 Input

The input is 2 arrays, `Q` and `R`, which are arrays of SIFT vectors. The SIFT vectors can be of type `float` or `half`, and in device or host memory

(however if in host memory we assume that the input is of type `float`). For best performance one would use `halfs` which are already in device memory. We also take in an array of `unsigned ints`. This is where we write the matches if there are any and $UINT32\_MAX$ if there are none, i.e the distance ratio between the 2NNs for the query point, do not satisfy the threshold. The matches array needs to be on device or allocated with `cudaMallocManged`. For the rest of the input we also need a cuBLAS handle, the type of the input arrays, the number of warps used by the reduction kernel, number of streams to use and the size the array we write our matrix-matrix multiplication results to.

4.1.4 Moving data to device, and converting to `halfs`

If the data is not already in device memory, we have to transfer it from host to device, and as we assume that data not in device memory, is of type `float` we will also have to convert to `half`. This will be different for the query vectors in Q and the reference vectors in R. For the reference vectors we need to transfer and convert the whole array at once, this is because of how we use batching in the algorithm (see 4.1.7). This will be done by using zero copy, meaning we read directly from host in the kernel which converts to `halfs`. This both lets us use less memory as we do not need to have an intermediate array, and it is what we found to give the best performance (see 5.3 for tests). On the other hand the query vectors do not need to be moved all at once, therefore we have 2 options:

1. Transfer and convert only the vectors we need for the current batch
2. Transfer and convert the whole array at once in the same way we did the reference array

When testing we found that the first option worked best.

4.1.5 Algorithm overview

As mentioned in the design we use two main kernels for this algorithm. One is the cuBLAS GEMM call and the other one is the reduction kernel. For the reduction kernel implementation see 4.1.10, for how we use cuBLAS GEMM see 4.1.9. We do sometimes have to transfer data and convert to `halfs`, this is also done with a very simple kernel see 4.3. Because of how memory intensive the algorithm is, and how cuBLAS works we need to use batching. For how we implemented batching see 4.1.7. Using batching means that on top of letting us process bigger queries, it also gives us the chance to use concurrent streams to reduce overhead, for how we do this see 4.1.8.

The main algorithm will look like this where every bathed run executes this algorithm on some subset of Q and all of R:

1. Dot every SIFT vector in Q with every SIFT vector in R (using cuBLAS), write the results to the `half` array we call Dist

2. Call the reduction kernel on Dist with number of blocks equal to the number of Q vectors in the batched run.

3. The reduction kernel will when done fill up the `unsigned int` array called matches with the index of the nearest neighbour in R for the query vector point at the same index in the matches and Q array.

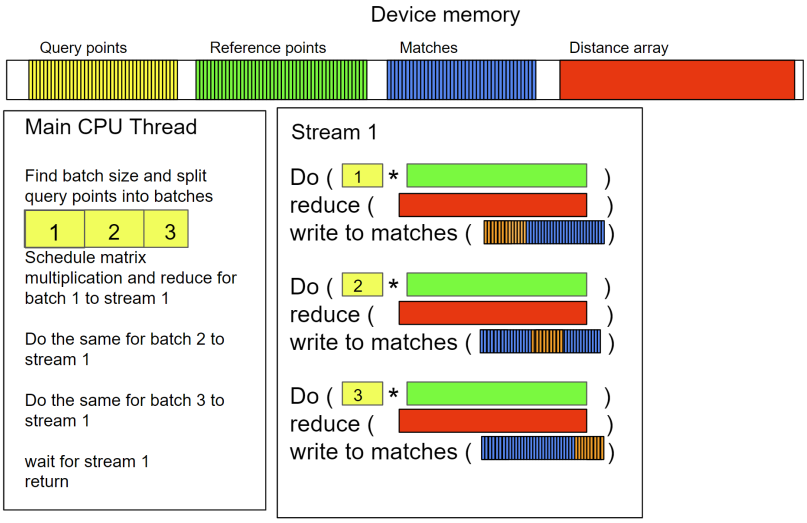

Figure 4.1: Simplified illustration of how the algorithm will work with one stream

4.1.6 Memory usage

This algorithm is extremely memory heavy, due to the fact that we need to allocate the array where we write the output of the cuBLAS GEMM call. The size of this array in bytes if we did not batch would be: (number of query points) * (number of reference points) * 2 (this is an array of `halfs` where the size of a `half` is 2 bytes). The GPU we test on only has 12 GB of memory, meaning that we would not be able to test on higher numbers than: Reference SIFT vectors = $100k$ and Query SIFT vectors $= 50k$ where $100k * 50k * 2byte = 10GB$. Therefore we need to batch our algorithm to test on reasonably large sets of SIFT vectors. Another motivation for batching is that cuBLAS seems to be more efficient on smaller queries.

4.1.7 Batching

We batch by seeing how many query points we can dot with the whole reference point array at a time, while still fitting the whole output array into the size we want to use for our cuBLAS GEMM results array. The size of the cuBLAS GEMM results array is changeable.

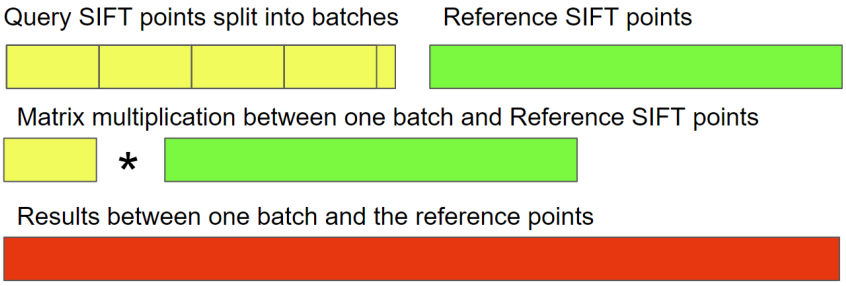

Figure 4.2: Simplified illustration of how we use batching for our cuBLAS SIFT feature matching brute force algorithm

For how changing the size of this array effects the performance see 5.4.2. For our GPU (RTX 3060) on the ANN_SIFT1M cuBLAS is the most effective when we restrict one stream to 0.766 GB of the total GPU memory space, meaning we would dot 383 query points at a time. We are not quite sure why this is the case, but it is probable that it has something to do with how cuBLAS handles shared memory when called on very big matrices. However we were not able to implement a formula which atomically let's us choose the best possible size for the results array.

4.1.8 Concurrent streams and occupancy for cuBLAS 2NN

As mentioned in the algorithm overview (4.1.5), we use batching because of how memory heavy our algorithm is, and because cuBLAS is better at smaller queries. This also means we have the option of using concurrent streams, which could give us better performance by increasing occupancy. That being said, cuBLAS calls, (especially GEMM calls on big matrices) will almost always occupy the whole GPU, meaning running other kernels concurrently will most likely not be very helpful. On the other hand, both the conversion kernel (used when converting to half) and the reduction kernel can probably benefit from concurrent streams, to some degree.

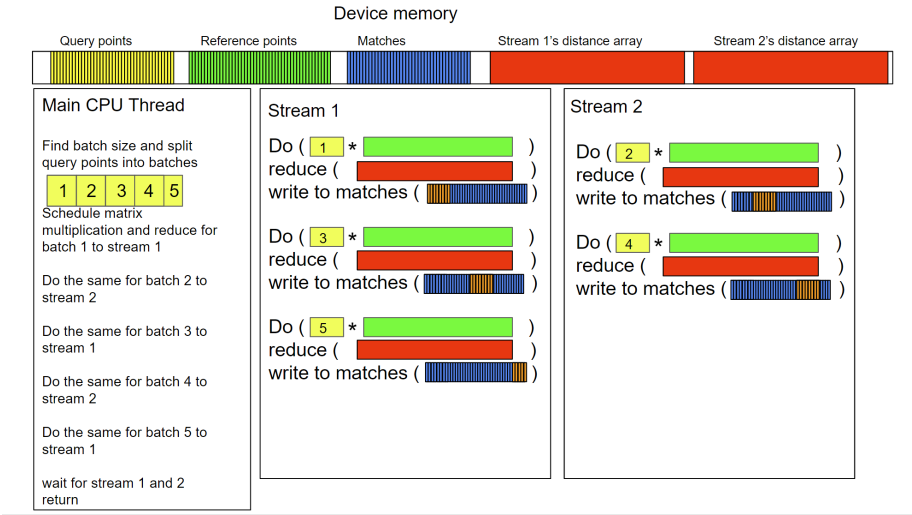

Figure 4.3: Simplified illustration of how our algorithm will run with 2 concurrent streams

When testing, we found that we only benefit from using concurrent streams when the number of query points is very high. See 5.4.5 for the tests and reflections.

4.1.9 Matrix matrix multiplication for cuBLAS 2NN

As mentioned in the design part, we need to perform the matrix-matrix multiplication $-2Q^TR$. This is done with cuBLAS. We use the cuBLAS call `cublasHgemm` which is the GEMM call for `halfs` in cuBLAS. This function takes 3 matrices and some scalar values as input. The 3 matrices will be the two input arrays of SIFT vectors Q, R and an output array we call dist. The dist can be thought of as a 2 dimensional array in the form $dist[Q\_size][R\_size]$. We also feed in two scalars: alpha and beta. By setting alpha to -2 we no longer need to negate or multiply the results ourselves. We set beta to 0. To negate the fact that cuBLAS works in column-major while C++ (and our data) is written in row-major, we do $-2R^TQ$ instead of $-2Q^TR$ (see section 2.8.2 for a detailed explanation on why this works). This call will look like this:

```
cublasHgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, r_n, q_n,
128, &a, (half *)R, 128, (half *)Q, 128, &b, (half *)dist, r_n);
```

Here q_n is the number of query points and r_n is the number of reference points.

Using tensor cores

To achieve the best possible performance when using cuBLAS with tensor cores we have to remember the conditions which must be met (see section 2.8.3). Conditions 4 - 6 are already met as we allocate the arrays with a CUDA function. 7, 8, 2 and 3 are also already met as lda, ldb and k are equal to 128 and `128 % 16 == 0`. This leaves 1, 3 and 9. However, as we

use the `CUBLAS_OP_N` for `op_B`, condition 3 is also met no matter the input size. This leaves conditions 1 and 9, which in our case are more or less the same condition, as they are both dependent on the number of SIFT points in the reference array R.

- Condition 1: `m % 8 == 0`
- Condition 9: `intptr_t(C+ldc) % 16 == 0`

Here both *m* and *ldc* are equal to the number of SIFT points in R, therefore the solution becomes fairly simple. As we already know that `intptr_t(C) % 16 == 0` (condition 6), we only need to make sure that `ldc % 16 == 0`. In doing so we inherently also solve `m % 8 == 0` as ldc == m, and if `ldc % 16 == 0` is true, then `ldc % 8 == 0` must also be true. We do this by padding the reference point array so that `size % 16 == 0`.

Padding

If the number of SIFT points in the reference array does not already fulfill `number % 16 == 0`, then we have to pad the array with 1 to 15 (depending on `size % 16 == 0`) new SIFT points, where each value is set to 0. If our input is of type `float`, all we have to do is allocate and write an extra 1 to 15 SIFT vectors when we convert from `floats` to `halfs`. However when the input is of type `half`, we have to allocate space in device memory, set the extra vectors to 0, then copy the data from the old array to the new. This obviously takes time, but nevertheless it does improve overall performance significantly despite the overhead as shown in 5.4.3.

4.1.10 Reduction kernel for cuBLAS 2NN brute force

This is a reduction on multiple *n* element long arrays, where we want to find the 2 smallest values in each array and keep the index of the smallest value. The size of *n* will be the same as the number of SIFT vectors in *R*, and the number of such arrays will be the same as the size of the subset of *Q* we are working on. The kernel we wrote solves this problem using one block per sub-array.

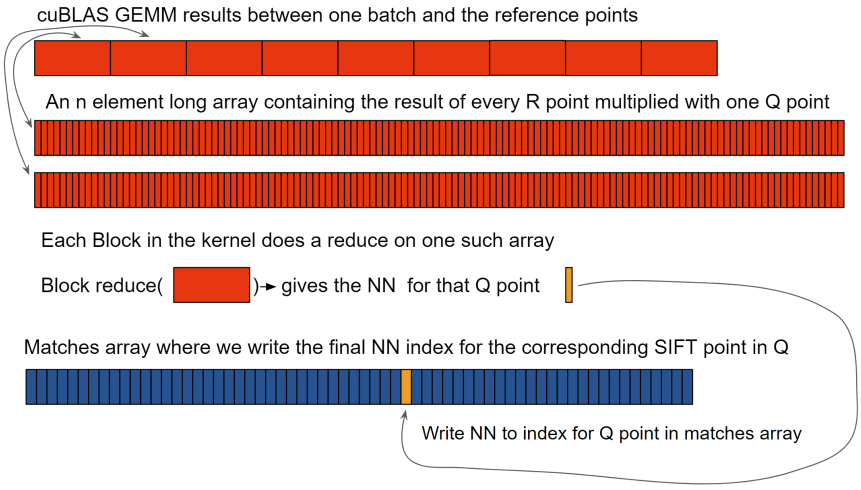

Figure 4.4: Simplified illustration of how we use one block per sub-array for our cuBLAS SIFT feature matching reduction kernel

The reduction

The number of warps per block is changeable (for how this changes performance on the ANN_SIFT1M data set, see 5.4.4). Each thread in the block reads 8 `halfs` at a time in a for loop, which it scans for the 2 smallest elements while keeping the index of the smallest element. The reads in the for loop are done in such a way that they will always be done coalesced. After scanning the whole array we do a warp wide reduction with CUDAs `__shfl_down`. Now the first thread of each warp has the 2 smallest values and the index of the smallest. Then we do a block wide reduction by using shared memory. We initialize 2 shared memory arrays, one of 32 `ints`, and one of 32 `half2s`. Now the thread of each warp with $id = 0$ writes both the index and 2 smallest values to shared memory. Each thread in the first warp reads the values in the shared memory arrays corresponding to their thread id within the warp. The we perform another warp reduction in the first warp only. The first thread in the first warp now has the 2 smallest values and the index of the smallest value.

