Real-time Face and Landmark Localization for Eyeblink-Response Detection

Paul Bakker

CE-MS-2017-16

Delft University of Technology

Abstract

Pavlovian eyeblink conditioning is a powerful experiment used in the field of neuroscience to measure multiple aspects of how we learn in our daily life. To track the movement of the eyelid during an experiment, researchers traditionally made use of potentiometers or electromyography (EMG). More recently, the use of computer vision and image processing alleviated the need for these techniques, but currently employed methods require human intervention and are not fast enough to enable real-time processing. We selected a combination of face and landmark-detection algorithms in order to fully automate eyelid tracking, and accelerated them to make the first step towards an on-line implementation. Various different algorithms for face detection and landmark detection (eyelid detection) are analyzed and evaluated. Based on this analysis, two algorithms are identified as most suitable for our use case: the Histogram of Oriented Gradients (HOG) algorithm for face detection and Ensemble of Regression Trees (ERT) algorithm for landmark detection. These two algorithms are accelerated on GPU and CPU, achieving speedups of 1753× and 11.49×, respectively. A combination of these algorithms is successfully implemented for a real neuroscientific use-case: eyeblink response detection, achieving an overall application runtime of 0.533 ms per frame, which is 1101× faster than the sequential implementation. Furthermore, this accelerated implementation was used to generate a database of 1440 eyeblink responses during the conditioning experiment. We made use of multiple machine-learning techniques to analyze this database and concluded that there is a correlation between the asynchrony of the eyelids and the eyeblink-response performance during the conditioning experiment.

TU Delft Delft University of Technology

CE Lab Computer Engineering Laboratory

Real-time face and landmark localization for eyeblink-response detection

A heterogeneous CPU-GPU approach

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Paul Bakker
born in Tilburg, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Real-time face and landmark localization for eyeblink-response detection

by Paul Bakker

Abstract

Pavlovian eyeblink conditioning is a powerful experiment used in the field of neuroscience to measure multiple aspects of how we learn in our daily life. To track the movement of the eyelid during an experiment, researchers traditionally made use of potentiometers or electromyography (EMG). More recently, the use of computer vision and image processing alleviated the need for these techniques, but currently employed methods require human intervention and are not fast enough to enable real-time processing. We selected a combination of face and landmark-detection algorithms in order to fully automate eyelid tracking, and accelerated them to make the first step towards an on-line implementation. Various different algorithms for face detection and landmark detection (eyelid detection) are analyzed and evaluated. Based on this analysis, two algorithms are identified as most suitable for our use case: the Histogram of Oriented Gradients (HOG) algorithm for face detection and Ensemble of Regression Trees (ERT) algorithm for landmark detection. These two algorithms are accelerated on GPU and CPU, achieving speedups of 1753× and 11.49×, respectively. A combination of these algorithms is successfully implemented for a real neuroscientific use-case: eyeblink response detection, achieving an overall application runtime of 0.533 ms per frame, which is 1101× faster than the sequential implementation. Furthermore, this accelerated implementation was used to generate a database of 1440 eyeblink responses during the conditioning experiment. We made use of multiple machine-learning techniques to analyze this database and concluded that there is a correlation between the asynchrony of the eyelids and the eyeblink-response performance during the conditioning experiment.

Laboratory : Computer Engineering
Codenumber : CE-MS-2017-16

Committee Members :

Advisor:	Christos Strydis, Neuroscience, Erasmus MC
Chairperson:	Zaid Al-Ars, CE, TU Delft
Member:	Stephan Wong, CE, TU Delft
Member:	Christos Strydis, Neuroscience, Erasmus MC
Member:	Henk-Jan Boele, Neuroscience, Erasmus MC

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

AFLW Annotated Facial Landmarks in the Wild

API Application Programming Interface

AVX Advanced Vector Extensions

CNN Convolutional Neural Network

CPU Central Processing Unit

CR Conditioned Response

CS Conditioned Stimulus

DNN Deep Neural Network

DoG Difference of Gaussian

DPM Deformable Parts Model

EMG Electromyography

EOH Edge Orientation Histogram

FLOPS Floating-point Operations Per Second

FMA Fused Multiply-Add

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

HOG Histogram of Oriented Gradients

ICF Integral Channel Features

LBP Local Binary Pattern

LoG Laplacian of Gaussian

NN Neural Network

NVCC NVIDIA CUDA Compiler

NVVP NVIDIA Visual Profiler

OMP Open Multi-Processing

OpenCV Open Source Computer Vision

PCA Principal Component Analysis

PCIe Peripheral Component Interconnect Express

SDK Software Development Kit

SIFT Scale-Invariant Feature Transform

SIMD Singe Instruction Multiple Data

SM Streaming Multiprocessor

SMT Simultaneous Multithreading

SURF Speeded-Up Robust Features

SVR Support Vector Regression

UR Unconditioned Response

US Unconditioned Stimulus

Acknowledgements

First and foremost, I would like to thank my thesis advisor, Christos Strydis of the Neuroscience department of Erasmus MC, for his guidance during the project and for always keeping the office lively and full of laughter. Furthermore, I would like to thank Zaid Al-Ars, my supervisor at Delft University of Technology, for helping me in the search for an interesting thesis topic and his valuable comments on this work.

A big thank you also goes out to Henk-Jan Boele and Bas Koekkoek for the introduction into the fascinating world of neuroscience, George Smaragdos for his valuable comments on the thesis, Rene Miedema for his constant advice on every step of the implementation process, and everyone that participated in the eyeblink-conditioning experiments that were used for this thesis.

Finally, I would like to express my gratitude to my parents, family, friends and Nienke, for their support throughout this project. This accomplishment would not have been possible without them. Thank you.

Paul Bakker
Delft, The Netherlands
November 29, 2017

1 Introduction

Classical conditioning is the process in which a response is paired with a stimulus that would normally not elicit this response. One of the best known examples of this procedure is Pavlov's experiment, in which a dog is conditioned to salivate when hearing the sound of a a bell [10]. In this experiment, the sound of the bell, a conditioned stimulus (CS), is paired with an unconditioned stimulus (US), in this case food. The salivation in response to the US is called the unconditioned response (UR), as it normally happens without any conditioning. After a period of training, the dog would relate the sound of the bell (the CS) to the food (the US) and start to salivate at the sound of the bell. This response, which would not have occurred without the conditioning, is called the the conditioned response (CR). This relatively simple experiment has been studied in great extent to get a better understanding of the way we learn and memorize things.

The classical conditioning method that is studied in the Erasmus MC Neuroscience department is called eyeblink conditioning. In this experiment, the CS is a sound that is paired with a puff of air in the eye, the US. The natural response to this puff is to close the eyelid. The CS is presented several hundred milliseconds before the US. After a period of training, the subject is conditioned to close its eyelid when it hears the sound. The eyelid closure after the sound, but before the air puff is the CR. This experiment is visualized in Figure 1.1

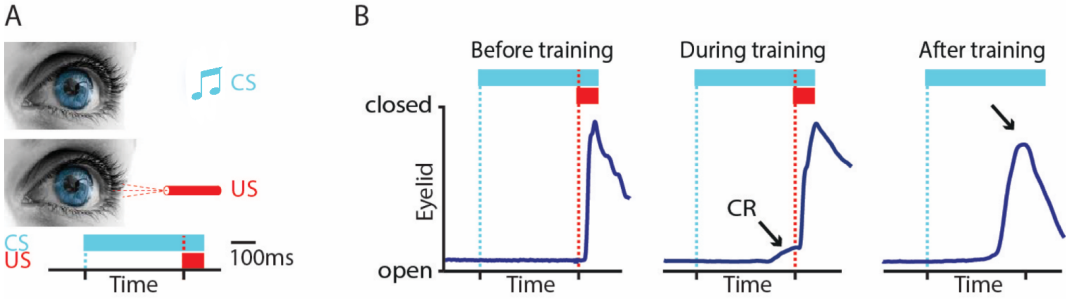


Figure 1.1: Visualization of eyeblink conditioning experiment. **A** The subject is presented with a sound as CS and a puff of air in the eye as US. The CS precedes the US by several hundred milliseconds. **B** Before training, the subject only shows a UR, as its eyelid only closes after the US. During training, the US is paired with the CS and the eyelid starts to close before the US, which is a CR. After training, the subject is only presented with the CS and shows a large CR.

To capture the eyeblink response, the subject is recorded with a high-speed camera. When the experiment is performed on mice, their eye is kept at a fixed position relative

to the camera. An algorithm is then applied to each image in the generated video (which only contains the eye), to determine the closure of the eye. Humans however, are not too keen on having their heads kept at a fixed position, and are moving relatively freely during the test. This requires a wider area to be recorded on video, and therefore an additional step to identify the region where the eye is situated in each image. At this point the process of eye localization in each image is slow and requires human intervention.

This thesis focuses on automating and accelerating the process of eyeblink-response detection from video in order to achieve real-time processing speed. This is the first step towards an on-line implementation, which alleviates the need for off-line video data storage and enables neuroscientists to adjust the conditioning experiment in real-time. Additionally, the automatic eyeblink-response detection algorithm is used to generate a database of eyeblink responses, on which a study is done on the relation between the asynchrony of the eyelids and the performance of the eyeblink response.

1.1 Motivation

While eyeblink conditioning was invented as early as 1922 [11], it is still one of the most studied forms of classical conditioning today [12]. Examples of recent studies that make use of this particular conditioning method are [12], in which is studied what happens in our brain when we learn new motor skills, [13], which studies the impact of several disorders from the autism spectrum on the CRs, and [14] which states that impaired learning of the CR indicates cerebellar abnormalities in schizophrenia.

At the Erasmus MC Neuroscience department, the eyeblink response is currently measured by analyzing the eyelid closure in a video, frame by frame. Each blink video is two seconds long, shot at 333 frames per second (FPS), resulting in a total of 666 frames per blink. This is done by manual selection of one half of the face in the first frame, followed by manual selection of the eye. Through a process of template matching [15], the eye region is cropped out for each subsequent frame and the eyelid closure is calculated. A typical frame of the eyeblink video can be seen in Figure 1.2.

Figure 1.2: Sample frame of a subject during eyeblink conditioning. The tube close to the left eye applies the puff of air (US).

The main drawback of this method is the manual intervention required: For every new trial, manual selection of the face half and eye is required. This also implicates the necessity of off-line storage, because the actual video needs to be seen and edited by a human before the eyeblink graph (such as the ones in Figure 1.1) can be extracted. Furthermore, the used template-matching procedure is not scale or rotation invariant, which results in dropped frames in the analysis when the subject in the video moves too much.

The automatic detection of face and eye at a processing speed that is able to keep up with the video frame rate would be the first step towards an online implementation of the eyeblink-response analysis. This would not only alleviate the need for human intervention and off-line storage, but it could also enable researchers to analyze the results of the subject in real-time and adjust the conditioning method based on the subject's performance.

