Statistical Pattern Recognition
Final Project Report

Maximum Likelihood Approximate Nearest Neighbor Method in Real-time Face Recognition

Submitted in partial fulfillment of the degree

Bachelor of Science
in
Software Engineering

Submitted by
Ali Gholami

Under the guidance of
Prof. Mohammad Rahmati

Department of Computer Engineering and Information Technology
Amirkabir University of Technology

Spring Semester 2018

Abstract

Image recognition is one of the fundamental tasks in computer vision. Modern face recognition which is one of the substantial subtasks of the image recognition, in most cases, uses *brute-force* nearest neighbor method to iterate through the reference faces database to find the desired identity of the input image. Nowadays, most of these methods extract feature maps of the images using Convolutional Neural Networks. In this project, we'll be using Convolutional Neural Networks to extract image feature maps. We'll also analyze the implementation process of the *Approximate Nearest Neighbor* method to find the best ID for the input image.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Recent Methods of Image Recognition in Large Scale Databases 1
  - 1.2.1 Approximate Nearest Neighbor Shape Indexing . . . 1
  - 1.2.2 Directed Enumeration Method . . . 2
  - 1.2.3 Directed Enumeration Alternatives Modification . . . 2

2 Problem Definition **3**
  - 2.0.1 MLANN; General Idea . . . 3
  - 2.0.2 How Statistical Face Recognition Works . . . 3
  - 2.0.3 Where Problem Arises . . . 4
  - 2.0.4 Speeding up the Search Process . . . 4

3 Implementation and Performance Analysis **5**
- 3.1 Analysis of Naive Template Matching . . . 5
  - 3.1.1 Bitmap Images . . . 5
  - 3.1.2 Serial Implementation of NTM . . . 5
  - 3.1.3 Timing the Serial NTM . . . 6
  - 3.1.4 CUDA Implementation of NTM . . . 6
  - 3.1.5 Launch Parameters on Test Cases . . . 7
  - 3.1.6 Occupancy Analysis . . . 7
  - 3.1.7 SM Activity Analysis . . . 8
  - 3.1.8 Speedup Analysis . . . 9
  - 3.1.9 Performance Analysis . . . 10
- 3.2 Implementation of FFT Based Template Matching . . . 11
  - 3.2.1 Real to Complex Conversion . . . 12
  - 3.2.2 Padding Data . . . 12
  - 3.2.3 2-D FFT Planning . . . 12
  - 3.2.4 Forward FFT Launch . . . 12
  - 3.2.5 Complex Conjugate Kernel . . . 13
  - 3.2.6 Complex Point-wise Kernel . . . 13

3.2.7 Inverse FFT Launch 13
3.2.8 Maximum Value and Number of Occurrences 13

4 Future Work **14**

5 Conclusion **15**

References **16**

List of Figures

3.1 Illustration of varying the size of blocks, registers and shared memory on the achieved occupancy. 7
3.2 Issue efficiency and SM activity charts for the first (bottleneck) kernel. 8
3.3 Computation and data transfer time for naive serial and parallel implementation. 10
3.4 Calculation of first kernel operational intensity and GPU memory bandwidth. 11

Chapter 1

Introduction

1.1 Motivation

One of the well-known issues of vision systems building, is a processing of large databases. Unfortunately, nearest neighbor (NN) rule and exhaustive search usually cannot be implemented in real-time applications[1]. To overcome these problems, we'll analyze some of the purposed methods in this this topic.

1.2 Recent Methods of Image Recognition in Large Scale Databases

In this section we'll take a brief look at the recent researches on image recognition systems and their improvements for large scale databases.

1.2.1 Approximate Nearest Neighbor Shape Indexing

This method relies on the rapid recovery of the nearest neighbors from the index. In high-dimensional databases, standard k-d tree search often performs poorly. having to examine a large fraction of the points in the space to find the exact nearest neighbor. However, a variant of this search which efficiently finds approximate neighbors will be used to limit the search time. The algorithm, which we have called Best Bin First (BBF) search, finds the nearest neighbor for a large fraction of queries, and finds a very good neighbor the remaining times [2].

1.2.2 Directed Enumeration Method

Directed enumeration method is proposed to improve image recognition performance. The method is applied with similarity measures which do not met metric properties. This method increases performance in 3 to 12 times in comparison with nearest neighbor. Recognition speed using *DEM* is increased when many neighbors are located at similar distances [3].