Optional SIFT threshold step

The first thread can now check if the values are sufficiently different to be considered matches. To do this, we first need to get the true distance. This is done by adding 2 and taking the square root. Our original equation was $dist^2 = (2 - 2Q^TR)$ and we have already done $-2Q^TR$. If they are sufficiently different we write the index of the smallest element into the matches array, if not we write the max value of an unsigned int signaling that there were no suitable matches.

4.2 LSH for SIFT on GPU implementation

Overview of how we implemented the LSH algorithm for GPU. For design see Section 3.2.

4.2.1 Assumptions

We assume that the input problem is of a size where we do not need to worry about a out of core scenario i.e we will have enough memory on both device and host for what ever input we get.

4.2.2 `half` vs `float`

We started out by trying to implement a version using `floats`, but found that the overall performance could be improved substantially by using `halfs`, which we ended up doing.

4.2.3 Input

The input is 2 arrays (Q and R) which are the query SIFT points and the reference SIFT points. We also take a matches array where we write the index of the match we find for each query point. The rest of the input arguments are: a cuBLAS handle, input type of the SIFT points, number of iteration to run the algorithm, number of bits to use per bucket and the number of streams to use.

4.2.4 Moving data to device, and converting to `halfs`

If the input arrays are not already in device memory, we have to transfer them from host to device. We assume that if the data is not in device memory it is of type `float`. We will therefore also have to convert to `half`. This will be done in the same way for both the query vectors in Q, and the reference vectors in R. We need to transfer and convert the whole arrays before we can start our LSH algorithm. We do this by using zero copy, meaning we read directly from host in the kernel which converts to `halfs`. This both lets us use less memory as we do not need to have an intermediate array, and it is what we found to give the best performance (see 5.3) for tests. This will be done with streams where we use one stream for each of the arrays.

If the arrays are in device memory but of type `float` we allocate a new `half` array and call the kernel which converts the data.

4.2.5 Memory usage

How much memory we use will depend on the number of streams and the number of query and reference SIFT points. Our implementation is not that memory heavy in general, but for very big input arrays memory will be a problem. However, it was not a problem for the data set we did most of the testing on. The main array used to keep track of the distance to

the 2NN and the index of the NN is the min2_index array. This array will be accessed by multiple kernels in different streams simultaneously, which is why each min2_index element is a 64 bit value usable with the CUDA atomic compare and swap.

4.2.6 LSH on GPU algorithm overview

For our implementation we use C++ threads and streams. One iteration of the algorithm implementation will have these steps:

1. Main thread will make random vectors, schedule the dot product between the random vectors and query/reference points, this will be done in 2 streams.
2. Main thread will then start a new thread which will be in charge of this iteration, the main thread will pass the 2 streams it used to schedule the dot product calls onto this thread. We call this thread the iteration thread.
3. The iteration thread will then start 2 new threads giving them one stream each.
4. Each of the 2 threads will then be in charge of either the reference points or the query points for this iteration. We call theses threads the query thread and the reference thread.
5. The query and reference thread will schedule the kernel which sets the bit values (see 4.2.9), then sort the index array by hash value with thrust (see 4.2.10).
6. The query/reference threads will then make buckets with the index array and hash value array (see 4.2.11), then they will terminate.
7. The iteration thread will then match the buckets made by the query/reference threads (see 4.2.12).
8. The iteration thread will then schedule a brute-force 2NN short list search (see 4.2.13) for each pair of buckets it found in both the query buckets and reference buckets.
9. After this the iteration thread will synchronize to the stream it scheduled the brute-force on and then terminate.

The movement of the different arrays between device and host will be done asynchronously. The above mentioned steps will be repeated as many times as needed to achieve the desired accuracy. We will also run multiple of these iterations simultaneously, in order to hide the latency of using the CPU, and to maintain high occupancy. The reason we schedule the dot product in the main thread, is that we had problems when we scheduled them in different C++ threads with the same cuBLAS context handle.

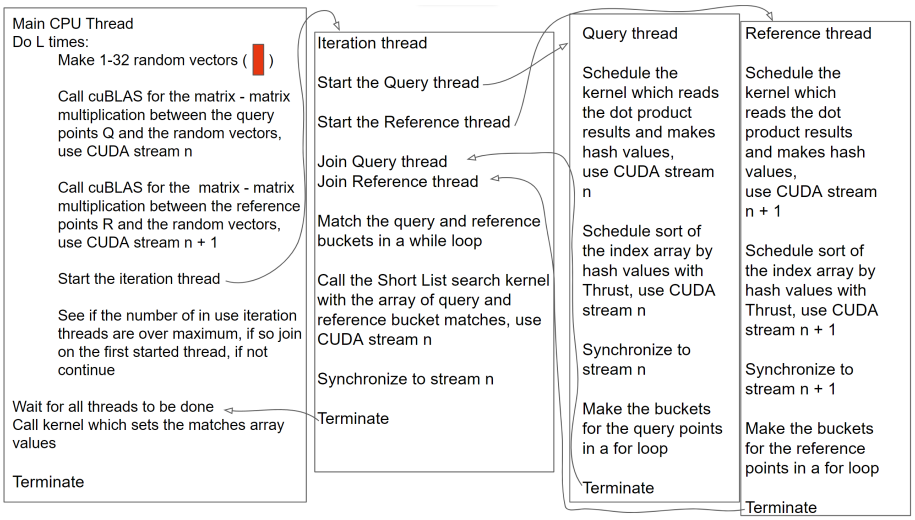

Figure 4.5: Illustration of how the different CPU threads work in conjunction

4.2.7 Random vectors

We make the random vectors on the CPU, and then transfer them to GPU with `cudaAsyncMemcpy`. This may not be the best solution, but for testing this was very helpful. We did try a few different types of random vectors, but we found that simply setting every value in the random vector to either 1 or -1 worked best. This splits the dot product values between negative and positive meaning we can split on 0.

4.2.8 Dot product using cuBLAS

To make the hash value for each query point and reference point, we need to dot every query and reference point with every random vector. This can be done with cuBLAS. cuBLAS is extremely efficient and we have therefore chosen to use it instead of making our own dot product kernel. To perform the dot product we use `cublasHgemm`, which is the matrix-matrix multiplication functions for `halfs` in cuBLAS. We want to perform the dot product, i.e given 2 vectors a and b we want ab^T, we want to do this pairwise between all points in the Q and R matrices and each vector in our random vector array. We have explained how to do this in section 2.8.2. Our input array will be the SIFT points (either the reference points or query points) and the random vectors. The output will be an array of arrays where each sub array contains 1 to 32 (depending on how many bits we use for the hash value) dot product values for each query/reference point.

Padding

We will pad the random vectors to meet the conditions mentioned in 2.8.3, as just as with the brute-force with cuBLAS, padding seems to be beneficial

as seen in 5.5.4.

4.2.9 Setting the bits in the hash values

The dot products using cuBLAS will leave us with 1 to 32 long arrays of `half` values for each SIFT point. Theses values are what we will use to form the hash values. There are a many different ways we can use these values to form the hash value, however how we read and write the data will be the same.

The kernel

For setting the bits of the hash values we wrote a CUDA kernel. This kernel reads each dot product value once, the reads are coalesced, and each thread will write one value to the output array, meaning we get coalesced writes as well. We use dynamic shared memory where the size will be equal to the number of dot products the block will have to read in. The kernel will have 2 main steps:

- Read in the values from the dot product array to shared memory. Here each warp will have its own part of shared memory which it will fill up. This is done with coalesced reads.
- Every thread in the warp will use the values from shared memory to set the bits of a 32 bit unsigned integer, after which the thread will write this value to the hash value array for the SIFT point it is in charge of, the writes will be coalesced. We also set the index in the index array for said SIFT point to use it when we sort.

We can change the number of warps, but we have to keep in mind that if we set this number too high we will end up using all of the shared memory. We found that 4 warps per block gave the best results.

Using dot product to make a hash value

After we have read in the values, each thread reads 1 to 32 values from shared memory and sets the bit of a 32-bit integer according to the values and the scheme we use for the hash values. We only implemented 2 such schemes:

- If the dot product value is above or below 0, set the bit in the hash corresponding to the same index as the dot product to 0 or 1.
- If the dot product value is in the range of 1-4, set the 2 bits in the hash corresponding to the same index as the dot products to 0 - 3.

We ended up using the first scheme, mostly due to the fact that it lets us be more precise, which made it easier to choose the number of bits to use.

4.2.10 Sort index array by hash value

We have 2 arrays, the array with the hash values, and the array with the index. The index array will return the same value as the value we access it with before we sort, e.g $index[2] = 2$. We sort with Thrust, utilizing a custom operator we made that lets us sort the index array by the values in the hash value array. This sorts the index array so that using the index array to access the hash value array will give us the hash values in ascending order, e.g $hash[index[0 -> n]] = [hash\_min, ..., hash\_max]$.

4.2.11 Making Query/Reference buckets

For both the query points and reference points we have the following:

- Hash values for each point.
- Index array, which when used to access the hash value will give us the values in ascending order.

We now want to make the buckets in the form of an `int2` array, where we want the start and size of each bucket. The size will be how many points collide on this value, and the start will be the index of the first element which has this hash value when we access the hash values with the index array. Since a GPU is not well suited for this problem, we do it on the CPU. We do this on the CPU with a basic for loop that iterates through the hash value array with the index array, i.e $hash\_value[index[0 -> n]]$ and does:

- If the new value is equal to the value before we increase the size of the bucket by 1
- If not we set the current index as the start of a new bucket and set the size to 1

We do this for both the reference points and the query points. We should now have an array of type `int2` where the first `int` will contain the start of the bucket in the index array and the second int will contain the size of the bucket.

4.2.12 Matching query buckets and reference buckets

We now want to match buckets with query points to buckets with reference points to create an array of type `int4` which contains:

- Start index of bucket in index array for query points
- Number of query points in the bucket
- Start index of bucket in index array for reference points
- Number of reference points in the bucket

This is done on the CPU, by using a while loop where we have two counters and iterate though the reference and query point `int2` arrays. If we find 2 buckets with the same value we write the `int2` values from the query bucket array and reference bucket array to the new `int4` array. This gives us an array matching the buckets including the start index and size of these buckets. We now have everything we need to perform the short list brute-force search.

4.2.13 "Short" list 2NN brute-force search

Because of how unpredictable this problem is, the kernel we have created is far from a perfect solution. The short list search will search for the 2NNs for the points in the given query bucket in the given reference bucket. It is given the index to where the two buckets start, and the size of each of these buckets in an array of type `int4`. For this kernel the most time consuming part will be redundant reads, therefore we want to minimize the amount of times we need to read any value from global memory. To do this we use shared memory. However, as there is no way of knowing how big a list in the "short" list search will be, we have to decide on a shared memory size which will not be optimal for all problem sizes. Using more shared memory than we utilize can lead to bad occupancy and on the other hand using too little will lead to very bad performance since the same values will be read from global memory multiple times. In the end we decided to compromise, using shared memory equal to the number of warps we run for each block. This is far from an optimal solution especially for the data set we use to test on.

2NN brute-force kernel

Our kernel can be thought of as a double for loop:

- In the outer loop we will read in the query points, we read in one point for each active warp in the block, the reads will be coalesced where each thread in the warp is in charge of 4 `half` values.
- Then in the second loop we first read in reference points equal to the number of active warps, each warp reads in one reference point and writes this to shared (this will be done in the same way as with the query point). Every warp then calculates the distance for the query point it is in charge of to every reference point in shared memory. This is done by using warp level primitives and `half` vector operations.