This thesis aims to employ current technologies to serve the neuroscientific effort and is part of a larger collaboration between the Computer Engineering laboratory of the Delft University of Technology and the Neuroscience department of the Erasmus MC. Other topics of this collaboration include:

- Acceleration of large-scale brain simulations through heterogenous HPC technologies, powered by neuroscience-friendly simulation flows [16; 17];
- Real-time whisker tracking in rodents for studying sensorimotor disorders [18];
- Brain rescue and brain-machine interfaces [19; 20; 21; 22];

1.2 Thesis scope and contributions

In this thesis, a number of algorithms will be considered and analyzed to achieve the goal of automated high-speed eyeblink-response detection. Because the researchers from the Erasmus MC Neuroscience department expressed their interest in other parts of the face (apart from the eye) during eyeblink conditioning, detection shall be split into two distinct phases: detection of the face, followed by the detection of the eyelid closure. The selected algorithm, or combination of algorithms, should strike an appropriate trade-off between accuracy and speed. Once a decision has been made on which algorithm to use, a number of different approaches for acceleration of the algorithm shall be investigated in order to achieve the maximum processing speed.

1.2.1 Thesis goal

The main goal of the thesis can be formulated as follows:

"The selection and acceleration of a combination of algorithms to achieve automatic detection of eyelid closure in video data at real-time processing speed."

The first step towards achieving this goal is the selection of an appropriate combination of algorithms. As stated before, it is considered beneficial if not only the eye, but the whole face is detected, as other regions of it may contain information that is interesting to the neuroscientists. This implies a two-stage approach, of which the first one is the detection of the face. There is an important trade-off between speed and accuracy here, since more accurate and robust face-detection algorithms often have a greater computational load. The first subgoal of the project can, therefore, be formulated as follows:

1. Select the face-detection algorithm with the minimal computational load that meets the accuracy requirements for our setting; specifically:
   (a) Specify a set of requirements and constraints that define the faces that need to be detected;
   (b) Investigate a number of existing solutions for face detection;
   (c) Compare the face-detector accuracy and speed with the help of an annotated face database.

With the face-detection algorithm, we can extract the region containing the face from each frame. This region can then be used as input to the eyelid closure detection algorithm. The second subgoal of the project is formulated as follows:

2. Select the algorithm to calculate the level of eyelid closure in each frame of the video; specifically:
   (a) Investigate a number of existing solutions for eyelid closure detection;
   (b) Compare the accuracy and speed of the eyelid-closure detection algorithms.

In the current eyeblink conditioning setup, the camera records the blinks at a framerate of 333 Hz, but it can go up to a maximum of 750 Hz. The neuroscientists have expressed their interest to increase the camera framerate to 500 Hz. This means that the maximum processing time for each frame is 2 ms. It is very unlikely that algorithms chosen in subgoal one and two will satisfy this requirement out of the box. Therefore, we will need to accelerate the chosen algorithms in order to reach the required processing speed. In case the original algorithms are implemented in a high-level language (e.g. Python, MATLAB), we can convert them to the lower-level language C(++). This will not only provide an initial speedup but also make them suitable for possible hardware accelerations.

The acceleration of the face and blink detection is divided into two separate subgoals:

3. Accelerate the face-detection algorithm to reach, together with the eyelid closure detection, a maximum cumulative processing time of 2 ms; specifically:
   (a) (Possibly convert to lower-level language);
   (b) Profile the face-detection algorithm;
   (c) Investigate possibilities for (hardware) acceleration of the algorithm;
   (d) Implement best acceleration method.

4. Accelerate the eyelid closure detection algorithm to reach, together with the face detection, a maximum cumulative processing time of 2 ms; specifically:
   (a) (Possibly convert to lower-level language);
   (b) Profile the eyelid closure detection algorithm;
   (c) Investigate possibilities for (hardware) acceleration of the algorithm;
   (d) Implement best acceleration method.

The two accelerated algorithms will then be combined to achieve the final solution and subgoal:

5. Combine the two accelerated algorithms to achieve automatic detection of eyelid closure in video data at real-time processing speed.

1.2.2 Additional goal

The accelerated automatic detection of eyelid closure should allow for the fast generation of new eyeblink conditioning data. To showcase one of the interesting research possibilities with the newly acquired data, we will try to answer a hypothesis formulated

by one of the neuroscientists:

"The asynchrony of the eyelids of both eyes during the conditioned response is related to the performance of the eyeblink response."

Together with the neuroscientist, a quantitative measure is formulated to measure the performance of an eyeblink response during the conditioning experiment. Features will then be selected to describe the level of asynchrony of the eyelids during the conditioned response. These features will be used as input to a number of different machine-learning approaches that will try to find the best function (if any) to map the input features to a value for the eyeblink response performance. A baseline performance will be defined to compare the learned model to. The additional goal is formulated as follows:

1. Research if there is a relation between the asynchrony of the eyelids in the conditioned response, and the performance of the eyeblink response; specifically:
   (a) Define a quantitative measure of the performance of an eyeblink response;
   (b) Develop a baseline model that can be used as comparison;
   (c) Select features to describe the asynchrony of the eyes during the conditioned response;
   (d) Use the asynchrony features as input to a selection of machine learning methods and select the best-performing method;
   (e) Compare the performance of the baseline model with the model that makes use of the asynchrony features.

1.2.3 Thesis contributions

The following contributions were made by the work presented in this thesis:

- The performance of three different face detectors is compared on a database of 24384 face images. The Histogram of Oriented Gradients was selected as the algorithm that best suited the project requirements because of its combination of accuracy and speed.
- The face-detection algorithm was accelerated on a GPU, which reduced the detection time by a factor of 1753× to 333 μs.
- An Ensemble of Regression Trees is selected as landmark-detection algorithm, which is used to estimate the amount of closure of the eyelid.
- The landmark-detection algorithm was accelerated with OpenMP on a 16-threaded CPU. This reduced the detection time by a factor of 11.49× to 251 μs.
- The accelerated face and landmark-detection algorithms were combined and their execution was overlapped by making use of a pipeline model. The final implementation for eyeblink-response detection achieves a speed of 1876 FPS, which is more than the 500 FPS required for real-time processing.

- The accelerated implementation for eyeblink-response detection is used to generate a database of 1440 eyeblink responses during the conditioning experiment. This database was analyzed with multiple machine-learning regression techniques, and the conclusion is drawn that there is a relation between the asynchrony of the eyelids and the blink-response performance.

1.3 Thesis organization

The rest of the thesis will be organized as follows: Chapter 2 provides background information on the eyeblink conditioning experiment, object detection in computer vision, and parallel computing systems. In Chapter 3, different solutions for face detection and eyelid closure detection are compared and a decision is made on which algorithms shall be used in this project. Chapter 4 covers the details of the selected algorithms, while Chapter 5 discusses the implementation of their acceleration. In Chapter 6, the final implementation of the accelerated blink response detection solution is described and evaluated. The analysis of eyeblink conditioning data, to identify whether there is a relation between the asynchrony of the eyelids and the performance of the eyeblink response, is assessed in Chapter 7. Finally, the thesis is concluded in Chapter 8.

2 Background

In this chapter, background information that is needed for a proper understanding of the rest of the thesis is presented. In Section 2.1, we will describe the current setup of the human eyeblink conditioning system at the Erasmus MC Neuroscience department. Section 2.2 will cover the basics of object recognition and detection in computer vision. Finally, in Section 2.3, parallel computing will be discussed.

2.1 Human eyeblink conditioning

The eyeblink conditioning method is described in Chapter 1 and visualized in Figure 1.1. In the case of the human eyeblink conditioning performed at the Erasmus MC Neuroscience department, the subject is facing a camera that is positioned approximately one meter away from the subject. The camera captures the subject's face as well as some surroundings. This enables the subject to move a little without immediately going out of the camera's scope. The subject's attention is drawn towards the camera (e.g. with a monitor showing a movie), to minimize the movement and rotation of the face. Too much movement or rotation prevents the camera from getting a clear image of the eye and would therefore render the captured video data useless. These properties of the position and posture of the subject are of great importance to the selection of face detection algorithm for this problem. One can imagine that a face looks different from the side than it does from the front, and therefore an algorithm that needs to detect a face from one of these angles requires different properties than for the other one.

Other methods to record the eyeblink response are using a potentiometer coupled to the eyelid [23], or using electromyography (EMG) on the muscle that closes the eyelid [24]. However, the use of computer vision allows for a much easier and less invasive way of recording the response.

During eyeblink conditioning, each subject performs a number of trials in which the puff of air is the US which is preceded by a short high-pitched sound, the CS. Most subjects show a quick learning curve and start responding to the CS after a few trials to have their eye closed at the time the puff arrives. During training, the amplitude of the CR gradually increases [25], and the timing of the CR relative to the US improves (the onset of the CR is closer to the time of the US) [26].

2.2 Object recognition and detection

To record the blink response with the use of computer vision, without human intervention, we shall primarily turn to the field of object recognition and detection. Object recognition in computer vision is the ability to tell whether an image, or region of the

image, depicts a certain object (e.g. a bike) or not. In a multi-class approach, the goal of object recognition is to tell to which of the X discrete classes (e.g. bike, plain, car, etc.) the object in the picture belongs. The goal of object detection is to find the location, if any, of a certain object in the picture. A much used approach to achieve this is to scan an image in a sliding-window fashion at multiple scales, and classifying every scanned subregion as being an instance of the searched object or not.

This field has been receiving a lot of attention recently, primarily due to the advent of (Convolutional) Neural Networks ((C)NN), a technique that will be explained in Section 3.2.1.3. This is however, just one of many techniques for object recognition. Most of these techniques make use of features to describe an image followed by a classifier to determine whether the features describing that image belong to a certain object class or not.

2.2.1 Features

A feature can be described as a characteristic or attribute of an object [27]. For example, hair color and height could be two features to describe a person in our everyday life. The mean and standard deviation are two (statistical) features that describe a time-series. In object recognition, examples of features are edges, corners, gray values, histograms of colors etc. The amount of possible features is almost unlimited, and which features should be used is dependent on the object and the setting in which it needs to be recognized. Good features have the characteristics of being descriptive and discriminative; they must be able to consistently describe a similar pattern as the same feature, but also be able to distinguish between different patterns [28]. Calculating the features of a collection of data (in our case, an image) is called feature extraction.

2.2.2 Training and testing

During feature extraction, the image is described as a number (N) of features. This means that every image is converted into a point in an N-dimensional feature space. To correctly identify an object, first a classification model needs to be trained. For binary image classification, i.e. determining whether an image belongs to one of two classes X and Y, this means determining which region in the N dimensional feature space belongs to class X, and which region belong to class Y. This is depicted in Figure 2.1 for a 2-dimensional feature space and two classes. In our case, the class to which each point in the feature space belongs is known at training time. This is known as supervised learning. Once the model is trained, it can be used to predict the class of new data based on its location in the feature space (which is used as input for the model). This is called the testing, or inference phase.