1.2.3 Directed Enumeration Alternatives Modification

In this method, a new modification of the method of directed alternatives enumeration using the Kullback Leibler discrimination information is proposed for half-tone image recognition. Results of an experimental study in the problem of face images recognition with a large database are presented. It is shown that the proposed modification is characterized by increased speed of image recognition (5-10 times vs exhaustive search) [4].

Chapter 2

Problem Definition

In this chapter, we'll take a close look at the core algorithm of *MLANN* method. The next is dedicated to the implementation procedure of the algorithm.

2.0.1 MLANN; General Idea

Despite the most of known fast approximate NN algorithms, the proposed method is not heuristic. The joint probabilistic densities (likelihoods) of the distances to previously checked reference objects are estimated for each class. The next reference instance is selected from the class with the maximal likelihood. To deal with the quadratic memory requirement of this approach, the author has proposed its modification, which processes the distances from all instances to a small set of pivots chosen with the farthest-first traversal. Experimental study in face recognition with the histograms of oriented gradients and the deep neural network-based image features shows that the proposed method is much faster than the known approximate NN algorithms for medium databases [5].

2.0.2 How Statistical Face Recognition Works

In face recognition, we are required to assign an observed image X to one of R classes which is specified by the database of reference images. In this method, feature maps extracted from observed and reference images are treated as probability distributions. The following subsection provides baseline assumptions for this task.

Key Assumptions

In face recognition task with statistical method, we assume that

1. The reference images from different classes are independent;
2. The probability distributions of the feature vectors from the same class are identical [5].

2.0.3 Where the Problem Arises

The core algorithm of many face recognition implementations use *Nearest Neighbor* method as the main approach to find the most similarity between the input image and the reference images in the database. Let X be the observed image and X_r $r \in 1, ...R$ be each of the images in the reference database. In that case, the optimal maximal likelihood solution of the face recognition task is achieved by

$$W_v : v = arg \min_r \ \rho(X, X_r) \tag{2.1}$$

where r is selected in a *brute-force* manner by the nearest neighbor algorithm and ρ is mean to be the *distance* of two image feature maps.

2.0.4 Speeding up the Search Process

To speed-up the search process, we can use *approximate* techniques. As an example, an approximate method is provided in [6]. This method is based on the following criterion:

$$W_v : \rho(X, X_v) < \rho_0 \tag{2.2}$$

which is the termination condition of the *ANN* method with respect to a bound of of ρ_0.

Chapter 3

Implementation and Performance Analysis

3.1 Analysis of Naive Template Matching

In this section, we'll analyze the implementation and the performance of the *Naive Template Matching* algorithm. First of all, let's consider the serial implementation case.

3.1.1 Bitmap Images

For this project, we've used the *bitmap* image library by *Arash Partow* which is accessible here.

3.1.2 Serial Implementation of NTM

The serial implementation of this algorithm consists of 2 main loops to iterate over image pixels. There is also a need for each pixel we are iterating through, to calculate the *Sum of Absolute Deviations* which is the similarity measure between the main image pixels and the template image. Please refer to the *Naive Template Matching* algorithm to see the code. Since the task is mainly focused on counting the number of occurance inside the main images, we can obtain this by finding the minimum value of *SAD* while iterating through the main image pixels and counting the number of occurrences of the found minimum inside the main image matrix.

3.1.3 Timing the Serial NTM

We can obtain the elapsed time for the serial iteration using the *chrono* library. The code for this section is provided here. Please refer to the serial implementation code for the details.

3.1.4 CUDA Implementation of NTM

In this section we'll walk through the kernel codes provided in the *kernel.cu*.

Compute Sad Array Kernel

The image data is being stored in an 1-D array. Thus, the best choice for CUDA kernel launch parameters would be to set grid dimensions as $(image\_width/block\_size\_x, image\_height/block\_size\_y, 1)$. The main purpose of this choice is not the way we have stored the data, it actually depends on the input data. Since we are dealing with images it is more suitable to have 2-D blocks of threads. Further explanations focus on the kernel implementation. Initially, each thread finds its own global 2-D identification as rows and columns. Then, each thread virtually sees a kernel (template) around it self. This is the core of the parallelization section. All threads can see all of the iterations in one place. We then compute the SAD of each pixel and load it inside an SAD array. The next section describes the reason to do so.