To minimize the amount of times we need to read/write to the global array which holds the 2NNs (min2_index) values for the query points, we will save up to 32 values before updating. We do this by letting each thread in the warp in charge of the query point hold one value, then using indexing to figure out which threads already have values. After all 32 threads have a value we do a warp wide reduction to get the 2NN and the index of the NN. We then write this to the min2_index array which

holds the 2NN and the index of the NN in global memory. However as this is an array which every thread will be able to read/write to, we have to use CUDA atomics to ensure that undefined behaviour does not occur. We explain how this is done in section 4.2.14

4.2.14 Atomic update of the min2 array

As we are running multiple streams which can all access the min2_index array, we face the problem of how to make sure that undefined behavior (such as when 2 threads attempt to update the same value) does not occur. To do this we use the CUDA compare-and-swap atomic. This atomic lets us define our own compare function and then safely swap, (or try to swap) the value in device global memory. We wrote a function (called `atomic_min2_update` in our code) which does this. The min2_index struct is made of one `half2` value where x is the distance to the NN and y is the distance to the second NN, it also holds one 32 bit unsigned int which is the index of of the NN. We made this value specifically to use it in our compare-and-swap atomic, as this atomic only works for 32 or 64 bit structs.

4.2.15 Writing to the output array

When we are done with our algorithm, we run a kernel which reads from the min2_index array and writes the results to the output matches array. The reads and writes will be coalesced and it is a fairly simple kernel. This is also where we can apply the distance ratio if we want to. We chose not to, since we can never be sure if our points are true NN or not, making it hard to say how efficient this would be.

4.3 Converting from float to half on device

This kernel will be used to convert `float` values to `halfs`. To convert we use a simple kernel where each block has 64 threads and each thread does the following:

- Reads 2 floats from the input array
- Converts to a half2
- Writes to the output array

Both the reads and writes are coalesced.

4.4 Naive CUDA 2NN brute-force

As a metric for testing the efficiency of our LSH on GPU and brute-force with cuBLAS, we compare it to a 2NN naive CUDA brute-force. This is a fairly straight forward brute-fore consisting of 2 kernels.

1. **Kernel 1** Each block has one warp and is in charge of calculating the Euclidean distance between one query point and one reference point. Both points will be read with coalesced reads. Then the warp will use warp level primitives to reduce the distance to the first thread, which will then write this distance to the distance array belonging to the query point.

2. **Kernel 2** Each block will scan a distance array for the 2NNs of the query point, this will be done with coalesced reads and warp/block wide reductions. When done it will write the results to the matches arrays.

This is a fairly naive implementation because we barely use shared memory at all, leading to a very high number of redundant reads of the same reference/query points.

Chapter 5

Evaluation and test results

In this chapter we test and evaluate the 2 algorithms we implemented. For brute-force with cuBLAS see section 5.4, for LSH on GPU see section 5.5. We mostly test on the ANN_SIFT1M data set, which we look closer at in section 5.2. We also test to see how to best transfer the SIFT vectors from host to device, when we want to use `half` values section see 5.3. We also compare our 2 implementations in section 5.6.

5.1 Testing environment used

When testing, the GPU is also used to run the screen so 100% accurate measurements are impossible, however the effects of this should be minuscule. We also often run multiple times and take the average making this even less noticeable. We test in Windows Subsystem for Linux (WSL) with Ubuntu. The CPU used for testing is a amd ryzen 5 5600x 6-core 3.7 ghz, and the GPU is a GeForce RTX 3060 with 12GB memory.

5.2 A closer look at the ANN_SIFT1M data set

For most of the tests we perform we use the ANN_SIFT1M data set introduced in the paper "Product Quantization for Nearest Neighbor Search" [9] by Hervé Jégou, Matthijs Douze and Cordelia Schmid. This is a data set which contains 10^4 query points and 10^6 reference points. It is a somewhat known data set, often cited when testing ANN algorithms. Also it is of a suitable size for what we want to test, we were not able to find other sets of similar sizes. To make sense of the results we get, we decided to run a number of tests on the data set itself. To do this we use a naive brute-force 2NN algorithm implemented with CUDA (see section 4.4 for implementation). As SIFT has an optional threshold step (see 2.1.4), we test for varying thresholds. For each of the 10^4 query points when matching we get 3 possible outcomes, wrongly matched, correctly matched and no match (the 2NNs for the query point do no satisfy the SIFT threshold).

Results

Matches using Naive CUDA brute-force, no threshold	
Wrongly matched	73
Correctly matched	9927
No match	0
Matches using Naive CUDA brute-force, 0.8 threshold	
Wrongly matched	0
Correctly matched	706
No match	9294
Matches using Naive CUDA brute-force, 0.9 threshold	
Wrongly matched	0
Correctly matched	1643
No match	8357
Matches using Naive CUDA brute-force, 0.95 threshold	
Wrongly matched	0
Correctly matched	3470
No match	6530
Matches using Naive CUDA brute-force, 0.99 threshold	
Wrongly matched	0
Correctly matched	7820
No match	2180
Matches using Naive CUDA brute-force, 1 threshold	
Wrongly matched	0
Correctly matched	9854
No match	146

Table 5.1: Shows how well a CUDA naive brute-force matches SIFT vectors from the ANN_SIFT1M data set, with differing SIFT distance ratio thresholds

Reflections

We see that with no threshold, the brute-force does not get all the 10^4 matches correct, however this is likely due to some of the 2NNs for some of the query points being equal. This can be seen in the last test, where we have a threshold of 1, meaning all solutions where the 2NNs are equal will be rejected. We see that there are 146 matches where this is the case, and 73 matches which are wrongly matched in the no threshold test. This is most likely caused by the order of which the points are compared. This can also be seen as using different number of warps in our brute-force when comparing would lead to different numbers of wrong matches for the no threshold test, because the order of which we compare the points will change. For the 0.8 threshold test which is what Lowe [11] recommends in his paper, we see that very few matches (706 of 10000) pass this test. We also see that for a threshold of 0.99 there are still 2180 points where no matches are found. As we can see from the tests the data is very clustered meaning we could probably be a bit more forgiving when testing maybe

even counting either one of the 2NN as a correct match when testing for LSH.

5.3 Copying data from host to device when using `halfs`

Copying `float` data from host to device, and converting to `halfs` is something we have to do for both our LSH implementation and brute-force with cuBLAS implementation. As in this case we need to copy all the data at once before we can continue working, we have 2 choices:

1. Use zero copy to read directly from host in the `float` to `half` conversion kernel
2. Allocate a float array, copy to device with `cudaMemcpy`, call `float` to `half` conversion kernel

We know that when moving data from host to device using `cudaMemcpy` is the fastest way. However we do not only need to move the data we also need to convert it to `halfs`. We test the difference in performance between choices 1 and 2 on arrays of varying sizes:

Results

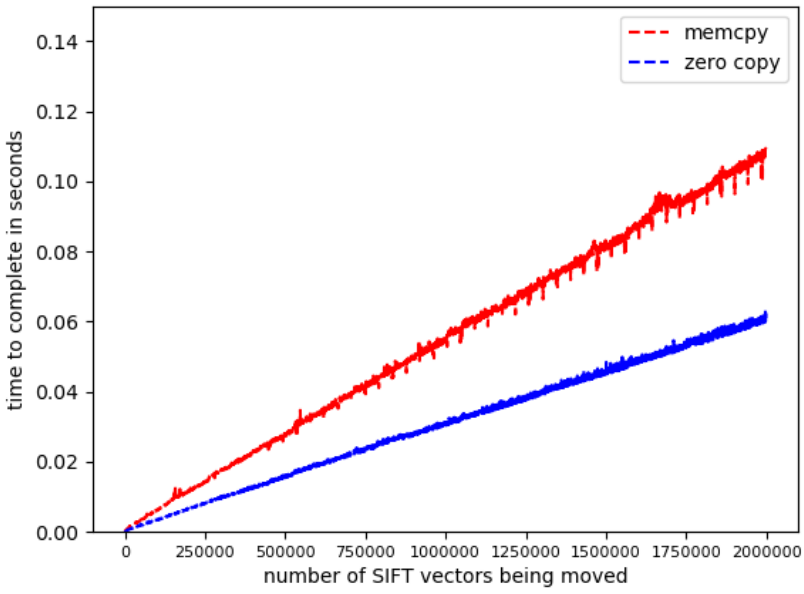

Figure 5.1: Shows the time used to copy and convert the `float` SIFT vectors from host to device using zero copy and using `cudaMemCpy`

Reflections

We clearly see that when there is a need to convert the data, reading directly from host using zero copy in the `float` to `half` CUDA conversion kernel is faster.

5.4 Evaluation of brute-force with cuBLAS for SIFT feature matching

In this section we test and evaluate the brute-force with cuBLAS implementation. For most of the test we use the ANN_SIFT1M data set, see section 5.2 for a short look at the data set. As this is a brute-force were the data does not affect the performance, we also test on pseudo data made with the C++ `rand` function. We test the recall with differing thresholds, and look at how using `floats` compares to using `halfs` in terms of recall (5.4.1). We test the `cublasHgemm` cuBLAS calls performance with different input sizes (5.4.2), and look at how padding affects the performance (5.4.3). We also evaluate the reduction kernel (5.4.4), test how using concurrent streams affects the performance for different problem sizes (5.4.5), and lastly we look at the overall performance in terms of time to complete on the ANN_SIFT1M data set averaged over 100 runs when using different levels of precision (5.4.6).

5.4.1 Recall

While this implementation is a brute-force, we have made an assumption, and that is that the norm of the input vectors will be the same for every vector. While this is the case on the pseudo random data we produce, it was not the case for the real data set ANN_SIFT1M. The average norms for both the query points and reference points are:

10^4 query points 10^6 reference points ANN_SIFT1M data set	
Average norm query points	Average norm reference points
508.637939	512.592285

Table 5.2: ANN_SIFT1M data set norms

As we can see from table 5.2 the average norm on the real test data set ANN_SIFT1M does not meet our assumption of equal norms on all vectors. To also see how our implementation does when our assumption is true, we also test on pseudo data. The norms for the pseudo data are:

10^4 10^6 pseudo data set	
Average norm query points	Average norm reference points
1.0	1.0

Table 5.3: pseudo data set norms

Recall on real and pseudo data with and without thresholds

As we have implemented both a version using `halfs` and a version using `floats` we will test the recall on both. Also as cuBLAS gives us the choice to specify what precision we want the GEMM calculation to be done in, we test both 16 and 32 bit calculations. When testing for each query point we have 3 possible outcomes:

- Correct match, we find the right reference point.
- Wrong match, we find the wrong reference point.
- No match, the distance ratio for the 2NNs found does not satisfy the distance ratio threshold, and we do therefore have no suitable solutions.

The recall when we use a threshold will be calculated out of the total number of query points minus the number for which we found no match.

On the ANN_SIFT1M data set we did 6 different tests with varying thresholds. These thresholds are, no threshold, 0.8, 0.9, ,0.95, 0.99 and 1. For the pseudo data set we did also do one test with threshold of 0.8, but as not a single point passed the threshold test we decided not to do anymore testing. This is after all pseudo random data which means it will be evenly distributed, leading to the distance between points being almost equal. Therefore the only data from the pseudo random data tests we have decided to include is the no threshold test. For the pseudo random data we use the naive CUDA brute-force 4.4 as the true values. For the real data we use the given true values, we only consider the query a correct match if we find the NN according to the true values.

Test results on the ANN_SIFT1M data set

Recall using `halfs`, no threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	1229	1186
Correctly matched	8771	8814
No match	0	0
Recall	0.8771	0.8814
Recall using `floats`, no threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	1173	1173
Correctly matched	8827	8827
No match	0	0
Recall	0.8827	0.8827

Table 5.4: Recall on the ANN_SIFT1M data set using `halfs` and `floatss` with no threshold

Recall using `halfs`, 0.8 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	6	6
Correctly matched	720	719
No match	9274	9275
Recall	0.991735537	0.991724137
Recall using `floatss`, 0.8 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	0	0
Correctly matched	487	487
No match	9513	9513
Recall	1.0	1.0

Table 5.5: Recall on the ANN_SIFT1M data set using `halfs` and `floats` with 0.8 threshold

Recall using `halfs`, 0.9 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	23	27
Correctly matched	1825	1832
No match	8152	8141
Recall	0.9875541125541125	0.9854760623991393
Recall using `floatss`, 0.9 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	0	0
Correctly matched	1256	1256
No match	8744	8744
Recall	1.0	1.0

Table 5.6: Recall on the ANN_SIFT1M data set using `halfs` and `floats` with 0.9 threshold

Recall using `halfs`, 0.95 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	113	107
Correctly matched	4004	4015
No match	5883	5878
Recall	0.9725528297303863	0.9740417273168365
Recall using `floatss`, 0.95 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	6	6
Correctly matched	2910	2910
No match	7084	7084
Recall	0.9979423868312757	0.9979423868312757

Table 5.7: Recall on the ANN_SIFT1M data set using `halfs` and `floats` with 0.95 threshold

Recall using `halfs`, 0.99 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	561	523
Correctly matched	7792	7813
No match	1647	1664
Recall	0.9328385011373159	0.9372600767754319
Recall using `floatss`, 0.99 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	268	268
Correctly matched	7247	7247
No match	2485	2485
Recall	0.964337990685296	0.964337990685296

Table 5.8: Recall on the ANN_SIFT1M data set using `halfs` and `floats` with 0.99 threshold

Recall using `halfs`, 1 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	945	905
Correctly matched	8558	8593
No match	497	502
Recall	0.9005577186151742	0.9047167824805222
Recall using `floatss`, 1 threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	1100	1100
Correctly matched	8760	8760
No match	140	140
Recall	0.9842696629213483	0.9842696629213483

Table 5.9: Recall on the ANN_SIFT1M data set using `halfs` and `floats` with 1 threshold

Recall using `halfs` on pseudo random data set, no threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	554	263
Correctly matched	9446	9737
Recall	0.9446	0.9737
Recall using `floats` on pseudo random data set, no threshold		
	16 bit calculations	32 bit calculations
Wrongly matched	36	0
Correctly matched	9964	10000
Recall	0.9964	1.0

Table 5.10: Recall on a 10^4 query points, 10^6 reference points pseudo random data set using `halfs` and `floats` with no threshold

Reflections

We see that with no threshold on real data (table 5.4) there is very little variance in recall no matter if we use floats, half, 16 bit or 32 bit calculations. Also there is an almost identical number of wrong matches, which makes us believe that the points we fail to find the true NN of either have a bad (in terms of our assumption that all norms should be equal) norm them self, or the NN has a bad norm. For the no threshold tests (table 5.4) 981 of the wrong matches were shared between all the tests, which at least suggests that these points are not easy to correctly match with our approach which uses the dot product. When using a threshold on the real data (tables 5.5 5.6 5.7 5.8 5.9), we see that the recall is over 0.9 for all tests, even when the threshold is 1 (meaning we only reject solutions when the 2NNs are equal). On the pseudo data (table 5.10), we see that the recall varies a lot more depending on the precision of the calculation and the type used, with variance from best recall at 1.0 to worst at 0.94. However we still see that even when we use `halfs` with 16-bit calculation we get a recall of over 0.94. This suggests that even when the data is uniformly distributed we can get good recall with `halfs` and 16 bit calculation, meaning it should be able to get very good recall when the data is real and meets the 0.8 threshold. Also if good recall is crucial one could always calculated the norms, but this would obviously be slower.