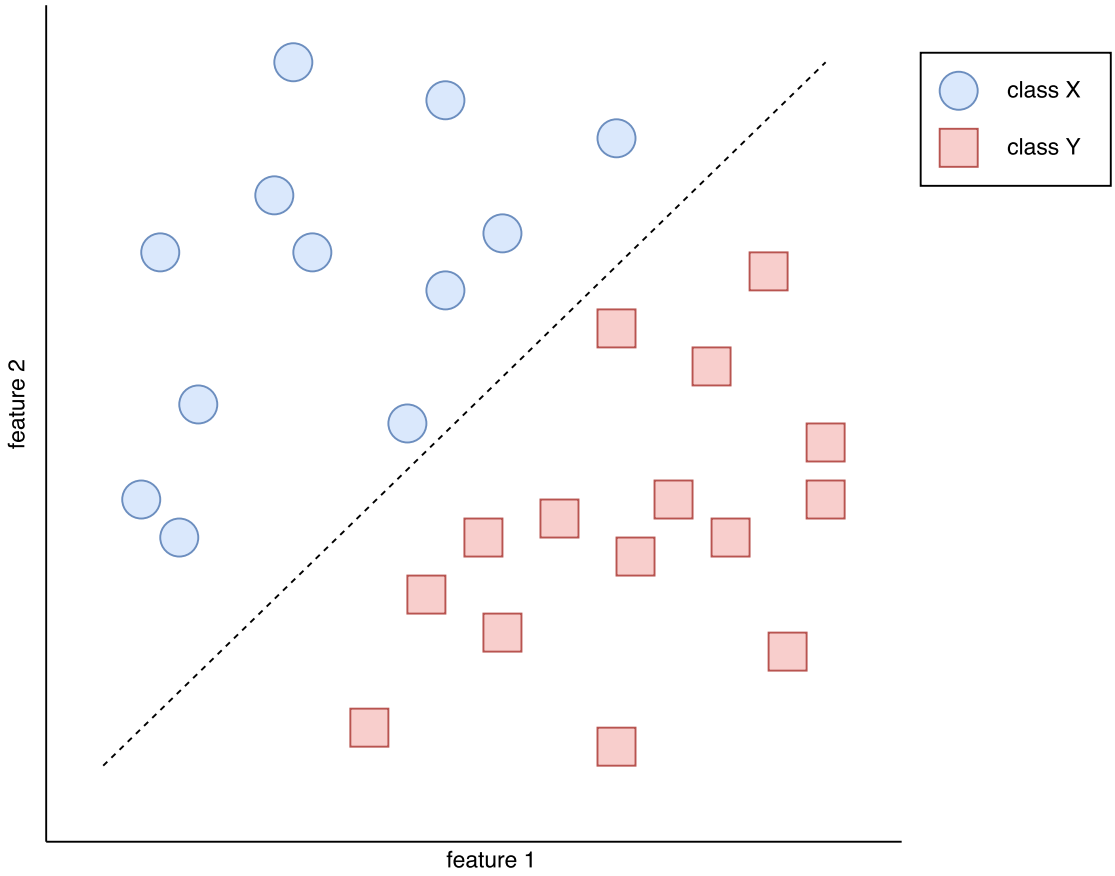


Figure 2.1: Example of a trained linear classification model in a 2-dimensional feature space. The squares and circles represent objects of the two different classes. The dotted line represents the division in the feature space and therefore what the model has learned. New samples will be classified as either a square or circle depending on their location relative to the division line.

In this project an image is converted into a set of 3100 features. These features are used as input for a binary classifier, that is able to tell whether they represent a face or not.

2.2.3 Object recognition difficulties

In Figure 2.2, several well-known difficulties in the field of object recognition are shown. While these are all easy for humans to overcome, each of these can pose a serious challenge for algorithms. Imagine for example, the impact of different illumination conditions on features that depend on the gray values of pixels. This means that certain features of the same object class can vary a great deal from image to image. For the selection of features for our problem, it is therefore important to consider the situation and condition in which the eyeblink videos are recorded. Good features should show little variation under these circumstances.

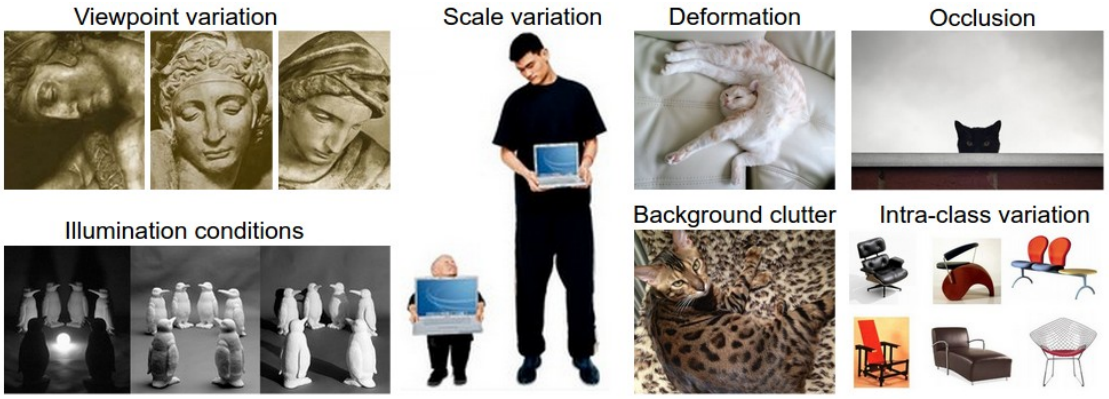


Figure 2.2: Common difficulties in object recognition that can lead to objects being falsely classified [1].

2.3 Parallel computing

Up until the year 2002, the processing power of desktop microprocessors increased by a factor of roughly 1.5 every year [1]. This advancement in speed was brought about by improvements in transistor technology, architecture and compilers. The frequency of microprocessors increased rapidly over the years, and the focus was on making a single processor as fast as possible. This came at a price, however, as the power consumption of these processors also increased rapidly. The progression of frequency and power of Intel microprocessors can be seen in Figure 2.3.

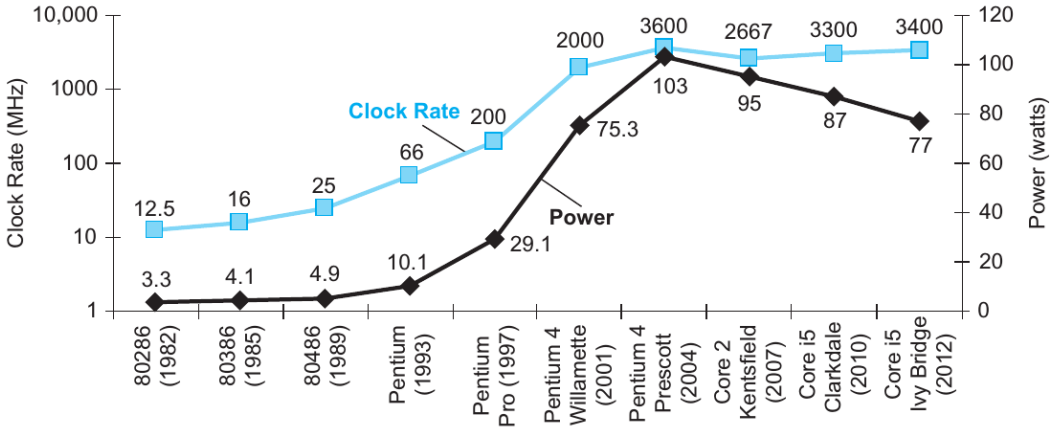


Figure 2.3: Increase in clock rate and power consumption in Intel processors in 25 years [1].

The reason for this coupled increase is that the two (frequency and power) are corre-

[1]Image taken from *http://cs231n.github.io/assets/challenges.jpeg*

lated. The power consumption of a single transistor in a microprocessor is given by the following equation:

$$Power \propto \frac{1}{2} \times Capacitive\ load \times Voltage^2 \times Frequency\ switched, \tag{2.1}$$

where *frequency switched* is a function of the clock rate. Up until recently engineers were able to limit the power increase by reducing the transistor voltage, but a further lowering of the voltage led to too much current leakage.

This problem is known as the **power wall**, and led to a switch of focus in computer engineering from single to multiprocessor systems. Instead of sequentially processing a number of tasks as fast as possible, multiply tasks can be processed in parallel by a number of processors, which leads to higher throughput. This switch is also visible in Figure 2.3, where after 2002 the frequency as well as the power stopped increasing.

Switching from single to multiprocessor systems brings challenges to the software developer, because programs that want to make use of multiple processors need to be written in a different way. Much as a group humans working together on the same task, it often requires:

- Scheduling - assigning work to a worker;
- Load balancing - making sure all workers are doing the same amount of work;
- Communication - communicating information to other workers that is required for them to complete their work;
- Synchronization - Making sure tasks have been completed that are required for the continuation of other tasks.

Central and Graphics Processing Units

CPUs, or Central Processing Units, are built as general-purpose processors. Over the years, several features were added to tackle a wide variety of computing problems as fast as possible. This led to complex and costly devices that consume a large amount of power. Therefore, in current consumer CPUs, only a few (typically two or four) microprocessors are put on a single die. This means that the amount of parallelism they can exploit is limited. Tasks where more data parallelism can be exploited, such as many image processing tasks, can benefit from the use of a Graphics Processing Unit (GPU).

While each individual processor in a GPU is less complex and less powerful than those in a CPU, modern GPUs can consist of several thousands of cores, making them very well-suited to accelerate programs with large amounts of data parallelism. Figure 2.4 shows a schematic overview of the difference between a CPU and GPU, in which can be seen that GPUs reserve a much larger area on the chip for data processing.

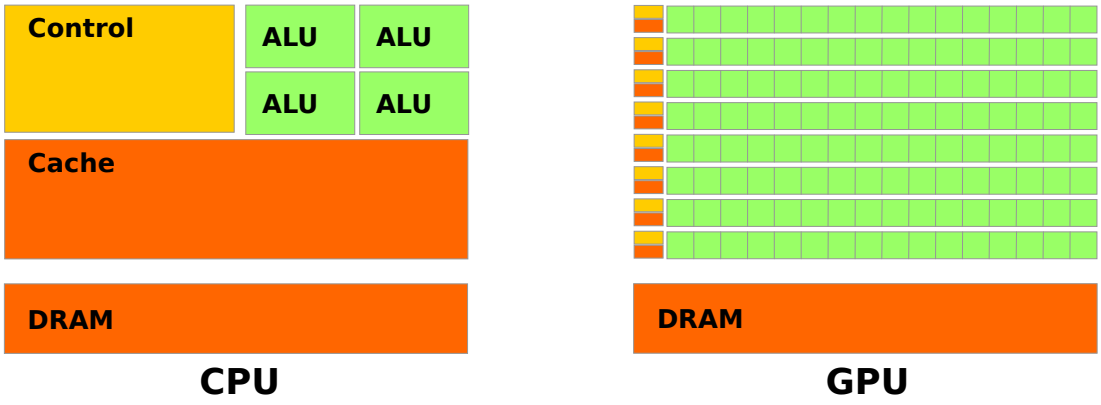


Figure 2.4: Schematic comparison between the chip layout of a CPU and GPU [2].

Data-parallelism on a GPU is achieved by dividing a problem into parallelizable pieces that each get processed by a thread. Threads are grouped into thread blocks, and the total number of thread blocks make up the grid (see Figure 2.5).