Find Minimum in Array Kernel

We can obtain the best possible match by finding the minimum SAD in SAD array. We'll do so using shared memory to speed up the process. Each thread helps loading the data inside a block into the shared memory of that block. We then use *fminf* to find the minimum inside each block. Since we are using the shared memory, we need to do a *reduction* to find the minimum across different blocks. We can obtain this reduction task defining a *mutex* to control the global memory write by first thread of each block.

Find Number of Occurrences

Since the main task is to find the number of occurrences of template image in main image, we should count the number of occurrences of minimum SAD in the SAD array. To speed up the process we've used shared memory again. The intuition of using shared memory is exactly the same as the previous section.

3.1.5 Launch Parameters on Test Cases

The test has been done on three different sizes of images. One of the medium sized images is a *bitmap* image with dimensionality of $1112 * 1500$. The template image is a $116 * 116$ image in this case. Considering a block size of 1024 on a *Nvidia 850m GTX* (compute capability of 5), the launch parameters are provided below:

	X	Y	Z
Grid Dimensions	38	47	1
Block Dimensions	32	32	1

3.1.6 Occupancy Analysis

Before getting into the details, it is important to mention that maximum number of active warps in theory is 64. *Nsight* profiler provides great details on the number of active warps per SM. In this case, there are 63.83 active warps on the first kernel (Compute Sad Array Kernel). The occupancy can obtained by dividing the number of active warps to the number of maximum warps available:

$$occupancy = \frac{63.83}{64} = 99.74\% \tag{3.1}$$

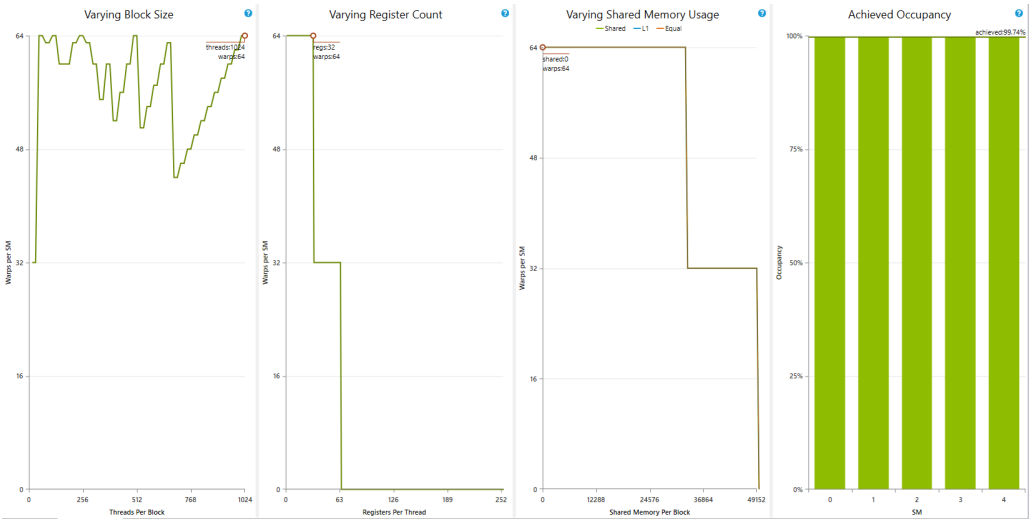

Figure 3.1: Illustration of varying the size of blocks, registers and shared memory on the achieved occupancy.

According to figure 3.1, varying the number of threads per block might yield the same result in other scenarios. The circled point shows the current number of threads per block and the current upper limit of active warps. Note that the number of active warps is not the number of warps per block (that is threads per block divided by warp size, rounded up). If the chart's line goes higher than the circle, changing the block size could increase occupancy without changing the other factors. Also, increasing the number of variables (registers per block) will drastically decrease the occupancy of the first kernel.