5.4.2 `cublasHgemm` evaluation with differing problem sizes

For how we use the `cublasHgemm` call see section 4.1.9. As we are performing matrix - matrix multiplications, the data we use when testing performance will have absolutely no effect over the results, therefore we use pseudo random data. As the cuBLAS GEMM call is the the most time-consuming part of our algorithm we test the `cublasHgemm` call on 3 problem sizes:

1. 10^4 query points and 10^6 reference points
2. 10^6 query points and 10^4 reference points
3. 10^4 query points and 10^4 reference points

By testing we mean looking at the amount of time in seconds it takes to perform the matrix-matrix multiplications needed to dot every query point with every reference point. When testing there is only one thing we can change to affect the the overall performance, and that is the number of query points we use per `cublasHgemm` call. Having a low number of query points will obviously lead to more calls needing to be done and vice versa. This is partly what made us decide what sizes we wanted to test on. For problem size 1 we will not be able to increase the number of query points per call to higher than around 4200, as we do not have memory for more. This means the minimum amount of `cublasHgemm` calls we can do is 3. For problem size 2 we will be able to increase the number of query points per call up to around 400k. For problem size 3 we will be able to do the whole query in one call.

Test results for problem size 1

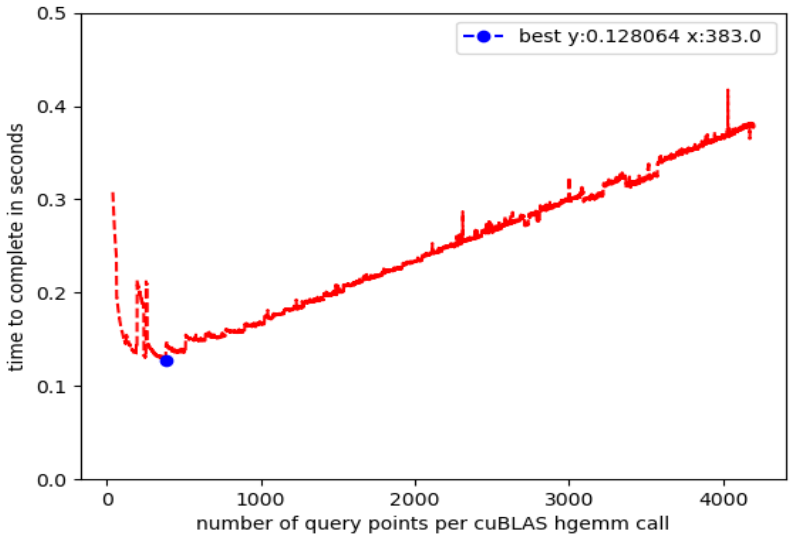

Figure 5.2: Shows time in seconds needed to complete all necessary `cublasHgemm` calls needed to multiply 10^4 query points with 10^6 reference points, where the number of query points per call is in the range 40 - 4200

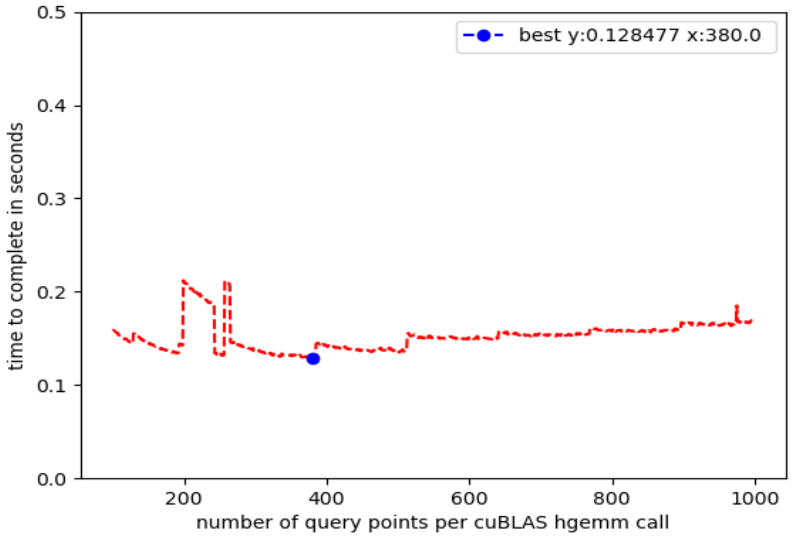

Figure 5.3: Shows time in seconds needed to complete all necessary `cublasHgemm` calls needed to multiply 10^4 query points with 10^6 reference points, where the number of query points per call is in the range 100 - 1000, and we average run time over 10 runs

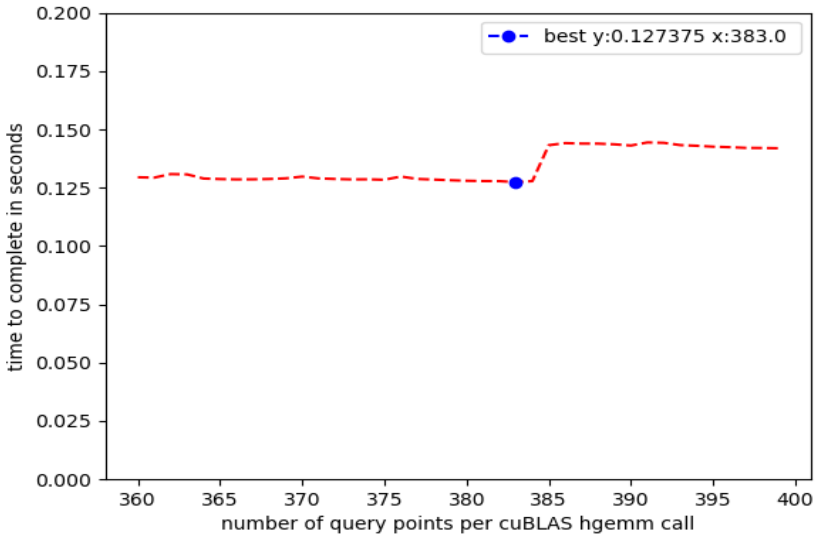

Figure 5.4: Shows time in seconds needed to complete all necessary `cublasHgemm` calls needed to multiply 10^4 query points with 10^6 reference points, where the number of query points per call is in the range 360 - 400, and we average run time over 100 runs

Test results for problem size 2

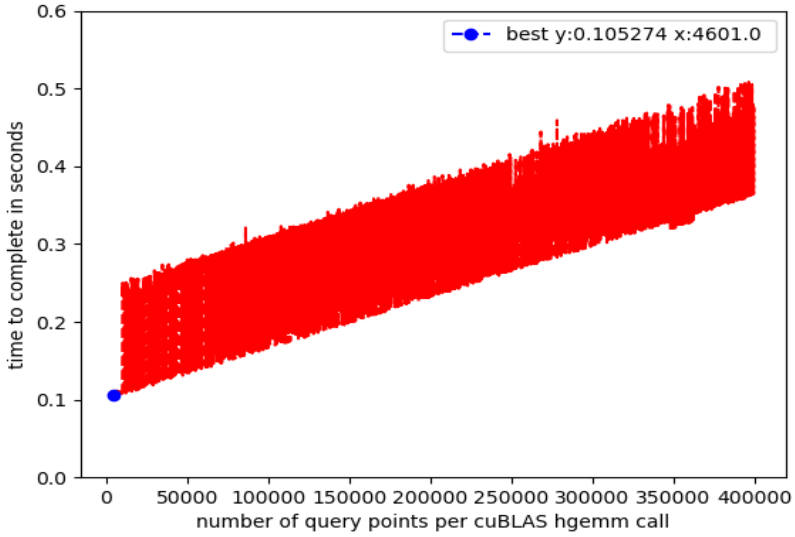

Figure 5.5: Shows time in seconds needed to complete all necessary `cublasHgemm` calls needed to multiply 10^6 query points with 10^10 reference points, where the number of query points per call is in the range 40 - 400000

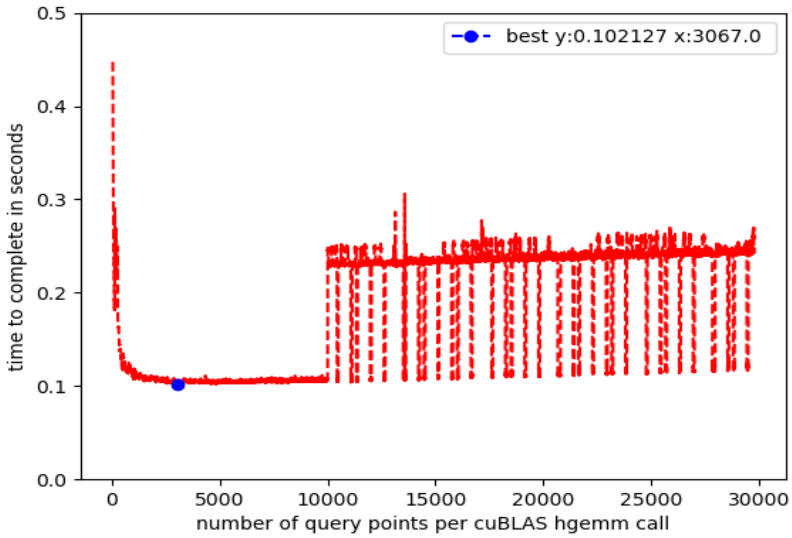

Figure 5.6: Shows time in seconds needed to complete all necessary `cublasHgemm` calls needed to multiply 10^6 query points with 10^10 reference points, where the number of query points per call is in the range 40 - 29729

Test results for problem size 3

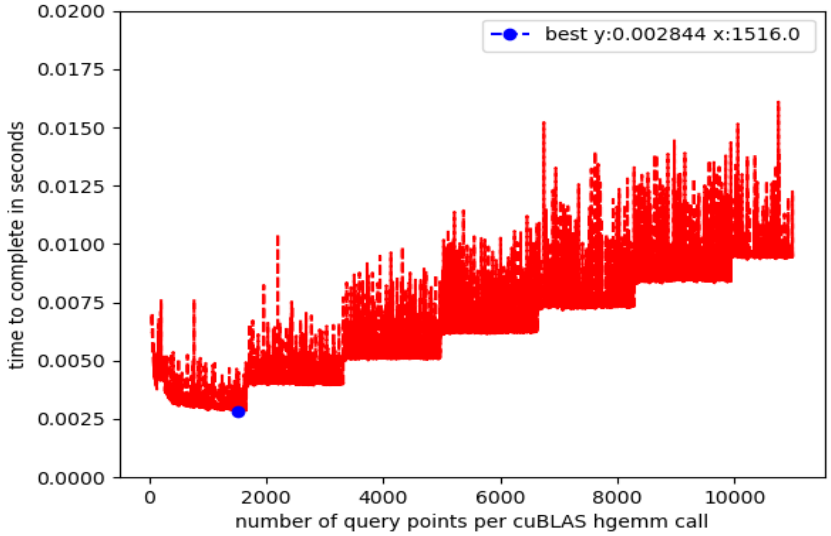

Figure 5.7: Shows time in seconds needed to complete all necessary `cublasHgemm` calls needed to multiply 10^4 query points with 10^10 reference points, where the number of query points per call is in the range 40 - 10500