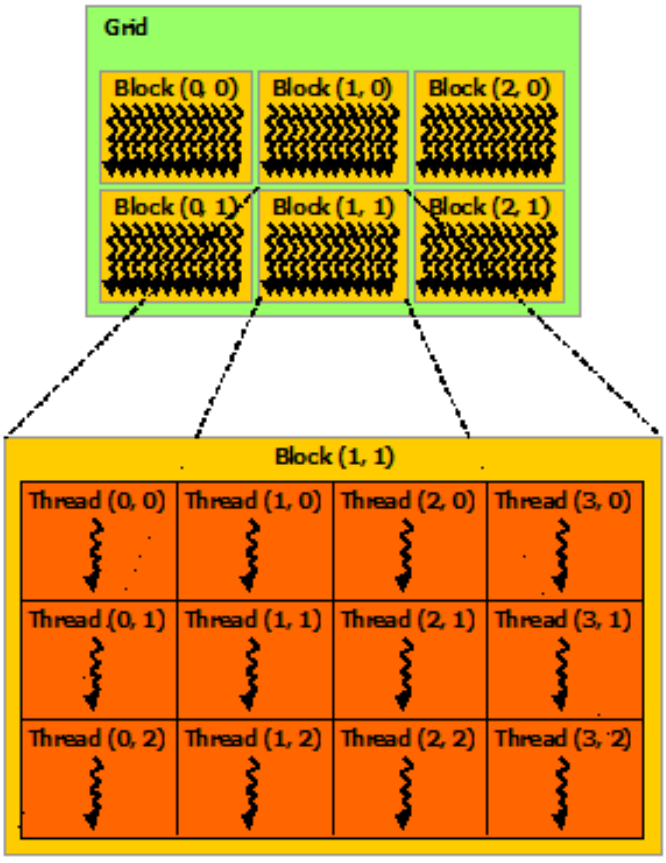


Figure 2.5: The grid consists of a number of thread blocks, and each thread blocks consists of a number of threads [2].

During execution, a thread block gets assigned to a Streaming Multiprocessor (SM), which contains a multitude of processing elements. Thread instructions are then executed in groups of 32, also called warps.

Image-processing algorithms are often well-suited for GPU acceleration, because operation are performed on pixels or groups of pixels that are independent from eachother. In this work, many of the feature extraction steps for face detection are performed on independent (groups of) pixels. In addition, the sliding window approach for face detection requires many image regions to be classified, which is also a process ideally suited for data-parallelism.

Lastly, an important note about GPUs is that they are latency-hiding machines. Instead of focusing on low latency operations, their power lies within hiding the latencies with thread and instruction level parallelism [29] and achieving high throughput.

3 Algorithm selection

The choice of algorithms is of paramount importance to this project. Not only do we need to make sure that the chosen algorithm is always able to detect the eyeblink response in our recording setting, but there is also a significant difference in detection speed between different alternatives. In this chapter, alternatives for both the face and eyelid-closure detection shall be investigated.

3.1 Requirements for eyeblink-response recording algorithms

In Section 2.1, the conditions in which the eyeblink-response videos are recorded are described. Together with the neuroscientists from the Erasmus MC Neuroscience department, a set of requirements was drafted regarding the location and posture of the subject, and the recording environment.

Rotation in the three spatial dimensions is defined as yaw, roll and pitch as can be seen in Figure 3.1. Since the subject is focused on the screen that is in the same direction as the camera, these are limited to $\pm 20°$, $\pm 25°$ and $\pm 40°$, respectively. The faces are only occluded when the subject is wearing glasses. This is a tricky requirement, because while the face may still be detected, when it is rotated such that the frame of the glasses partially occludes the eyes, the blink response can not be recorded. The setting in which the videos are recorded must be well-lit, but the high-speed camera captures the light flicker, which causes illumination differences among subsequent frames in the video. Furthermore, a subject must sit close enough to the camera for their face to cover at least 20% of the image. Finally, at this moment the camera records at 333 FPS, but the desire has been expressed to increase this to 500 FPS. Therefore, the required frame rate shall be set at 500 FPS. The requirements are summarized in Table 3.1.

Table 3.1: Conditions under which eyeblink response recording must work

Yaw	**Roll**	**Pitch**	**Occlusion**	**Lighting**	**Face size**	**Frame rate**
$\pm 20°$	$\pm 25°$	$\pm 40°$	Glasses	Setting is well-lit, but videos suffer from light flicker	20%+ of image	500 FPS

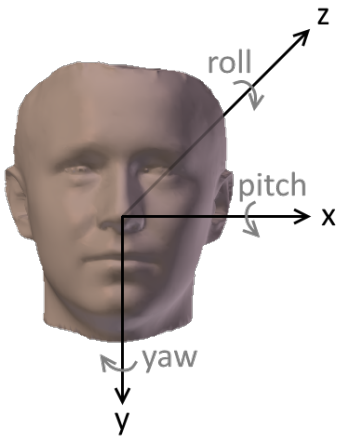


Figure 3.1: Rotation in each of the three spatial dimensions [1].

3.2 Face-detection algorithms

Face detection has been one of the most studied subjects in computer science and computer vision for over 15 years. The number of applications that makes use of this technology, as one might expect, is extremely large. The subject can, therefore, be considered as a mature one in the field, with a large variety of well-working alternatives to use. Existing techniques can be divided into two categories: so-called rigid templates and deformable-parts models (DPMs) [30]. Rigid templates learn to model the face as a whole, while DPMs model a face by describing it as sum of its parts.

3.2.1 Rigid-template models

Rigid-template models generally consist of two main parts. The first part consists of selecting a set of features to describe the image with (feature extraction), while the second part consists of training a classifier with these features. These rigid-template models can further be divided into two categories. Earlier models, such as the popular Haar-cascade face detector by Viola and Jones [31], make use of handcrafted features. Expert knowledge is required to determine which features to extract from an image to make it easily classifiable as a face or not. Newer models use Deep Neural Networks (DNN) or Convolutional Neural Networks (CNN), which are a type of NN specialized in image recognition. During the training process, these networks learn the important features in the images themselves, omitting the need for expert feature knowledge.

[1] Image taken from *https://skybiometry.com/faq-2/*

3.2.1.1 Handcrafted features

In 2000, Viola and Jones [31] created one of the first face detectors that was able to run in real-time (30+ FPS), which is still very popular. Several contributions of their paper are still used in many of the more recent face detectors, which is why this detection method is described in detail here.

The features they extracted from an image were *Haar rectangles*: the difference in cumulative pixel intensities of two to four different rectangular regions of the image; see Figure 3.2. Even for small images, the amount of subregions per image to calculate is very large, resulting in a vast amount of Haar features. To calculate all these features faster they made use of the integral image [32], which is a table stating the cumulative pixel gray intensities in the rectangular area from the top left corner to each of the points in the image. With the help of this integral-image table, calculating the cumulative gray intensity of an arbitrary rectangle in the image is reduced to adding and subtracting four elements in this table.

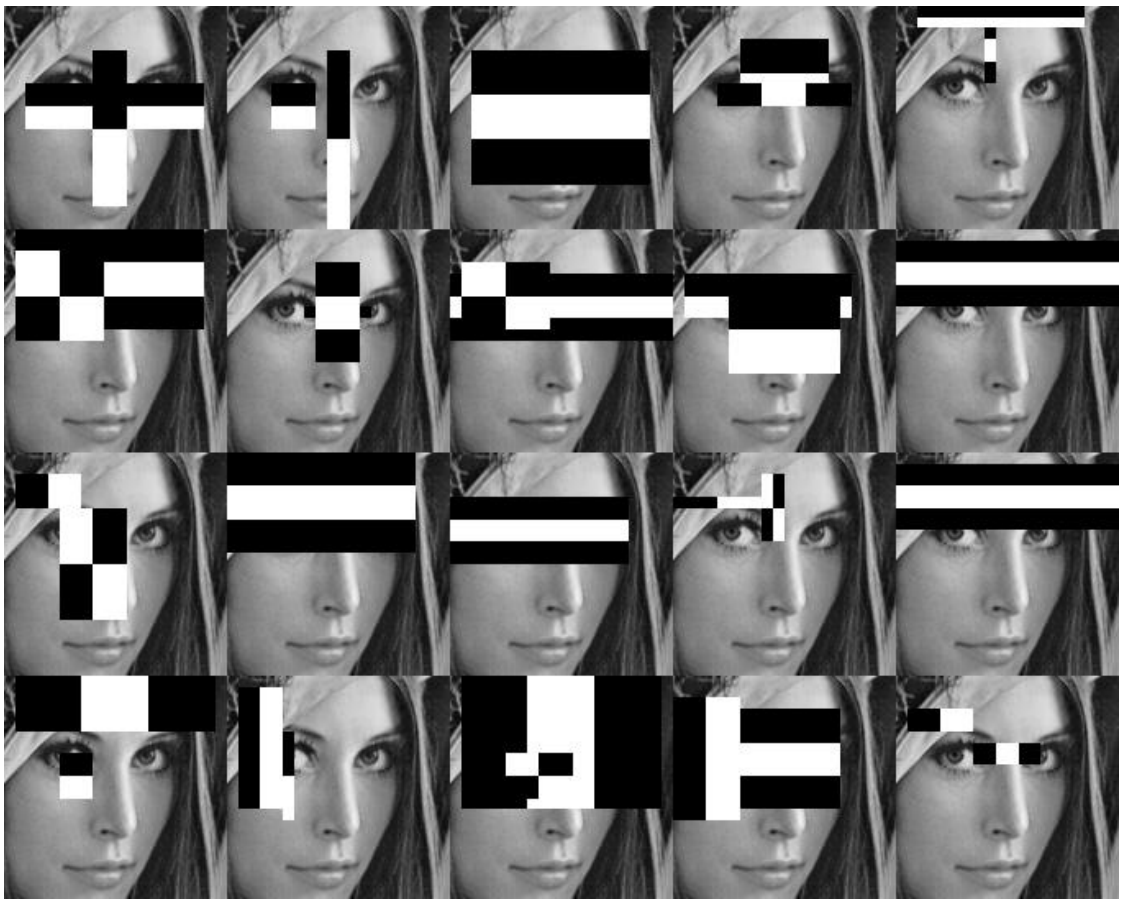


Figure 3.2: Examples of the many different Haar features that can be computed from the image of a face [2].

[2]Image taken from *https://memememememememe.me/post/training-haar-cascades/*

Viola and Jones were able to find the right combination of Haar features to classify an image as a face by using Adaboost [33], one of the first practical boosting algorithms. Boosting is the process of finding the right combination of several weak classifiers to create one accurate classifier [34]. Each of the Haar features by itself is not descriptive enough for reliable face classification, but by finding the right subset of these features, we are able to create a much stronger classifier.

Another important contribution of the Viola-Jones paper was the use of a cascaded classifier structure. This means that all image regions are first subjected to a smaller boosted classifier (one containing a smaller number of weak classifiers). These smaller classifiers are relatively computationally inexpensive, but able to filter out many of the image subregions already. This process continues for several steps: increasing classifier size and complexity on a smaller and smaller pool of image regions. Using this cascaded structure, the computationally expensive process of analyzing each image subregion with the most complex and expensive classifier is avoided.

Haar features are simple and effective features for face detection and above all, they are fast to compute. In the next section, we shall discuss more complex feature alternatives.

3.2.1.2 Feature selection

As stated in Section 1.2.1, the choice of the right features for this project boils down to strike the right balance between speed and accuracy. In this section, a summary of possible features with a short description is provided. Note that there are too many different features to discuss in the scope of this project. Therefore, the number of features discussed here is reduced to the ones that have made a monumental impact on the field of object and/or face detection, based on the number of citations of their respective papers; as with the Viola-Jones paper [31], this is in the range of 10.000 and more. For a more detailed survey, the reader can refer to the survey in [30] and the many papers that the survey points to.