3.1.7 SM Activity Analysis

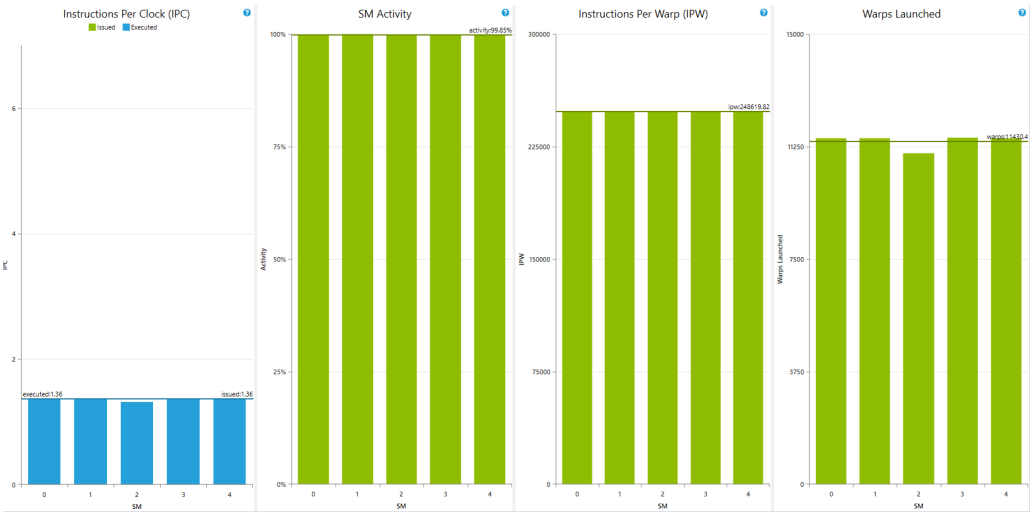

Figure 3.2: Issue efficiency and SM activity charts for the first (bottleneck) kernel.

The IPC chart demonstrates the achieved instructions throughputs per SM for both, issued instructions and executed instructions. The theoretical maximum peak IPC is a device limit and defined by the compute capabilities of the target device.

Issued IPC

The average number of issued instructions per cycle accounting for every iteration of instruction replays. Optimal if as close as possible to the Executed IPC. As described in the background section of this document, some assembly instructions require to be multi-issued. Hence, the instruction mix affects the definition of the optimal target for this metric. In this case, the issued

IPC throughput is minimum since there weren't any multi issue instructions in the first kernel.

Executed IPC

The average number of executed instructions per cycle. Each warp scheduler of a multiprocessor can execute instructions independently, a target goal of executing one instruction per cycle means executing on average with an IPC equal to the number of warp schedulers per SM. The maximum achievable target IPC for a kernel is dependent on the mixture of instructions executed.

SM Activity Chart

The second chart provides information on the activity of each SM during the kernel launch. A multiprocessor is considered to be active if at least one warp is currently assigned for execution. An SM can be inactive - even though the kernel grid is not yet completed - due to high workload imbalances. Such uneven balancing between the SMs can be caused by a few factors: Different execution times for the kernel blocks, variations between the number of scheduled blocks per SM, or a combination of the two. In this case, non of the above has occurred and the activity result is almost 100%.

IPW Chart

One of the most common patterns for varying IPWs is conditionally executed code blocks. IPW can be useful while having variations in SM activity chart which in this case, non of these have occurred.

3.1.8 Speedup Analysis

In this section, we'll analyze the possible theoretical speedup. For this purpose, we'll calculate the serial computation time for the *collection* and *collection_coin* images as described above. This table describes the average speedup of parallel naive template matching algorithm compared to serial implementation. As can be seen in the figure 3.3, the speedup obtained on GPU is dedicated to many different parameters. In order to calculate the theoretical speedup a direct use of *Amdahl's law* doesn't fit in this problem. One might say that the number of cores in this case is N (Same as the number of threads), but GPU does not have as many physical cores as the number of threads that can be launched. Additionally, significant portion of kernel time is begin spent just waiting for the data to be *read/written* from/to global memory. Also, the frequency and operational power of CPU

cores is higher than GPU cores. In this section, we'll use a modified version of the Amdahl's law [**?**]:

$$S = \frac{1}{(1-p) + (k * p/N)} \qquad k = freq(CPU)/freq(GPU) \tag{3.2}$$

The core frequency of my Intel 4710HQ processor is 3490 MHz. The core frequency of each GPU core is 902 MHz. These information are extracted using beloved *CPU-Z* and *GPU-Z* programs. Replacing these information into 3.2 yields

$$S = \frac{1}{(1-p) + 3.86 * p/N} \tag{3.3}$$

since the size of input is constant in this law, we'll assume that 3.3 is correct for the third experiment. Let's suppose that $\frac{1}{4}$ of the main program is parallelized, thus $p = \frac{1}{4}$ and the number of threads to run is 1828864 (1786 blocks of 1024). Theoretical speedup will be

$$S = \frac{1}{(\frac{3}{4}) + 3.86 * \frac{\frac{1}{4}}{1828864}} = 1.33 \tag{3.4}$$

	Exp 1	**Exp 2**	**Exp 3**	**Exp 4**
*Image * Template*	(630 * 459) * (116 * 116)	(1203 * 460) * (116 * 116)	(1211 * 1500) * (116 * 116)	(3840 * 2160) * (214 * 214)
Serial NTM	10787 msec	22615 msec	92204 msec	1722131 msec
Parallel NTM	339 msec	644 msec	1975 msec	27983 msec
Parallel NTM + Data Transfer	345 msec	665 msec	1980 msec	28977 msec
Achieved Speedup	***31.82***	***35.11***	***46.68***	***61.54***

Figure 3.3: Computation and data transfer time for naive serial and parallel implementation.