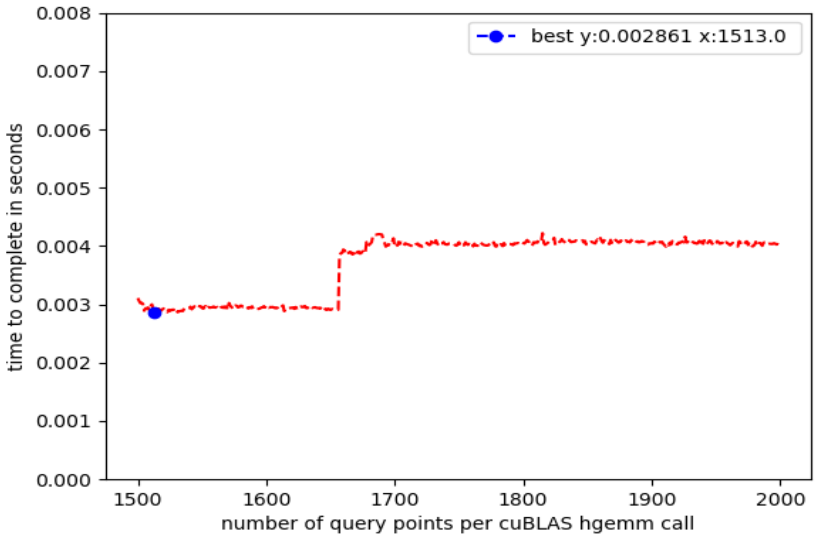

Figure 5.8: Shows time in seconds needed to complete all necessary `cublasHgemm` calls needed to multiply 10^4 query points with 10^10 reference points, where the number of query points per call is in the range 1500 - 2000, and we average run time over 100 runs

Reflections

We see that the performance is extremely dependent on the on the number of query points we use per call, this is true for all of the problem sizes. We see that the time used can in the worst case be up to 5 times slower as seen in figure 5.5, where the best case is approximately 0.1s and the worst case is 0.5s. One of the main purposes of doing these tests was trying to find a formula which gives us the optimal performance for any query. However from the data we have here this seems very hard. Also most likely cuBLAS is optimized differently for different GPUs, so how much meaning would be in finding such a formula we are not sure. What we can see very clearly from the data however is that the best performance is not achieved when we use the maximum number of query points possible per `cublasHgemm` call. On the contrary the best performance is usually achieved more or less within the first 20% of the possible max, this is true even when we can do the whole query in one `cublasHgemm` call as seen in figure 5.7. It seems that if performance is of importance, testing with the approximate problem size on the GPU which is to be used, is the only way to find the a suitable size for the number of query points per call.

5.4.3 Padding for cuBLAS `cublasHgemm`

We test to see if it is beneficial to use padding to make sure our `cublasHgemm` call meets all the conditions which should be met for maximum performance when using cuBLAS with tensor cores see (2.8.3). The conditions in question are:

- Condition 1: `m % 8 == 0`
- Condition 9: `intptr_t(C+ldc) % 16 == 0`

The conditions can both be met by padding the reference point array. However to do this when our input is of type `half` in device memory it would mean we would have to allocate a new array, set the part of the array we need to pad to zero, and then copy from the old array to the new one. We test to see if this time loss is worth it or not. We test by comparing the overall time it take to complete the algorithm on a data set with 10^4 query points and 7 - 1040007 reference points.

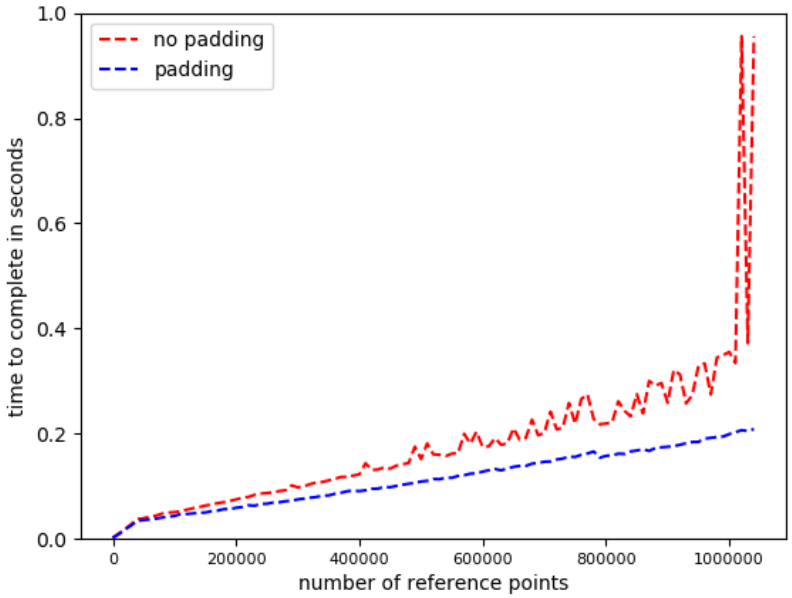

Figure 5.9: Shows the difference in execution time when using padding vs when not using padding on data which does not meet cuBLAS's conditions for maximum performance if we do not pad

Reflections

We see that despite the time loss caused by having to pad the reference array we still get better performance when the number of reference points is of any significant size. We also see that padding will save us more time the bigger the reference points array is, this is despite the fact that this would mean a bigger overhead from padding.

5.4.4 Reduction kernel evaluation

We test to see how varying the number of warps per block will affect the outcome in terms of time used for our reduction kernel, for implementation see 4.1.10. As what number of warps work best will vary according to

the problem size etc. we will only test for the real data set we have i.e ANN_SIFT1M. For the number of query points per batch we use 383 as this is what gave us the best performance when testing `cublasHgemm` for this test size.

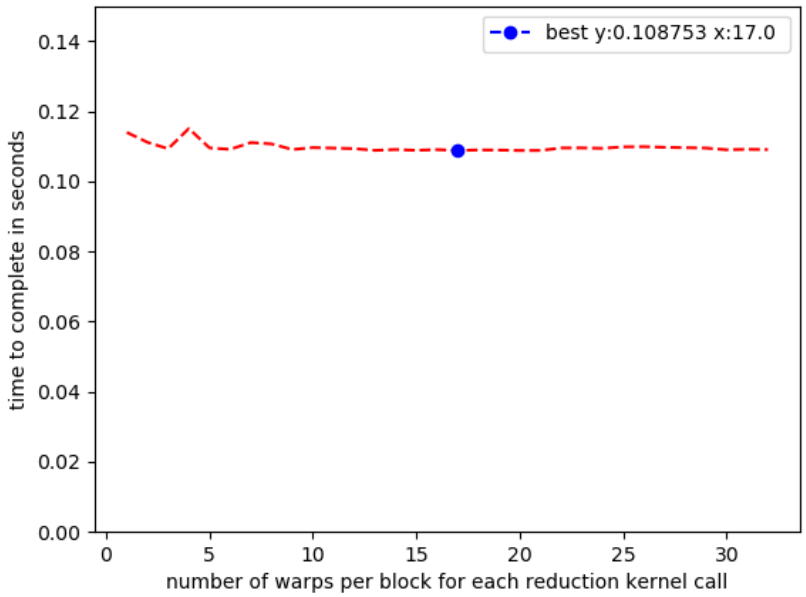

Figure 5.10: Shows how varying the number of warps per block affect time used for the query when testing on the ANN_SIFT1M data set, we average over 100 runs

Reflections

We see that the best performance is achieved with 17 warps per block. We also see that the variance is minuscule, with a maximum variance of around 7ms. This suggests that the number of warps used is less of a priority when optimizing.

5.4.5 Concurrent streams for cuBLAS brute-force

As we have made it possible to use concurrent streams (see 4.1.8) we test to see how using concurrent stream affect the overall performance on 3 different problem sizes:

1. 10^4 query points and 10^6 reference points
2. 10^6 query points and 10^4 reference points
3. 10^4 query points and 10^4 reference points

We test when the data is either in device or host memory and of type `float` or type `half`.

Results

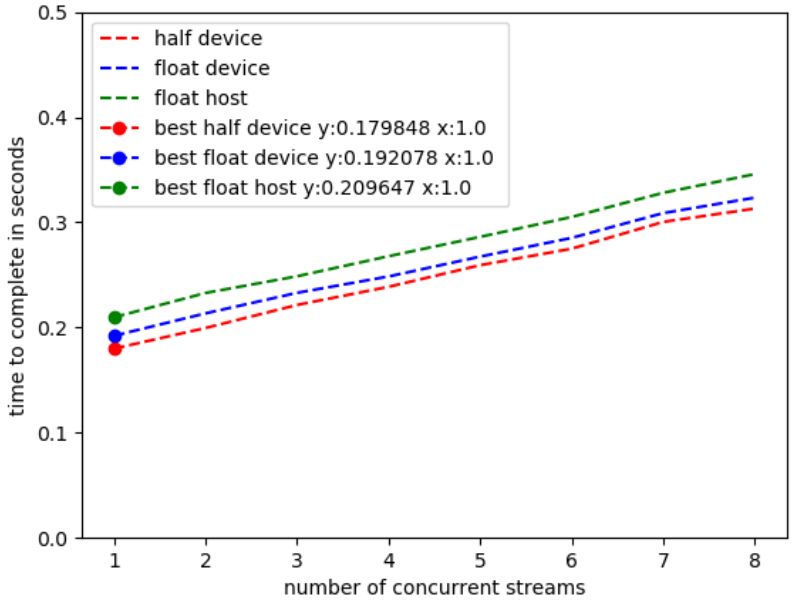

Figure 5.11: Shows how the number of concurrent streams affects performance when testing with 10^4 query points and 10^6 reference points

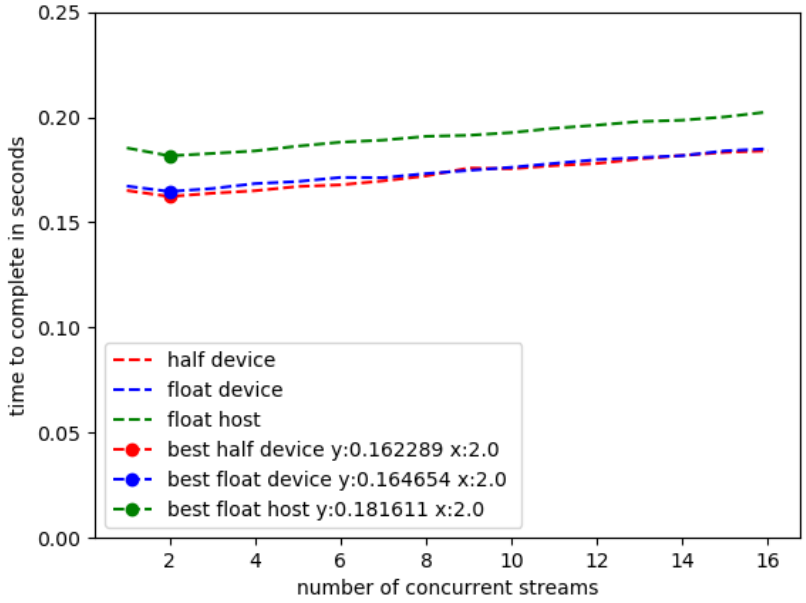

Figure 5.12: Shows how the number of concurrent streams affects performance when testing with 10^6 query points and 10^4 reference points

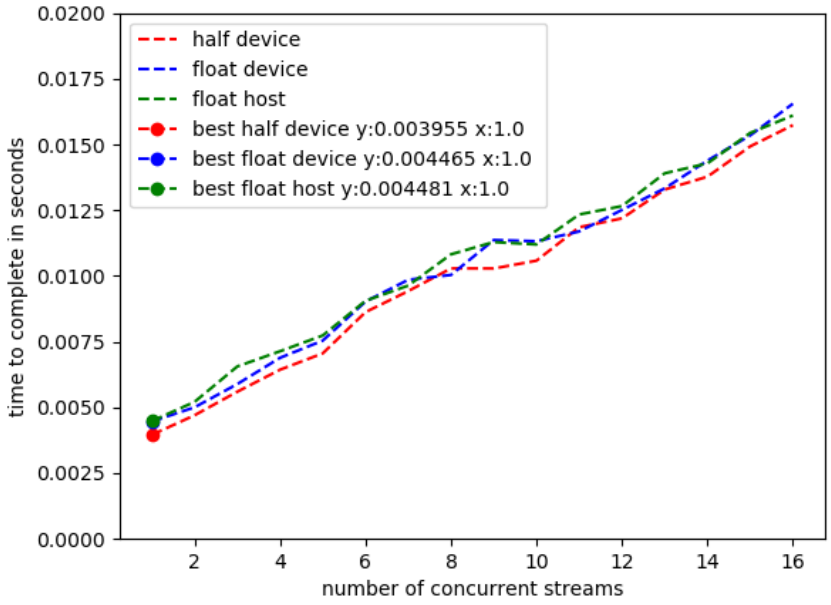

Figure 5.13: Shows how the number of concurrent streams affects performance when testing with 10^4 query points and 10^4 reference points

Reflections

We see that we only benefit from streams on problem 2 (figure 5.12). In hindsight this is somewhat expected as we saw that cuBLAS in general works best when we are only using 1 stream. It is however a bit surprising that we get a speed up from using 2 streams even when the input is of type `half` and in device memory. Our original thought was that we would get a speed up because we would get better occupancy when converting from `float` to `half` and when reading from host memory. Two plausible causes are: the reduction kernel benefits from concurrent streams, or that simply scheduling in concurrent streams helps reducing the overhead from starting a kernel/calling a cuBLAS function.