Local Binary Pattern (LBP) [35] (9291 citations) - Divides the image into cells. Compares each pixel in the cell with its 8 neighbors and, based on this, creates an 8-bit number. For each cell, a histogram is created based on the occurrence of each 8-bit number. Finally, the histograms of all cells are concatenated.

Histogram of Oriented Gradients (HOG) [36; 37; 38; 39; 40] (18744 citations) - Divides the image into cells. The gradient magnitude and angle are computed for each pixel within the cell. Per cell, the results of each pixel are put into a histogram of 9 or 18 bins corresponding to different angle regions. The concatenated histograms are the feature vector fed to a classifier. **Edge-Orientation Histogram** (EOH) [41] is a method that is very similar to HOG. To each pixel, 5 Sobel masks [42] are applied to determine in which of five directions the gradient magnitude is the strongest. Histograms are computed for different (local) regions and used as input to the classifier.

Scale-Invariant Feature Transform (SIFT) [43] (41460 citations) - Uses Difference of Gaussians (DoG), an approximation of the more costly Laplacian of Gaussian (LoG) at multiple values of scaling parameter σ to detect scale-invariant keypoints in the image. Low-contrast and edge keypoints are filtered out to keep only the most important ones. Of a square region around these keypoints, a gradient histogram much like in the HOG algorithm is constructed. The histogram is shifted according to the orientation of the keypoint, to make the keypoints rotation-invariant. These histograms are the final features to be classified. According to [44], "SIFT and HOG are blockwise orientation histograms, a representation we could associate roughly with complex cells in V1, the first cortical area in the primate visual pathway".

Speeded-Up Robust Features (SURF) [45] (8388 citations) - Developed in response to SIFT, makes use of box filters to approximate the LoG for keypoint localization, and Haar wavelets [46] for the keypoint orientation. The advantage of these methods is that they are both able to make use of the integral image method. The final features are horizontal and vertical Wavelet responses in a region around the keypoints. According to [47], SURF accuracy is comparable with the SIFT method, while achieving a speed that is three times faster. SURF features cope well with in-plane rotation and blurred images, but have difficulty with viewpoint and illumination change.

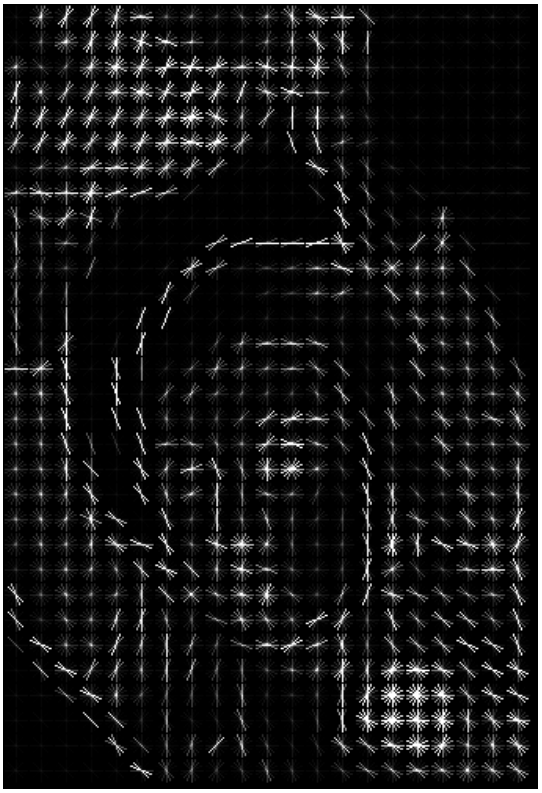
LBP, HOG, SIFT and SURF features are visualized on the same original image in Figure 3.3.

Many of these feature classics have also had more recent upgrades to alleviate their shortcomings. While LBP, EOH and HOG are not scale-invariant, the images they run on can be scaled up and down in a pyramid-like fashion, rerunning the complete algorithm on each scale. If the algorithm is not tolerant to rotations in the 3D plane, multiple classifiers can be trained for each rotation angle. The features of an image are fed into each of these classifiers, which results in higher classification accuracy at the cost of higher computation time. For example, this has been done for HOG in [48] for a total of 22 different models which correspond to different rotations of the head. In [49], Haar features are also computed for rectangles that are diagonal in the image, making the face detector more invariant to in-plane rotations compared to the original Haar feature face detectors.

Another common way to improve the accuracy of the face detection algorithms is to combine a number of different features. This way, the strong characteristics of each feature set can be combined. One of these approaches is called Integral Channel Features (ICF)[50], where "Integral" refers to the integral image method the authors use, and the "Channels" represent the different feature sets (grayscale intensities, gradient magnitude and color information). In [51], histograms are collected after applying a number of different filters to the image: gradient, LoG and Gabor.

Other feature sets specifically focus on shape information [52; 53], or simply compare every pixel in the image[54] for features even simpler than the Haar ones.

(a) Effect of Local Binary Pattern filter on image.



(b) Histogram of Oriented Gradients features per image cell.

(c) Keypoints detected by SIFT algorithm.

(d) Keypoints detected by SURF algorithm.

Figure 3.3: Features extraced by different algorithms on the same image.

3.2.1.3 Automatic features

As seen in Section 3.2.1.2, the choice of which features to use is not a trivial one. Recently, Neural Networks (NN) and their deep counterparts (DNN) have received renewed interest. This is partly thanks to the use of general-purpose computing on Graphics Processing Units (GPGPU) for the training phase of these networks; a very computationally intensive task. Convolutional Neural Networks (CNN) [55] are networks whose architectures are specifically designed to classify images. These networks consist of multiple subsequent layers that all apply their own convolutional filters to the image. This can be seen in Figure 3.4.

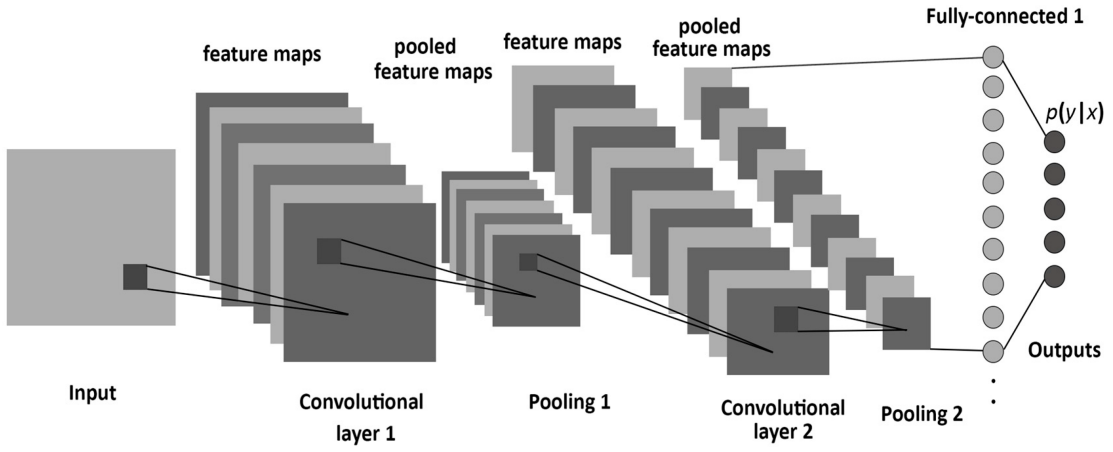


Figure 3.4: Example of the architecture of a convolutional neural network. The convolutional layers apply a number of convolutional filters to the feature map of the previous layer. Pooling layers reduce the size of the feature map by taking the maximum or average of a feature map region. The features in the 2-dimensional feature maps are concatenated and weighed in the fully-connected layer, of which the output is used for the final classification [3].

In the training phase, these networks are shown a lot of images with their corresponding labels, i.e. the object the image depicts. From this process they learn automatically which features are important to describe a certain object, or which features are important to distinguish one class of objects from another. Earlier layers are known to capture low-level features of the object such as corners and edges, while later layers capture more high level features. According to [56], the features extracted by the first layer of a CNN are similar to the SIFT features described in Section 3.2.1.2.(D)CNNs are considered the state-of-the-art in many object classification tasks [55] and can be trained to cope well with 3D rotations of an object, irregular illumination and occlusion as long as the set of training images contains enough of these examples.

While CNNs are very good at image classification, i.e. identifying what kind of object an image depicts, object detection is a somewhat harder problem. Like many of the handcrafted feature methods, the usual approach to tackle this, is to use a sliding-window on multiple up and downscaled versions of the image [57; 58]. Each subregion of

the image is scaled to the network's input size and tested with the trained network. For our project, the network would output whether the inputted subregion of the image is either a face or a non-face. While this is viable for the less compute intensive handcrafted features, (D)CNNs are much more computationally intensive. Testing every subregion of the image on multiple scales is therefore a very time consuming task.

Therefore, other approaches have tried a two-step technique where a more general object detector or image segmentation is the first step of the process [44]. This first "module", such as selective search [59], scans the image for possible objects of any class and is computationally less intensive. The possible face-objects are then used as input for the CNN. This reduces the number of subregions that have to be tested with the CNN but still leaves more than 2.000 regions [57], resulting in a computationally suboptimal task that in some way resembles the cascaded structure of the Haar-cascade classifier in Section 3.2.1.1.

In [60], the authors tackle the task of object detection as a regression problem instead of a classification problem. The networks tries to predict a "binary mask" on the inputted image, where '1' and '0' correspond to a pixel lying within, or outside a bounding box region of the object, respectively. The downside the authors state, is that using this approach, a huge amount of training data is needed because the network needs to be trained with objects of all sizes on almost every location.

Lately, interest has been given to decreasing the inference time and model size of Neural Networks, also known as model compression. Squeezenet [61] is one of these models, where the authors achieve the same accuracy as the popular image classification network AlexNet [55], with 50 times less parameters. Techniques like pruning [62], deleting irrelevant parts of the network, or the binarization of weights [63], are used to decrease the size and number of parameters in the network without suffering (too much) accuracy. While most of the CNN face detectors will still require the network to test a large amount of subregions per image, reducing the inference time in this way could be a investigated to see whether the use of CNNs could be a viable option for high-speed face detection.

3.2.2 Deformable parts model

The deformable parts model for object detection is described in [4]. The idea of the model is that any object consists of multiple parts that are connected to each other in a certain geometrical configuration. First, a sliding-window "root" filter scans the image on multiple scales on a coarser level for whole object candidates. The features that are extracted and classified to determine this, are HOG features in [4], but other features can be used as well. An area around a detected object is scanned again, now for the individual parts/components that make up the object. This is done at a resolution twice as high as the original root filter for greater accuracy. For a face object, this would mean that the root filter captures the edges that determine the boundary of the whole face, while the part filters try to determine the location of e.g. the eyes, mouth and nose. This can be seen in Figure 3.5. The final score for an object detection is based on the

detection certainty of the individual parts, minus a "deformation score": their position relative to the other parts of the object compared to their expected normal position. While this method copes well with slight variations between objects of the same class, multiple models are often trained to improve performance for object rotations. In [48], the authors achieve state-of-the-art performance by using a total of 6 DPM models for face detection.