3.1.9 Performance Analysis

In this section, we'll analyze the performance of the *Compute Sad Array Kernel*. Firstly, lets find the arithmetic intensity of this kernel. Arithmetic intensity I, also called *operational intensity*, is the ratio of *arithmetic operations* or *work* (W), to the *memory traffic* (Q) [**?**]:

$$I = \frac{W}{Q} \tag{3.5}$$

and denotes the number of operations per byte of memory traffic. When the work is represented as *FLOPS*, the arithmetic intensity will be *FLOPS/Byte*.

According to the *Naive Roofline Model*, the *attainable performance* is bound either by the *peak performance* or *memory bandwidth * arithmetic intensity*. In this case, the operational intensity can be obtained by following the memory access patterns and the work being done in the main kernel: It is given

Statement	*Operations*	*# Global Memory Writes * bytes*	*# Global Memory Reads* bytes*	*# Constant Memory Reads* bytes*	*# Shared Memory Reads* bytes*
row, col	()()()(* + * +)	0	0	0	`6 * sizeof(int)`
1st conditional	()()(< <)()	0	0	`2 * sizeof(int)`	0
1st loop	()()(<)()	0	0	`1 * sizeof(int)`	0
2nd loop	()()(<)()	0	0	`1 * sizeof(int)`	0
m_r	()()(*)()+ + + * +	0	`1 * sizeof(unsigned char)`	`1 * sizeof(int)`	0
m_g	()()(*)()+ + + * +	0	`1 * sizeof(unsigned char)`	`1 * sizeof(int)`	0
m_b	()()(*)()+ + + * +	0	`1 * sizeof(unsigned char)`	`1 * sizeof(int)`	0
t_r	()()(*)() + * +	0	`1 * sizeof(unsigned char)`	`1 * sizeof(int)`	0
t_g	()()(*)() + * +	0	`1 * sizeof(unsigned char)`	`1 * sizeof(int)`	0
t_b	()()(*)() + * +	0	`1 * sizeof(unsigned char)`	`1 * sizeof(int)`	0
SAD Write conditional	()()(*<)() +	`1 * sizeof(int)`	0	`1 * sizeof(int)`	0
Final Operational Intensity	5 / (6 * sizeof(unsigned char) + 1 * sizeof(int) = 0.5				
Nvidia 850m GTX DDR Bus Frequency	1001 MHz				
Nvidia 850m GTX DDR Bus Bandwidth	2 * 128 = 256 bits				
Nvidia 850m GTX Memory Bandwidth	32 GB/s				

Figure 3.4: Calculation of first kernel operational intensity and GPU memory bandwidth.

that the *single precision* processing power of a *Nvidia 850m GTX* is 1155 GFLOPS [**?**]. Given these results, the kernel is actually memory bound since $0.5 < 1155/32 = 36$. The kernel arithmetic intensity should be at least 36 to be considered a compute bound kernel.

3.2 Implementation of FFT Based Template Matching

In this section, we delve into the details of *FFT* based template matching using *cufft* library from *Nvidia*. Note that this implementation is still in progress and there are multiple serial loops conducted in preprocessing step which slows down the process compared to the *naive template matching* procedure. As mentioned before, the overall procedure is to convert the spatial domain of input images to the frequency domain. In the frequency domain, we can apply a correlation based operation. This operation yields a signal of the same size as main image size. We then apply an inverse Fourier transform to turn the results back to the spatial domain. We then find the maximum of that signal and the number of occurrences of that maximum value in that

signal. The procedure is well defined below[**?**]:

$$c = real(IFFT_{2D}(FFT_{2D}(main\_image).*FFT_{2D}(template\_image))) \quad (3.6)$$

according to 3.6, c contains a list of values which we are desired to find the maximum of it (for maximum similarity). To obtain this in CUDA, we've used multiple functions which are described further. Note that * symbol shows the complex conjugate of the second signal.