5.4.6 Best results achieved on the ANN_SIFT1M data set

Considering all the results from the above tests, the best results in terms of time used averaged over 100 run we were able to achieve on the ANN_SIFT1M data set were:

ANN_SIFT1M data set, `halfs` and 16-bit calculations			
input type	`float` host	`float` device	`half` device
time in seconds	0.208720	0.184456	0.177090

Table 5.11: Shows time in seconds used on the ANN_SIFT1M data set with `halfs`, with differing input types and locations, we take the average over 100 runs

Time in seconds used on the ANN_SIFT1M data set		
	16 bit calculations	32 bit calculations
`float`	0.446409	0.903390
`half`	0.184456	0.212507

Table 5.12: Shows time in seconds used on the ANN_SIFT1M data set, with differing types and levels of precision used (16-bit and 32-bit), input is type `float` in device memory, we take the average over 100 runs

Reflections

We see that using `halfs` over `floats` gives us 5X speed up, when we use 32-bit calculations for the `floats` and a 2.4X speed up when we use 16-bit calculations for the `floats` (table 5.12). We also see that there is a 1.17X speed up of having the input vectors in device memory and of type `half` compared to `floats` in host memory. Note that the float test times can most likely be improved, as we did not do the same amount of testing to reach the optimal configuration in the same way we did for `halfs`. However a minimum of at least a 2X increase in speed is to be excepted.

5.4.7 Overall reflections and real time matching of SIFT points

Having to test to find the best configuration for the input is not optimal. Using concurrent streams does not seem to be a good idea when we are using a brute-force like approach as it is very easy to get good occupancy with one stream, however not using the default stream i.e calling everything in stream 1 instead of 0 is important especially when using cuBLAS so we can get asynchronous calls.

Real time

One of our goals was to create an algorithm which would let us match SIFT vectors in real time so that it could be used to calculate depth. On a RTX 3060 12GB GPU we can achieve real time when at 27 fps with 45k query points and 45k reference points (0.036530s per frame).

5.5 LSH GPU Evaluation

For our LSH on GPU implementation we only test on the ANN_SIFT1M data set. As the data set contains values for up to the 100 nearest neighbours and LSH is an ANN algorithm we will also see how our implementation does if we count up to the 1 - 100 NN as a true match. Also we will look at the difference in performance when we only count points which pass the 0.8 SIFT threshold test. In the tests we will write this as follows:

- Recall @ 1, means we only count the NN
- Recall @ 100, means we count any of the 100NNs as a true match.

- Recall @ 1 0.8 threshold, means we only count the values where the 0.8 threshold is met, we use a naive brute force as a base for what points pass or not. For the 0.8 threshold there are 706 points which pass, as shown in (5.2).

We refer to this as different levels of precision.

For LSH on GPU we run 7 tests. In section 5.5.1 we test to see how good recall we can get in under 4 seconds. In section 5.5.2 we test to see how long it takes to get 0.8 recall. In section 5.5.3 we look at how the number of bits affects the number of iterations needed to get 0.8 recall. In section 5.5.4 we look at how padding the cuBLAS call affects performance. In section 5.5.5 we look at how running concurrent streams affects performance. In section 5.5.6 we look at the time each kernel in the LSH algorithm uses. And lastly in sections 5.5.7 we reflect on our LSH implementation.

5.5.1 Best recall under 4 seconds on the ANN_SIFT1M data set

As LSH is an ANN algorithm, how high recall we want becomes a balancing act with time used. One of the tests we do is to see how high recall we can get under 4 seconds. LSH has 2 components we can change to affect the performance, number of bits and number of iterations. We will see how changing these will affect the overall performance when trying to get the maximum recall possible in under 4 seconds.

How we test

We run the algorithm in a double for loop where we increase iterations in the inner loop and number of bits in the outer. Every time the algorithm takes over 4 seconds 3 times in a row, we break the inner for loop and increase the number of bits used in the buckets by one (meaning we double the number of buckets). We do however not change the number of iterations again as the number of buckets just doubled. There is no particular reason for choosing 4 seconds as the max time, however any more would take very to long to test.

Best recall in under 4 seconds by number of bit and iterations run

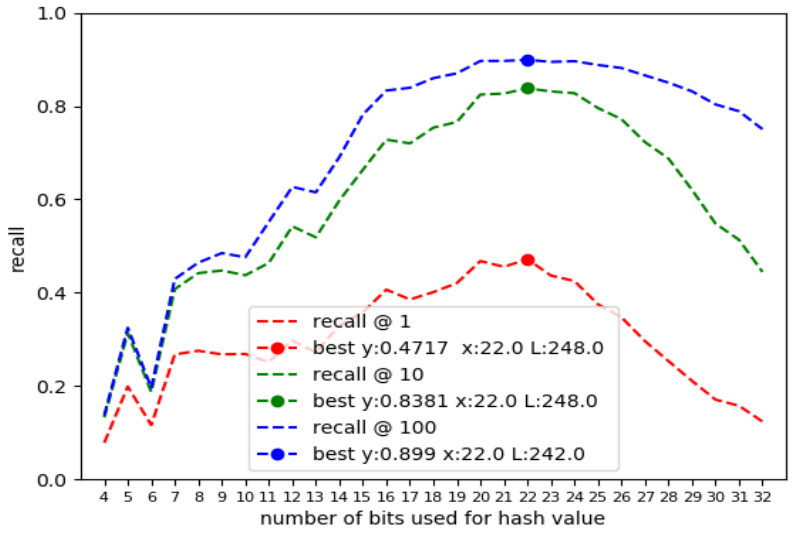

Figure 5.14: Shows the best recall in under 4 seconds by number of bits, and iterations run on the ANN_SIFT1M data set, recall @ 1, 10, 100

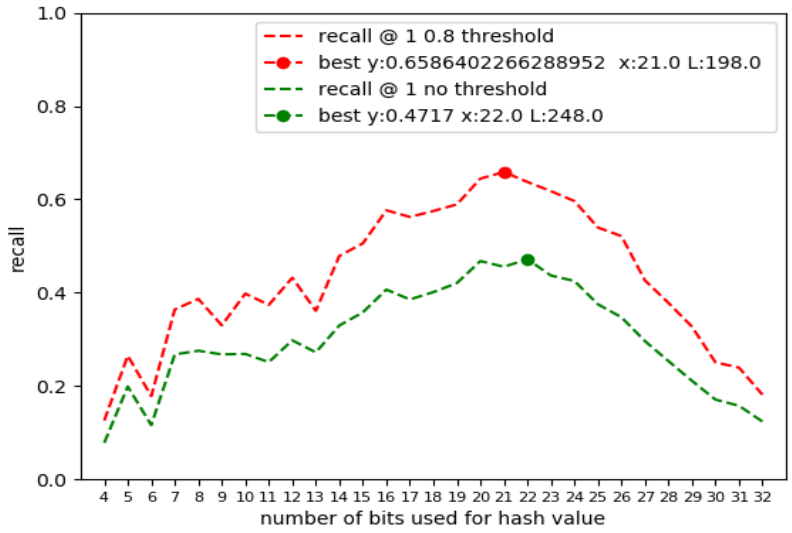

Figure 5.15: Shows the best recall in under 4 seconds by number of bits, and iterations run on the ANN_SIFT1M data set, recall @ 1, 1 0.8 threshold, 100

Reflections

As we pick the best single runs from our test results, this is a more optimistic view rather than the average. This is especially true for test with lower number of bits, as we will only have time to do a few iterations meaning the whole algorithm becomes a lot more chance based. When the number of bits increase however the variance is much smaller giving us much more stable and realistic results. We see that the best recall we get when only the NN counts is about 0.47 (figure 5.14), on the other hand when we use a the 0.8 threshold we get 0.65 (figure 5.15), this suggests that LSH would work bettor on SIFT data which follows the 0.8 distance ratio rule. We see that in general for this data set around 20-22 bits seems to give the best performance. For recall @ 100 we get a recall of 0.899 which is almost the double of what we get when we only count the NN as the true value (figure 5.14).

5.5.2 Best time for recall of 0.8 on the ANN_SIFT1M data set

Getting around a 0.8 in recall is desirable if we were to use this algorithm. We test to see how much time it would take to reach this recall.

How we test

We test by ruining the algorithm in a double for loop where we increase iterations in the inner loop and number of bits in the outer. Every time we get over 0.8 recall we add 1 to a flag, when the flag is 3 we break the inner loop and go the the next. We want to get 0.8 recall at least 3 times as this is a probability based algorithm. We only test up to 27 bits, as going any higher requires a lot of time and when testing the performance seems to peak around 20-22 bits anyway.

Results

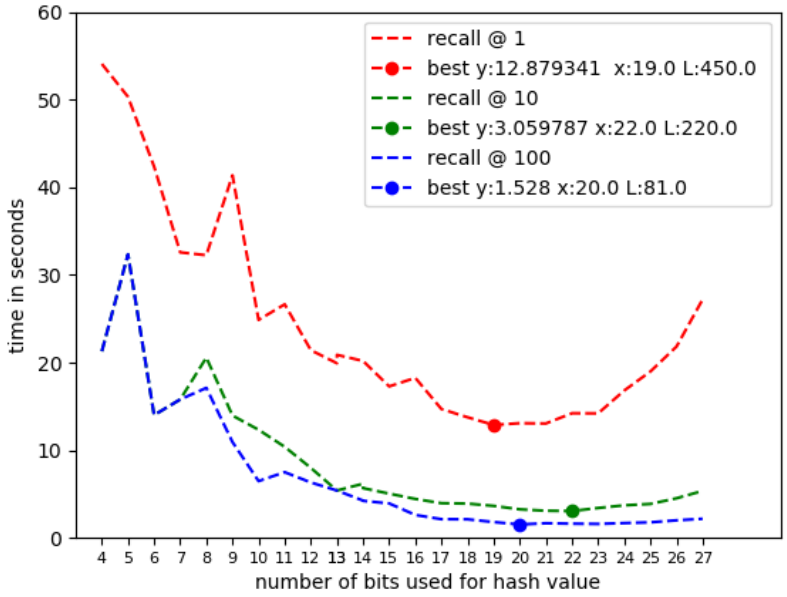

Figure 5.16: Shows best time in seconds for 0.8 recall on the ANN_SIFT1M data set, recall @ 1, 10, 100

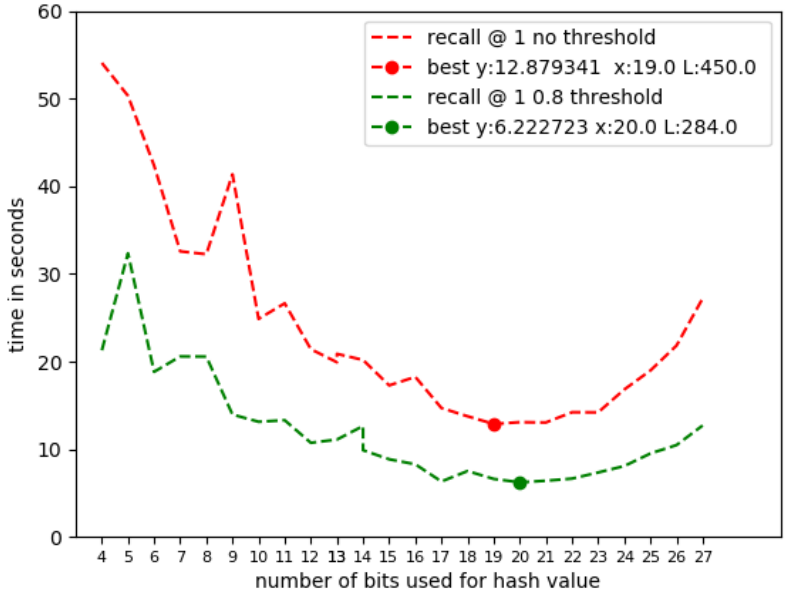

Figure 5.17: Shows best time in seconds for 0.8 recall on the ANN_SIFT1M data set, recall @ 1, 1 0.8 threshold

Reflections

We see that the best results in terms of time used for all levels of precision is in the range 19 - 22 bits. The time for getting 0.8 recall when only the NN counts is about 4X slower than when we only need to find one of the 10NN (figure 5.16). This suggests that LSH is much better at finding the ANN vs the NN. We also see that it takes about half the time to get 0.8 recall when we use the 0.8 threshold vs when we don't (figure 5.17).

5.5.3 Number of bits vs number of iterations needed to reach 0.8 recall

We look at how the number of bits used for hash values change how many iterations we need to do to get over 0.8 recall for different levels of precision. The data we use is the same as for the test at 5.5.2 where we show the number of iterations instead of time used.