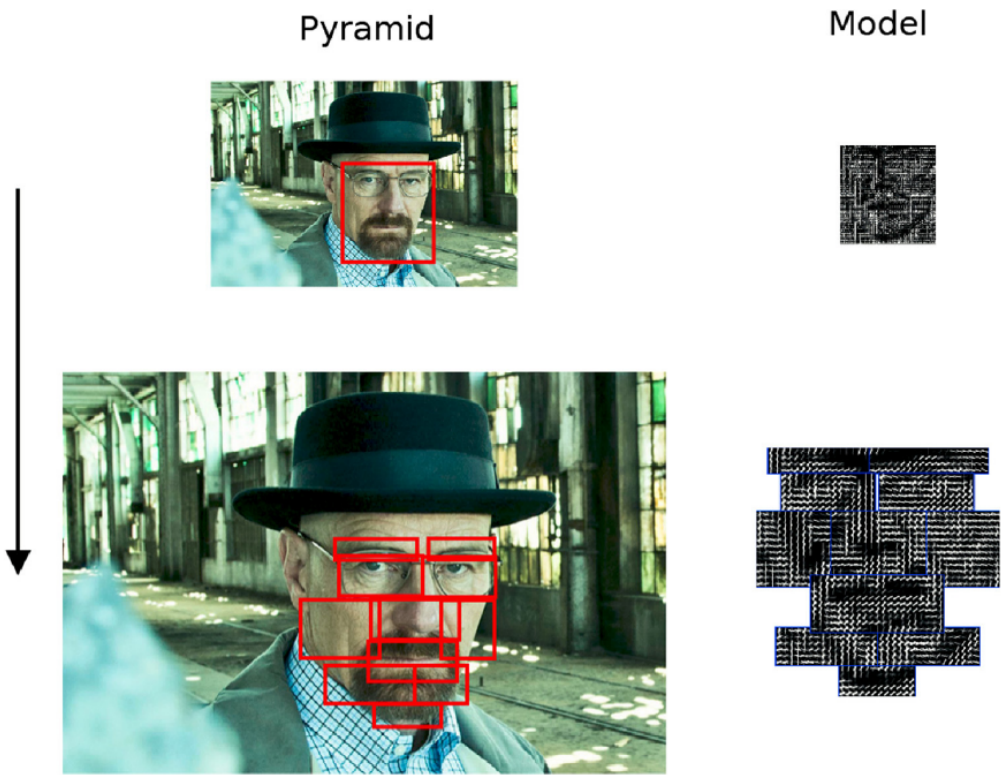


Figure 3.5: The root filter face detection is scaled up in resolution to detect the individual parts that make up the face. Image from [4].

The downside of this method however, is that it is relatively computationally intensive [30]. Every part or component of the object has to run through a classifier, and multiple DPM models have to be combined to achieve state-of-the-art performance [48; 64]. The authors of [64] even report 40 seconds to fully scan an image, while [30] reports "many seconds per image". While there are methods to accelerate the detection process, this is a long way away from our desired detection speed. In contrast to CNNs, this longer testing time is not compensated by an automated learning process of the best features. Also, the training of DPMs is a very time consuming task, with [48] reporting a total training time of roughly one week.
Due to these reasons, DPMs are not deemed suitable for this project and shall therefore not be investigated any further.

3.2.3 Testing phase

Once a classifier has been trained using one or more of the features described above, it is able to tell whether an image depicts a face or not. However, to actually tell where in an image a face is located, some additional steps are required. The usual approach is to slide a window over multiple scaled versions of the image (depending on whether the feature extraction method is scale-invariant or not), extracting every subregion of the image and feeding their features into the trained classifier. According to [65], to detect faces of a minimum size of 20×20 pixels in a 640×480 image, over one million windows need to be classified. This is a small face size and the window in this example slides with a stride of one, but it still gives an indication of the vast amount of image regions to be classified. The classifier is able to tell which of these inputted subregions contain a face and which do not. If multiple detections arise from the same object in the image, non-maximum suppression [66] can be used. This methods computes which of the detections (partly) overlap and are actually the same object. This can be seen in Figure 3.6.

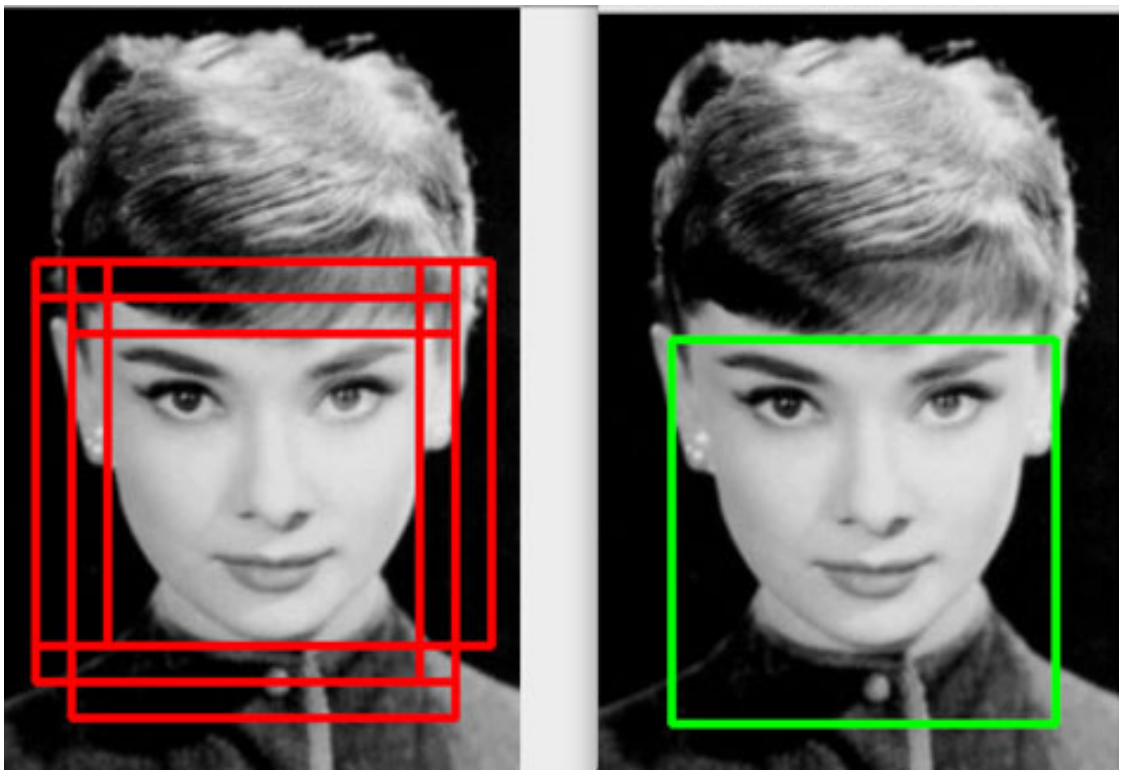


Figure 3.6: Example of the effect of non-maximum suppression: six initial detections are reduced to one [3].

[3]Image taken from *https://www.pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/*

3.3 Face detection algorithms comparison

To determine which of the previously discussed algorithms is best suited for our particular problem, we need to test their accuracy and speed. To do this, a large collection of images of faces with annotated locations is needed. One of these collections is the "Annotated Facial Landmarks in the Wild" (AFLW) database [67].

3.3.1 AFLW database

The AFLW database contains 21123 images with 24384 annotated faces. The "Wild" part of AFLW means that these images are not taken in controlled settings, but in real-life situations. This means the lighting, position, size and rotation of the faces, background and occlusion are all uncontrolled. This makes the face detection a more challenging task than in controlled situations. Examples of harder, somewhat unconventional images in the database can be seen in Figure 3.7. In addition to the location of the faces, the database also contains annotations for 21 landmarks (when visible), sex, occlusion, glasses, use of color, and three rotation angles (roll, pitch and yaw). This extensive annotation makes it easier to test on a subset of images that meets our project's requirements. The resolution of the images in the database varies. An overview of the characteristics of the faces in the database can be found in Table 3.2.

Table 3.2: Characteristics of AFLW database.

Number of images	21123
Number of faces	24384
Number of annotated landmarks	21
Subject male / female	10056 / 14328
Faces occluded / not occluded	1628 / 22336
Faces glasses / no glasses	2048 / 22336
Image greyscale / color	5614 / 18770

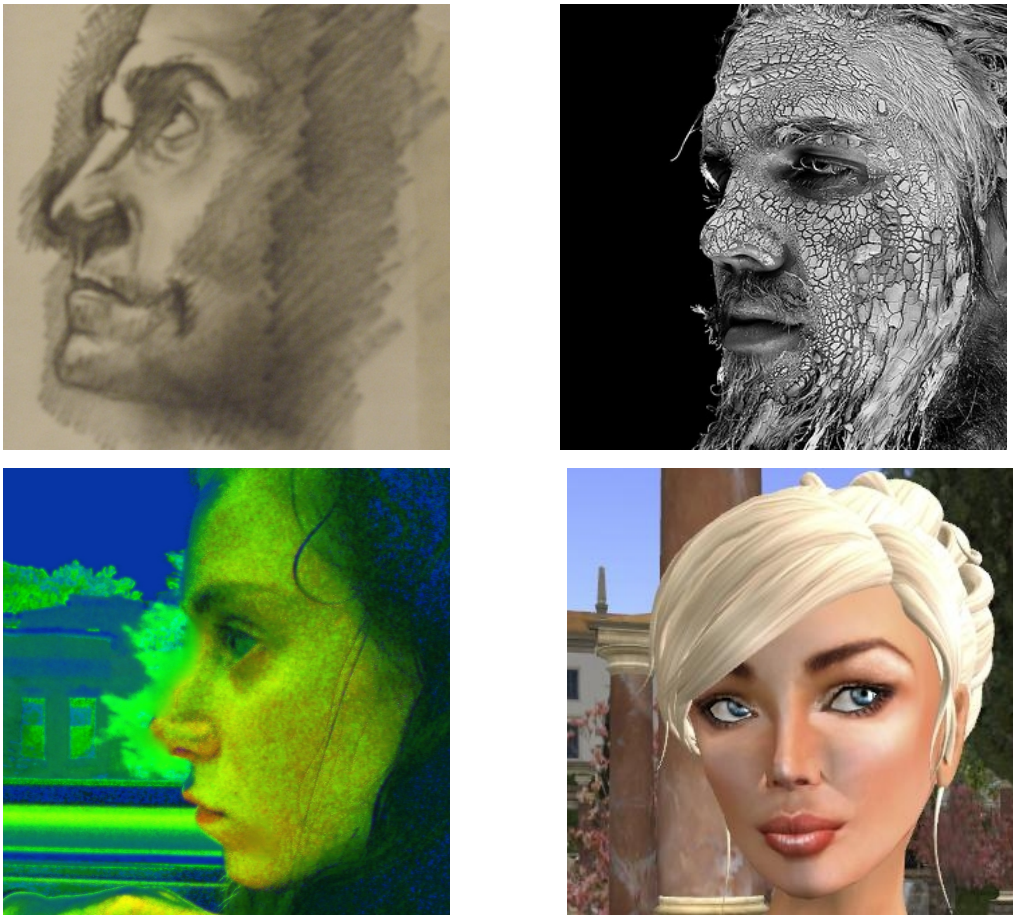


Figure 3.7: Unconventional, hard images in AFLW database.