3.2.1 Real to Complex Conversion

For this part, we have not focused on the performance. We use simple *for* loops to convert the image data matrix to the proper format of *cufft* library which is called *cufftComplex*. *cufftComplex* is a simple typedef of *float* type in C.

3.2.2 Padding Data

Before performing a *2-D Forward Fourier Transform* we are urged to perform a data padding on template signal to make it the same size as the main signal. Figure 2.1 illustrates this phenomenon pretty well. The duty of data padding is simple. Allocate new memory with a new size. Copy signals into their corresponding location in new allocated memory and set other places to 0.

3.2.3 2-D FFT Planning

cufft library uses the concept of *plan* to provide the baseline setup for the main APIs provided in the library. A plan can initiate a 1-D, 2-D or 3-D data processing API from *cufft* library. In this implementation, we need a setup for a 2 dimensional Fourier transform, thus we use *cufftPlan2d* to prepare the library for the main transformations.

3.2.4 Forward FFT Launch

In order to launch a forward *complex to complex* Fourier transform using *cufft*, we'll use *cufftExecC2C*. The constant *CUFFT_FORWARD* shows the direction of the Fourier transform which is *Forward* in this case.

3.2.5 Complex Conjugate Kernel

This kernel implements the conjugation procedure. The implementation is pretty straightforward. Each thread multiples the corresponding signal point's complex part to -1 and loads it back to the input signal.

3.2.6 Complex Point-wise Kernel

This kernel provides a point to point multiplication manner for each of the points in main and template input signals. It uses another kernel called *ComplexMul* to perform a *complex multiplication*. It then scales the result using *ComplexScale*. Scaling or normalization is mainly done by dividing the signal values to the size of the signal.

3.2.7 Inverse FFT Launch

Finally, we'll perform an inverse Fourier transform on the result of the *Complex Point-wise Kernel*. The resulting signal is in the spatial domain.

3.2.8 Maximum Value and Number of Occurrences

To find the number of occurrences, we *serially* iterate over the resulting signal of the previous section. Note that these steps are not *100%* efficient and are still in improvement.

Chapter 4

Future Work

We admit the possible drawbacks and inefficiencies existing in this project, thus we have the following key improvement to-do list:

- Reorganization of the project structure, header files and .c files.
- Reorganization of the project source code for a C unified implementation.
- Usage shared memory techniques to improve the performance of the *NTM* implementation.
- Usage of highly-optimized image libraries such as *OpenCV* for *image rotation*, *image matrix extraction* and general *bitmap processing*.
- Implementation of CUDA kernels for real-to-complex signal conversions.
- Implementation of CUDA kernel to find the number of occurrences of template signal in main signal in *Fourier* based template matching.

Chapter 5

Conclusion

In this project, we analyzed various algorithms for the task of *Image Template Matching*. It is fair to say that there are already highly optimized libraries with template matching capabilities such as *OpenCV*, but as mentioned before, the main purpose of this project is to improve the understanding of the CUDA environment and parallel implementation of highly-demanded computer vision tasks such as template matching. We analyzed two main methods to perform this task:

1. Naive Template Matching
2. FFT Based Template Matching

For further read, please refer to methods such as *Stereo Matching*, *Image Registration* and *Scale Invariant Feature Transform*.

References

[1] Tan, Xiaoyang, et al. Face Recognition from a Single Image per Person: A Survey. Pattern Recognition, vol. 39, no. 9, 2006, pp. 17251745.

[2] Beis, Jeffrey S., and David G. Lowe. "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces." cvpr. IEEE, 1997.

[3] Savchenko, Andrey V. "Real-time image recognition with the parallel directed enumeration method." International Conference on Computer Vision Systems. Springer, Berlin, Heidelberg, 2013.

[4] Savchenko, Andrey V. Image Recognition with a Large Database Using Method of Directed Enumeration Alternatives Modification. RSFDGrC11 Proceedings of the 13th International Conference on Rough Sets, Fuzzy Sets, Data Mining and Granular Computing, 2011, pp. 338341.

[5] Savchenko, A. V. "Maximum-likelihood approximate nearest neighbor method in real-time image recognition." Pattern Recognition 61 (2017): 459-469.

[6] Arya, Sunil, et al. "An optimal algorithm for approximate nearest neighbor searching fixed dimensions." Journal of the ACM (JACM) 45.6 (1998): 891-923.