Results

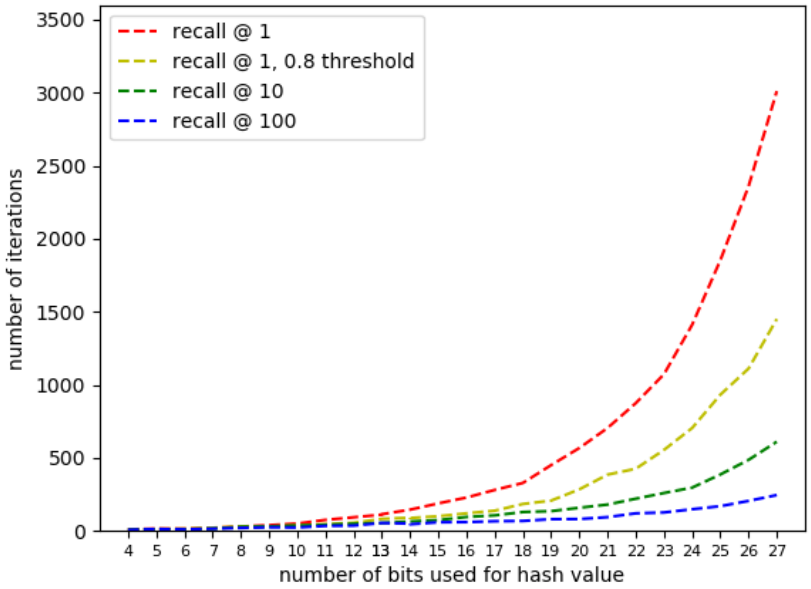

Figure 5.18: Shows how changing the number of bits change the number of iterations needed to achieve 0.8 recall on the ANN_SIFT1M data set

Reflections

We see that around 27 bits the recall @ 1 needs around 3000 iterations, while the recall @ 100 needs around 250, this is a 12X increase. The difference between recall @ 1 with and without the threshold is probably the most notable here, as it shows that LSH is better at finding the matches where the SIFT points pass the SIFT distance threshold test.

5.5.4 Padding

As with the brute-force with cuBLAS we have the choice of padding our random vectors to meet all the conditions required for best performance possible when using tensor cores. In doing so however, as the random vector array is only 1-32 vectors long it means we may end up doubling the length. If for example we want to use 17 bits, in which case we would pad so that it is of length 32. This would approximately double the amount of operations needed to be done. We test by seeing if there is any difference in performance over 10 runs on on the ANN_SIFT1M data set with 450 iterations and 19 bits, with and with-out padding.

Results

Testing on the ANN_SIFT1M data set	
Padding	No padding
12.531182s	12.784333s

Table 5.13: Shows time in seconds used on the ANN_SIFT1M data set with and without padding for the random vectors, average over 10 runs

Reflections

The results will most likely vary depending on the number of iterations and bits and the data set, but we see that for 19 bits and 450 iterations on the ANN_SIFT1M data set padding gives us slightly better performance. If this is pure coincidence by the randomness of the vectors or if this is because of the padding is a bit hard to say as we only took the average over 10 runs. We can however be fairly certain that the padding does not lead to significantly worse performance even though it means we go from having to dot our query/reference vectors with 19 random vectors to dot with 32 for each iteration.

5.5.5 Concurrent streams

To avoid the overhead caused by doing some parts of the algorithm on CPU and to keep the GPU occupied at all times we use concurrent streams. To see how the number of concurrent streams we run affects our run-time we run our algorithm with 2 - 16 streams. We do this with 19 bits and 450 iterations and take the average over 10 runs for each number of concurrent streams.

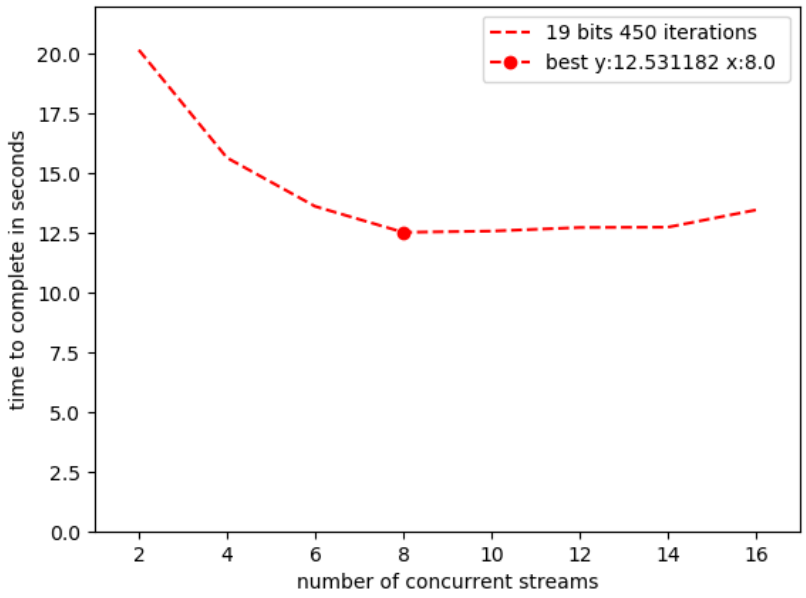

Figure 5.19: Shows how changing the number of concurrent streams affects performance on the ANN_SIFT1M data set

Reflections

The results are more or less what we expected them to be, we see an increase in performance until around 8 concurrent streams. This is most likely because at around 8 streams we will get the best occupancy we can, in other words increasing the number of streams any more will just make it harder for CUDA to schedule. Obviously the number of streams needed to achieve the best performance will be dependent on the problem size, desired recall, number of bits used and number of iterations we run the algorithm.

5.5.6 Time used by the different kernels and the overhead

To understand better where the bottleneck in our implementation is we look at the time used by the different parts of our algorithm. As we are running multiple concurrent streams, and as LSH is a somewhat random algorithm it is very hard to get 100% accurate measurements. The results will also change depending on our desired recall, we test for 0.8 recall @ 100 and 0.8 recall @ 1.

How we test

We run the algorithm 20 times and take the average time, we then remove one kernel and take the average over 20 runs again. We keep removing kernels and taking the average time until only CPU and data

movement/allocation is left. For this to work however the first kernel we need to remove is the brute force. To know approximately how much time each part used, we can now for each kernel take the difference from the total time without the kernel and the time with the kernel. While this will not give us the most accurate results possible it will give us a clear image of what parts are the most time consuming.

Results

Time in seconds used on the ANN_SIFT1M data set to achieve 0.8 recall		
	Recall @ 1	Recall @ 100
Total time used	12.728130s	1.897476s
Short-list brute-force	10.543828s, 0.82%	1.427292s , 0.752%
Thrust sort index array	1.435962s, 0.11%	0.286857s, 0.151%
Setting bits in hash value	0.242682s, 0.019%	0.04392s, 0.023%
Dot product with cuBLAS	0.207879s, 0.016%	0.037911s, 0.019%
CPU and data movement/allocation	0.297779s, 0.023%	0.101496s, 0.053%

Table 5.14: Shows approximately the time used by the different parts of the LSH algorithm

Reflections

We see that the by far most time consuming part of our algorithm is the short list brute-force, this it true for both when we look at recall @ 100 and @ 1, where the short list search uses respectively 75% and 82% of the total time. That the brute-force is the bottle neck is what you would expect and what most papers on LSH note. In addition the short list brute force we have written is not really suited that well for the ANN_SIFT1M data set. This is because it is optimized for the case where there are an equal number of query and reference points, however the ANN_SIFT1M data set has a 1 to 100 ratio between query and reference points. The second most time consuming part is the sorting of the index array by hash values. As we mentioned in the design 3.2.2 we had 2 obvious choices here, sort, or use a big empty array with linked lists. We chose to sort, if this was the best choice or not we are not sure as we only had time to implement with sort, we do however see that sorting is quite time consuming which may hint that using a different approach could be beneficial. We see that using cuBLAS to perform the dot product is very fast, even faster than actually setting the bits in the hash value.

5.5.7 Reflections LSH on GPU

Testing is very hard because of how we have implemented our solution. A kernel may work well on its own, but it needs to work well with the other kernels in the different steams. Writing a good short list brute-force search in CUDA for LSH is hard as we would have to know approximately what the data sizes would be, or write something extremely complicated. Testing

to find to optimal configurations i.e number of bits used for hash values, and number of iterations we run the algorithm, is a very time consuming process. LSH is a ANN algorithm and not a NN algorithm, while we can get good recall fast if we only look at recall @ 10 or recall @ 100 we will need to do many times the work to get good recall @ 1.

5.6 Comparison between the brute-force for SIFT vectors with cuBLAS, Naive CUDA brute-force and LSH on GPU

We look at how the different implementations compare on the ANN_SIFT1M data set. Where LSH gets a recall of around 0.8.

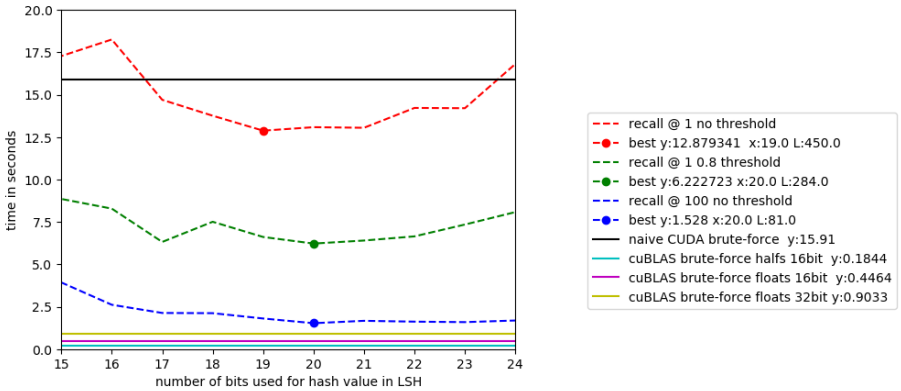

Figure 5.20: Shows time used for LSH on GPU, brute-force using cuBLAS and Naive CUDA brute force on the ANN_SIFT1M data set. Time is in seconds and is represented by the y value, the lower the better. The brute-force approaches are represented as straight lines, while LSH on GPU shows the best results at different number of bits used for hash value, at different levels of precision. The brute-force with cuBLAS bit stands for the level of precision used for the cuBLAS GEMM call, 16 or 32 bit

5.6.1 Reflections

We see that LSH does out perform CUDA naive brute force, even when we only count the NN as a true match, getting a 1.23X speed up. However it is 69.844X slower than the brute-force with cuBLAS. Even when we count any of the 100NNs as a true match it is 8.286X slower. On a much bigger data set LSH would in theory be able to outperform the brute-force with cuBLAS as the big O complexity of a matrix-matrix multiplication is much worse than that of LSH (if this is the case even for our implementations on the other hand we are not sure). For real-time and on a data set of this size however, we would most likely have to rethink our approach if we want LSH to outperform the brute-force with cuBLAS (using the CPU to do certain parts is to expensive because of the memory transfers). We see

that the brute-force with cuBLAS has a 86X speed up over the naive CUDA brute-force.

Chapter 6

Conclusion and Future work

In this chapter we sum up the answers to the research questions and note some reflections (section 6.1), we reflect around how efficient our research approach was (section 6.2), talk about some limitations we met (section 6.3), talk about some overall reflections (section 6.4), and lastly go over some possible future work (section 6.5).

6.1 Research questions

We had 4 main research questions we aimed to answer in this thesis. Here we go over how well we answered them and reflect on what we found.

6.1.1 LSH on GPU with CUDA

Question:

- Can locality sensitive hashing be efficiently implemented with CUDA to match SIFT vectors with good recall, and how will such an implementation compare to a brute-force CUDA implementation in terms of speed?

We have shown that implementing LSH with CUDA will indeed grant better results in terms of time used to match (when testing on the ANN_SIFT1M data set), compared to the naive CUDA brute-force we used as a benchmark, while also keeping around 0.8 recall as shown in figure 5.20. We get a speed up of 1.23X when looking at only the NN with no SIFT threshold, a 2.56X speed up when only looking at points which pass the 0.8 SIFT distance ratio threshold, and a 10.48X speed up when we accept any of the 100NNs as the true nearest neighbour.

Reflections

While we do get a speed up of 1.23X when looking for the NN, we also see quite clearly that there is a significant difference in performance when we look for the NN compared to any of the 100NNs, or when we only look for points which pass the SIFT distance ratio threshold. This strongly

suggests that LSH is much better suited to finding the ANN rather than the NN. The difference in the amount of work in terms of iterations of the LSH algorithm we need to do to find the NN vs finding one of the 10 NNs is well over doubled as seen in figure 5.18. However, even when looking for the ANN we do not come close to the brute-force which uses cuBLAS, and we believe that it is almost impossible without changing our approach on a data set of this size (we can not use the CPU in the way we have, because the transfers between host and device are simply too expensive). We also see that even on GPU the bottleneck of the LSH algorithm is the brute force, using over 75% of the time as shown in table 5.14. For LSH the brute-force seems to be where we can gain the most from further optimization.

6.1.2 Using Thrust and cuBLAS for LSH on GPU

Question:

- Can Thrust and cuBLAS be used when implementing LSH, and how will using these CUDA libraries affect performance and complexity?