3.3.2 Algorithm testing

On this database the accuracy and speed of the algorithms mentioned in Section 3.2 can be tested. Because training one of these models is a very time consuming and non-trivial task [30], and because there are several excellent face detector implementations available on-line, in this work we shall refrain from creating our own face detector from scratch. Instead, a face detector will be adjusted to meet the requirements of the project. From the detection algorithms of Section 3.2, three algorithms will be selected with increasing complexity (and therefore, presumably, decreasing speed).

The Haar features of the Viola Jones face detector offer a simple, fast and still very popular approach for face detection. The Histogram of Oriented Gradients method for face detection was developed later and its features are slightly more complex than the Haar features of Viola Jones. The most complex, and most recent, method for face detection is the use of Convolutional Neural Networks. These three methods are very common in the literature for face detection and represent, according to the author of this work, three distinct levels of complexity of face detectors. For all three algorithms,

feature extraction methods and trained classifier models are available in open-source machine learning libraries.

The OpenCV (Open Computer Vision) library [68] is a popular computer vision and machine learning library and offers an implementation of the Viola Jones face detector. Dlib [69] is a machine learning library which has been developed since 2002, and offers implementations of both the HOG face detector [70] and the CNN face detector. The implementations of these libraries shall be used to test the accuracy and speed of the three face detection methods. Both libraries are open source and free to use for commercial purposes, with the exception of the patented SIFT and SURF algorithms (which is also taken into account with the initial algorithm selection).

3.3.2.1 Complete AFLW database

First, the three algorithms are tested on the complete AFLW database. This means that none of the faces are excluded from the test, even though they exceed the requirements for the faces to be detected specified in Section 3.1. This is a good way to compare the accuracy of the three algorithms on an unconstrained set of images. Note that the reported execution time is relative to the machine that runs the algorithm. For comparison, each algorithm has ran on a single core of the same CPU. Recall and precision are two measures of detector accuracy and formulated as follows:

$$Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$TP = true\ positives,\ FN = false\ negatives,\ FP = false\ positives$$

A true positive is the detection of a face that is annotated in the database, a false negative is when there is a face annotated in the database that is not found by the face detector, and a false positive is when the face detector falsely detects an image region as a face. The results of the three face detection algorithms on the complete AFLW database can be seen in Table 3.3 and Figure 3.8.

Table 3.3: Results of the Haar, HOG and CNN face detector on the complete AFLW database.

Algorithm	**TP**	**FN**	**FP**	**Recall (%)**	**Precision(%)**	**Time (s)**
Haar	10015	14369	1828	41.1	84.6	4029
HOG	15198	9186	**1568**	62.3	**90.6**	**3272**
CNN	**21367**	**3017**	3135	**87.6**	87.2	197764

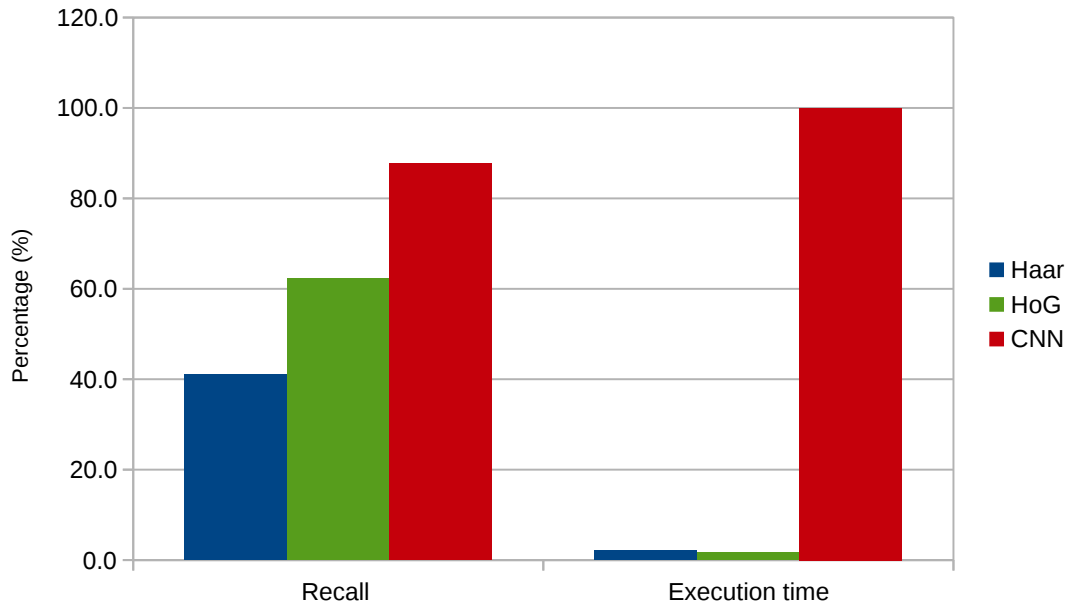


Figure 3.8: Results of Haar, HOG and CNN face detectors on complete AFLW database. Execution time is scaled relative to the slowest method (CNN).

Looking at the recall, unsurprisingly, the CNN performs best. These models are known to cope well with changes in appearance, pose and illumination as long as the dataset they are trained on contains a large enough amount of varying examples. This however, comes at a cost, as the CNN is about 50 times slower than the other two methods. HOG performs second best, with a performance drop of roughly 25% compared to the CNN. Haars recall is the worst, with another recall drop of 21% compared to HOG. This can be explained by the fact that both HOG and Haar are known to suffer in accuracy when there is too much object rotation. Furthermore, since Haar features are based on the subtraction of gray intensity values of different parts of the face, they suffer from irregular lighting. HOG features are more robust in this aspect, since the gradient orientation and magnitude for these features are based on a comparison with neighboring pixels, which have the same lighting conditions.

What is more surprising, is the difference in speed between the Haar and HOG algorithms. Because of the simplicity of the Haar features, the algorithm is expected to be faster than HOG. This can be explained by the use of Advanced Vector Extensions (AVX) 2 Single Instruction Multiple Data (SIMD) instructions. While both libraries support the use of AVX2, not all three algorithms may benefit from this equally. Since it is very time consuming to run all three algorithms on the complete dataset without AVX2 enabled (especially the CNN), a comparison shall be made using the first 100 images from the database. On these images, the face detectors shall run with AVX2 instructions enabled and disabled in order to measure the speedup. The results can be seen in Table 3.4 and Figure 3.9. Note that the amount of benefit these algorithms have from the use of AVX2 SIMD instructions could be an indication for the amount of data-

level parallelism that can be exploited when accelerating the algorithm in a later stage. However, this statement does assume that all three algorithms are optimized equally well for SIMD instructions in their current implementations.

Table 3.4: Comparison of face detection time with SIMD instructions on and off on 100 images

Method	Time no AVX2 (s)	Time AVX2 (s)	Speedup
Haar	**27.9**	27.5	1.01
HOG	378.5	**18.3**	**20.68**
CNN	1957.7	1566.1	1.25

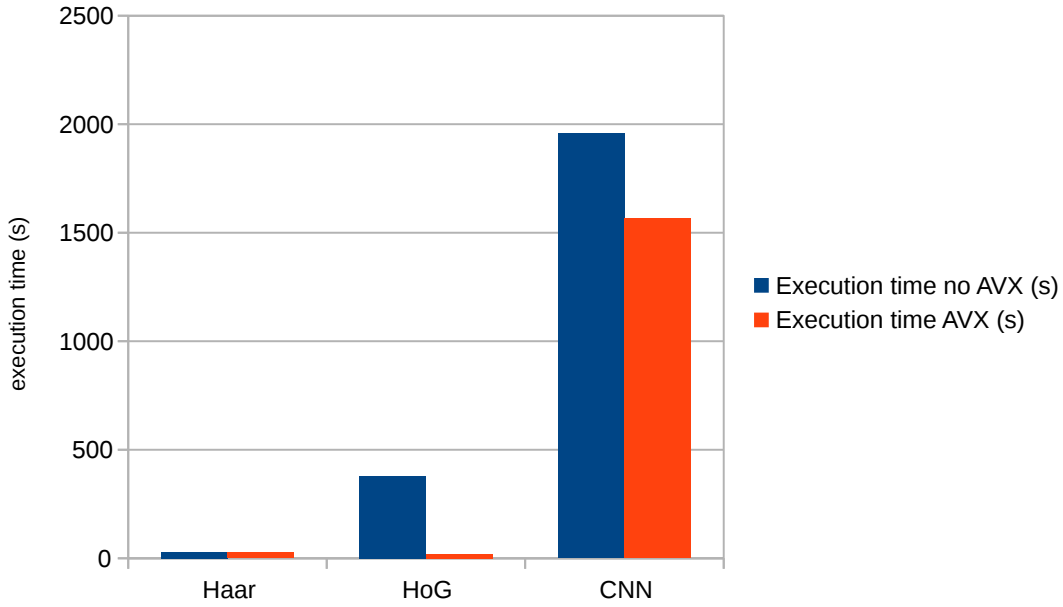


Figure 3.9: Execution time of Haar, HOG and CNN face detectors running with AVX2 SIMD instructions enabled and disabled.

With the algorithms running without the AVX2 SIMD instructions, it can be seen that the Haar face detector is computationally the least expensive, followed by the HOG and finally the CNN. While Haar benefits almost nothing from the SIMD optimizations, the HOG face detector achieves a speedup of over 20×. The CNN gets a small speedup of 1.25×.

The precision and number of false positives also show surprising results and shall be discussed in Section 3.3.6.

3.3.3 Performance on constrained AFLW subset

While the performance on the complete dataset shows the difference in accuracy of each method, the requirements listed in Section 3.1 put constraints on the type of faces that need to be detected in our project, hereby easing the task.

From the AFLW database, a subset of faces can be selected that is within the determined regions of rotation (yaw $\pm 20°$, roll$\pm 25°$, pitch$\pm 40°$). Glasses and nonglasses should both be detected. Lighting conditions are not annotated and can therefore not be filtered. Occluded faces are not filtered out because it is not clear in what way or to what extent they are occluded.

Because some images contain multiple faces, it can occur that one or more face(s) in an image meet the requirements, while others don't. In this case, the image is initially left in the subset. Only when one of the methods is unable to detect a face in the image that doesn't meet the project requirements, the face is excluded from the subset to prevent this method being unfairly penalized compared to the other methods. The results on the remaining subset of 6034 faces can be seen in Table 3.5 and Figure 3.10.

Table 3.5: Results of face detection algorithms on subset of AFLW database that meets project requirements.