We have shown that we can use both the Thrust and cuBLAS libraries when implementing LSH in CUDA. We use cuBLAS for calculating the dot products used for the hash values. We do this with a matrix - matrix multiplication with great success. We use Thrust to sort the index arrays used when indexing the query and reference points for the short list search. As we have seen that cuBLAS is very fast, it is safe to assume that using cuBLAS gives us an increase in performance. However, if this leads to less complexity depends on the desired performance. If we were to actually implement a CUDA kernel for the dot product, we could do so without too much complexity. However, we would not be able to get even close to the performance the cuBLAS library gives us, without making the kernel much more complex than using cuBLAS. So while using the cuBLAS library can be fairly complicated, it is absolutely worth using if performance is of importance. In the end we did not use the Thrust library as much as we expected. However, it is safe to assume that not having to implement a sorting algorithm significantly reduced the workload, as implementing a good sorting algorithm in CUDA is very complicated and time consuming.

Reflections

Using libraries in CUDA seems to be very beneficial in general, both in terms of performance and reducing the complexity of the implementations. This is especially true if we want to implement something that performs well on other NVIDIA GPUs as well.

6.1.3 Using `half`s over `float`s when matching SIFT vectors

Question:

- CUDA is optimized for half-precision (16 bit), is using `halfs` instead of `floats` when dealing with SIFT vectors viable, and how does using `halfs` when matching SIFT vectors affect recall and performance?

We have show that using `halfs` is indeed a viable choice. For the brute-force with cuBLAS it gave us a 5X speed up compared to `floats` with 32-bit calculations, and an approximate 2.4X speed up compared to `floats` with 16-bit calculations (see table 5.12), while only leading to a 0.0056 loss in recall on the ANN_SIFT1M data set (see table 5.4), and a 0.0554 loss in recall on a pseudo random data set of the same size as the ANN_SIFT1M data set (see table 5.10).

Reflections

When using `halfs` we do however need to be aware of what form the SIFT vectors are in, since if it is the case that the values are in the range 0 - 255 (if the norm is 512) we will have to divide by 512 to prevent overflow when using cuBLAS. Also one important thing to note here is that using `halfs` is more complex, exemplified by the fact that every `half` value has to be stored in a pair for best performance, and that we have to use the CUDA math API to perform any actions on the `half` values.

6.1.4 Brute-force 2NN SIFT matching algorithm with cuBLAS

Question:

- Can an efficient brute-force for SIFT vectors be implemented with the help of cuBLAS in CUDA, and will such an implementation be able to perform SIFT vector matching in real time?

We have shown that using cuBLAS to implement a very fast brute-force SIFT point matching algorithm is possible, giving us a 86X speed up on the ANN_SIFT1M data set compared to a naive CUDA brute-force. Using a RTX 3060 12GB, this implementation can match 10^4 Query vectors with 10^6 reference vectors in approximately 177ms, as shown in table 5.11. This implementation gets good (0.8771) recall on the ANN_SIFT1M data set, despite the data in the data set being far from ideal for our solution using the dot product. We have also shown that real-time SIFT feature point matching is possible at 45k query points and 45k reference points, where we can achieve 27 fps (0.036530s per frame) on a RTX 3060 12 GB.

Reflections

Since this is a brute-force it was fairly straightforward to implement. Most of the complexity came from using cuBLAS and batching. While it was not 100% accurate on the ANN_SIFT1M data set, this was because the data was both very compact (only 706 of 10000 query points had a match which passed the SIFT distance ratio threshold when at 0.8, as seen in 5.2), and the data was not normalized precisely enough. We believe that it would

get much better recall on a data set where every SIFT vector has the same norm, and probably even better recall if we only count points which pass the SIFT distance threshold.

6.2 Research method reflections

We chose to use the technocratic paradigm Eden [4] or design ACM [3], as our research methodology. We worked in iterations, implementing, testing, looking for probable improvements, implementing, testing and so on. In hindsight we believe this was the right approach, especially as optimizing in CUDA can also be an iterative process. When working in CUDA, the more things you consider (memory transfers, shared memory, occupancy, etc.), the more complex writing the programs becomes, however the performance you can achieve also increases. When implementing we started by considering the things we believed to be most important, such as memory transfers from global memory to the individual threads (coalesced reads), and occupancy. Towards the end of the process we started looking more into the use of CUDA streams and memory transfers between host and device.

6.3 Limitations

While we were able to achieve most of what we set out to do, time and complexity was a big limitation for our LSH implementation. While there were undoubtedly things we could have done to circumvent this it is also a part of what makes GPU programming so hard.

The problem with testing different versions of LSH

LSH has many variants, and knowing which of these variants works best on GPU is hard. To say with any certainty we would have to implement and test them, however with CUDA being as complex as it is, and LSH also being a fairly complex algorithm, the amount of work we need to do to test any other version suddenly increases by a lot, when we start to optimize for one version of LSH. As mentioned in our design, we aimed at implementing our LSH on GPU in a way that would allow us to test as many different LSH variants as possible. However, this turned out to be much more difficult than we initially believed. Because of this we were only able to test using different random vectors. In hindsight, we believe we should have started by implementing less optimized versions to test, ideally by using libraries instead of coding in CUDA ourselves. In that scenario, only after we have tested and found what works best, would we optimize it in CUDA. However this would still be a very time consuming process, and it is hard to say how much of a difference it would have made.

6.4 Conclusion

In this paper we have shown one way of implementing LSH on GPU with CUDA. This implementation achieves better performance in terms of speed over a naive CUDA brute-force, while also keeping recall above 0.8. This implementation makes use of `halfs`, and the CUDA libraries cuBLAS and Thrust. We have also shown how to use cuBALAS and `halfs` to implement a very fast brute-force for SIFT vector matching, which is 86X faster than a naive CUDA brute-force on the ANN_SIFT1M data set. We have show that using `half` values instead of floats when dealing with SIFT vectors in CUDA is viable, and that it leads to an increase in performance of up to 5X while only losing 0.005 recall when used in a brute-force implementation tested on real data. We show how cuBLAS works better on smaller queries, and that for LSH the bottleneck is still the brute-force even on the GPU. For real-time, brute-force with cuBLAS is probably the go-to solution, at least it is more suited to the task than LSH the way we implemented it. For very big data sets LSH could probably be better, but as the goal is real-time, cuBLAS brute-force is most likely the better solution. One if not the biggest downside with LSH on GPU is how hard it is to optimize, while a brute-force is fairly straightforward.

Optimization on GPU

When programming and optimizing on GPUs with CUDA, you often want to solve a problem in a way where you achieve the best performance possible, or close to it. If the problem is well defined this can be fairly straightforward. However, this is rarely the case. When optimizing in CUDA, things can quickly become very complicated if the problem is vague in any way, because of the abundance of options. To combat the complexity we often have to make assumptions or compromises. For instance in our brute-force with cuBLAS and LSH implementations we assumed that all SIFT vectors would fit in device memory, and in doing so we made the implementations more manageable, as we did not need to consider the out of core case. One of the more complex things we need to consider when using CUDA is where and when we need our data on device or host. One tool CUDA offers to solve this is unified memory with `cudaMallocManged`. However, while using an approach like `cudaMallocManged` may reduce the complexity, it can also reduce the performance (when using `cudaMallocManged` we can not be sure that we are getting asynchronous copies). Still, for most problems making assumptions and compromises is necessary. This is especially the case if we want to implement something which works well on multiple GPUs. Different GPUs and CPUs will mean different optimal optimizations making it virtually impossible to optimize for all cases. Meaning no matter how much you optimize for a certain GPU, there is no guarantee that it will work well on any other GPU. Using libraries like cuBLAS lets us to some degree circumvent this. This is because they often offer different implementations depending on the GPU, leading to less complexity while still having good

performance.

6.5 Future work

There are a few things which we believe to be worth looking at in more depth.

6.5.1 Brute-force for SIFT vector matching with cuBLAS

For the cuBLAS brute-force for SIFT point matching there are a few things we would have done if we had more time. For example, we could implement an out of core version capable of matching SIFT feature points even when they do not fit in device memory. This can be done with zero copy for memory transfers for the first batch, then `cudaMemcpyAsync` for the later batches. If all of the reference points fit in memory, a slight modification in how and where we store the output would suffice, however, how efficient this would be is hard to say and testing different ways of batching would probably be necessary. Another thing we would look at is how to automatically optimize the configurations, i.e the size of the array used for the output of the cuBLAS call. When testing we saw that optimizing this gave us a big increase in performance. However, optimizing this manually is time consuming. Since we have seen that this brute-force works efficiently and gives very good recall, trying to use it in real-time applications for measuring depth etc. seems like the logical next step.

6.5.2 LSH on GPU for SIFT vector matching

For LSH on GPU, time was a major limiting factor. There are many things we would have liked to test if we had had more time. For example using CUDA atomics to implement a linked list type hashing table, so we no longer need to sort. If this would be more efficient than our approach to sorting is hard to say, but it seems plausible as around 10% - 15% of the time used by the algorithm is spent sorting. For the LSH brute-force short list search we could program tensor cores (CUDA has an API for programming tensor cores) and use the dot product to measure distance (rather than the Euclidean distance), also using shared memory more would most likely give much better performance. There are many other versions and improvements of the LSH algorithm we would have liked to test on the GPU if we had more time, such as using lattices for the hash function and multi-probe hashing. Finding vectors which split the SIFT points better would also be worth looking into.

Bibliography

[1] Jon Louis Bentley. 'Multidimensional binary search trees used for associative searching'. In: *Communications of the ACM* 18 (9 Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007.

[2] Edgar Chávez et al. 'Searching in metric spaces'. In: *ACM Computing Surveys* 33 (3 Sept. 2001), pp. 273–321. ISSN: 0360-0300. DOI: 10.1145/502807.502808.

[3] D. E. Comer et al. 'Computing as a discipline'. In: *Communications of the ACM* 32 (1 Jan. 1989), pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239.

[4] Amnon H. Eden. 'Three Paradigms of Computer Science'. In: *Minds and Machines* 17 (2 Aug. 2007), pp. 135–167. ISSN: 0924-6495. DOI: 10.1007/s11023-007-9060-8.

[5] Vincent Garcia et al. 'K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching'. In: IEEE, Sept. 2010, pp. 3757–3760. ISBN: 978-1-4244-7992-4. DOI: 10.1109/ICIP.2010.5654017.

[6] Carsten Griwodz, Lilian Calvet and Pål Halvorsen. 'Popsift'. In: ACM, June 2018, pp. 415–420. ISBN: 9781450351928. DOI: 10.1145/3204949.3208136.

[7] Mark Harris. *Mixed-Precision Programming with CUDA 8*. 2016. URL: https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/ (visited on 25/05/2022).

[8] Piotr Indyk and Rajeev Motwani. ''Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality''. In: ACM Press, 1998, pp. 604–613. ISBN: 0897919629. DOI: 10.1145/276698.276876.

[9] H Jégou, M Douze and C Schmid. 'Product Quantization for Nearest Neighbor Search'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33 (1 Jan. 2011), pp. 117–128. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2010.57.

[10] Hervé Jégou et al. 'Query-Adaptative Locality Sensitive Hashing'. In: May 2008, pp. 825–828. DOI: 10.1109/ICASSP.2008.4517737.

[11] David G. Lowe. ''Distinctive Image Features from Scale-Invariant Keypoints''. In: *International Journal of Computer Vision* 60 (2 Nov. 2004), pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94.

[12] Qin Lv et al. 'Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search'. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 950–961. ISBN: 9781595936493.

[13] Marius Muja and David G Lowe. 'Fast approximate nearest neighbors with automatic algorithm configuration.' In: *VISAPP (1)* 2.331-340 (2009), p. 2.

[14] Edson Cavalcanti Neto et al. *Depth Calculation using Computer Vision and Sift*. 2010. DOI: 10.1007/978-90-481-9151-2_44.

[15] NVIDIA. *'cuBLAS Library - User Guide'*. Version DU-06702-001_v11.6. Jan. 2022. URL: https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf.

[16] NVIDIA. *'CUDA C++ Best Practices Guide - Design Guide'*. Version DG-05603-001_v11.6. Jan. 2022. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.

[17] NVIDIA. *'CUDA C++ Programming Guide - Design Guide'*. Version PG-02829-001_v11.6. Feb. 2022. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[18] NVIDIA. *'CUDA Math API'*. Jan. 2022. URL: https://docs.nvidia.com/cuda/pdf/CUDA_Math_API.pdf.

[19] NVIDIA. *'Thrust Quick Start Guide'*. Version DU-06716-001_v11.6. Mar. 2022. URL: https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf.

[20] Ives Rey Otero and Mauricio Delbracio. ''Anatomy of the SIFT Method''. In: *Image Processing On Line* 4 (Dec. 2014), pp. 370–396. ISSN: 2105-1232. DOI: 10.5201/ipol.2014.82.

[21] Malcolm Slaney and Michael Casey. 'Locality-sensitive hashing for finding nearest neighbors'. In: *IEEE Signal processing magazine* 25.2 (2008), p. 128.