Algorithm	**Faces**	**TP**	**FN**	**FP**	**Recall (%)**	**Precision (%)**	**Time (s)**
Haar	6034	4936	1098	680	81.8	87.9	1016
HOG	6034	5242	792	**520**	86.9	**91.0**	**828**
CNN	6034	**5304**	**730**	858	**87.9**	86.1	58489

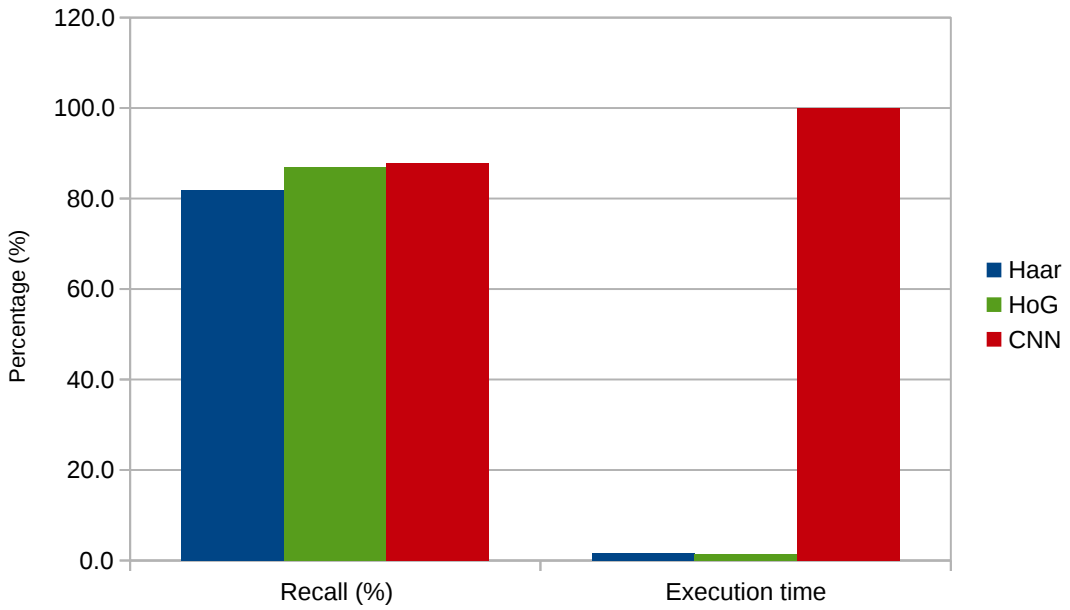


Figure 3.10: Results of Haar, HOG and CNN face detectors on AFLW subset that meets project requirements. Execution time is scaled relative to the slowest method (CNN).

From these results we can conclude that the Haar and HOG face detectors were clearly suffering heavily from the facial rotations in the complete dataset. When these are excluded from the test set, Haar gets a performance boost of over 40% and jumps to a recall of 81.8%. HOG was a little more robust to rotated faces, as its performance was already better on the complete dataset, and gets a performance boost of roughly 25% to a recall of 86.9%. This is very close to the performance of the computationally more intensive CNN, which has a recall of 87.9%. As expected, CNN proved to be very invariant to face rotations as its performance remains almost unchanged compared to its results on the complete dataset.

3.3.4 Upscaling the images

The three face detectors each have a minimum size for a face to be detected. This is the size of the images with which the classifier is trained. A face smaller than this size could therefore be missed by the face detector. To detect smaller faces, images can be upscaled, but this comes with a trade-off; a larger image means more pixels to compute features of, and more image subregions to be classified. To see the effect of the minimum face size of each face detector on the detecting accuracy, all images have been upscaled to double width and height. The results of this upscaling on the same limited-rotation subset of the AFLW database can be seen in Table 3.6 and Figure 3.11.

Table 3.6: Results of face detectors on limited-rotation subset of AFLW database with images upscaled to double width and height.

Algorithm	Faces	TP	FN	FP	Recall (%)	Precision (%)	Time (s)
Haar	6079	5048	1031	1339	83.0	79.0	3141
HOG	6079	5786	293	**853**	95.2	**87.2**	**2795**
CNN	6079	**5956**	**123**	1437	**98.0**	80.6	193449

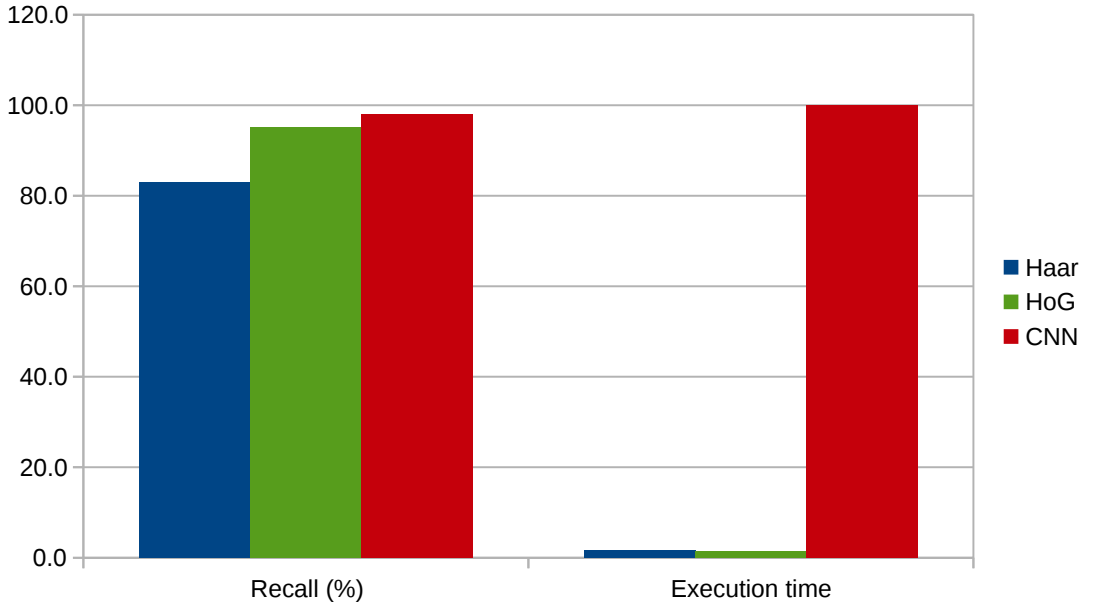


Figure 3.11: Results of Haar, HOG and CNN facedetectors on upscaled AFLW subset that meets project requirements. Execution time is scaled relative to the slowest method.

The Haar face detector is trained with a small input image size (24×24) and doesn't benefit much from the image upscaling. Both HOG and CNN do get a significant boost in recall and detect almost all the faces in the subset of the database. They were trained with a larger input image size of 80×80, which means the upscaling lets them detect faces with a minimum size of 40×40 now. It can also be seen that the upscaling has a significant effect on the detection speed, with all methods taking more than three times as long to analyze all the images in the AFLW subset.

For this project however, the upscaling will not be needed, since the faces in the video will fill up at least 20% of the image, which has a resolution of 640×480. This means that the face is at least 128×96 pixels in size. If we can determine and limit the range of scales of the sliding window of the detection method in our project, this will greatly increase the detection speed.

3.3.5 False negatives

From the remaining false negatives of each method, 96 examples are shown in Appendix A.1.

When looking at the faces the Haar faced detector failed to identify we can still see a great variety of possible causes. Many of the faces seem to be partly occluded by a great variety of objects: hair, glasses, sunglasses, hands, leaves, a popsicle, a microphone etc. Other faces are slightly rotated, suffer from bad lighting or seem somewhat blurry.

The HOG face detector seems to suffer from the same problems, although to a lesser extent. Face occlusion is still a problem, with sunglasses being the number one cause. There are also quite some images that have extremely irregular illumination. While the local gradient calculations should be robust to illumination variation, certain parts of these images are too dark or light for correct gradient calculations. Another cause of face detection failures is greasepaint on the face.

Although the images are upscaled to twice the original width and height, one of the most common causes of false negatives of the CNN remains a face size that is too small. This could be solved by further upscaling the images. Two other, already discussed, causes are heavy occlusion of the face and the use of greasepaint.

3.3.6 False positives

One important aspect that has not been discussed yet is the amount of false positives of each method, although it is important for the project that the face detector does not generate (too many) false positives. Appendix A.2 shows the first 96 false positives of each face detection method. We define a false positive as a face that is detected by the face detector, but is not annotated in the AFLW database.

It turns out the AFLW database contains annotations mistakes. Out of the first 96 "false positives" of the Haar face detector, 20 images do in fact depict a face, which is close to 21%. This means the Haar face detector actually produces less false positives than the previous tests suggest. Images of false positives detected by the HOG method confirm the annotation mistakes. In the first 96 "false positives", as many as 71 images contain a face, roughly 74%.

To see which of the "false positives" are valid and which are not, we shall use the CNN to reclassify the false positive images as face or non-face. With a recall of 98.0%, the CNN is not perfect but will give a strong indication of the amount of true false positives of each method. Both the results of the Haar and HOG face detector can be seen in Table 3.7. Since we cannot use the CNN to reclassify its own false positives, we are not able to re-evaluate the false positives of the CNN in this way. In the first 96 false positives of the CNN depicted in Figure A.6 however, every single image depicts a face, and is therefore in fact not a true false positive but an annotation mistake. This leads to believe that the precision of the CNN is much higher than the 80.6% reported in Table 3.6, and the highest of the three compared algorithms.

Table 3.7: Number of false positives re-evaluated with CNN classification because of AFLW annotation errors. FP AFLW indicates the number of false positives according to AFLW database annotations, while FP CNN indicates the remaining false positives after CNN reclassification has been performed.

Algorithm	**FP AFLW**	**FP CNN reclassification**	**TP**	**Precision (%)**
Haar	1344	897	5048	84.9
HOG	**791**	**114**	**5786**	**98.1**

Both methods clearly have a higher precision than previous testing on the AFLW database suggested. The number of false positives decreases with 33% for Haar and 86% for HOG. It also increases the relative difference in accuracy between the two methods, with Haar now resulting in nearly seven times as much false positives as HOG.

Note from Table 3.5 and Table 3.6 that the number of false positives classified by each method became much greater when the image was upscaled. Limiting the number of scales and not upscaling the image in this project will not only result in faster computation times, but also in the reduction of the number of false positives; less subregions to scan also means less potential subregions to be falsely classified.

3.3.7 Conclusion face detection algorithm

The three face detectors show clear differences in both speed and accuracy. The CNN is the most accurate face detector. It is reliable in varying conditions and settings, and was even able to detect many faces that were not annotated in the AFLW database. Judging from Table 3.4 however, it is also much slower than the other two methods. In the controlled setting where the eyeblink responses are recorded, the method is somewhat of an overkill. Since we are also focusing on speed, the CNN shall not be be used as face detector in this project.

Both the Haar and HOG algorithms provide faster alternatives, but where Haar lacks in recall and precision on the controlled AFLW subset, HOGs accuracy on the AFLW subset is almost on a par with the CNN. The amount of false negatives and false positives are respectively three and seven times higher with Haar than with HOG face detection. This is very undesirable and will, in our case, result in missing parts of the blink response and faulty random estimates of the closure of the eyelid on objects that are not a face. Although the recall of the HOG algorithm is not perfect, note that the AFLW database contains much harder images than we will typically come across in the controlled eyeblink recording setting (compare Figure 1.2 with Figure 3.7). Furthermore, the speedup the HOG method achieves from using AVX2 SIMD instructions could indicate a significant possible gain from extra data-level parallelism.

For the above-mentioned reasons, the **Histogram of Oriented Gradients** algorithm shall be used for face detection in this project.

3.4 Eyelid closure detection

Once the face has been located in an image, the second step is to detect the eyelid closure. We can approach this problem in two different ways: eye detection followed by eyelid closure detection, or landmark detection where the landmarks on the eyelids are used for the eyelid closure detection.

3.4.1 Eye detection followed by blink detection

The first solution requires a similar approach as the face detection. Features are calculated on the part of the image containing the face. A window slides across the face image at different scales, and each subregion is classified as eye or non-eye. Once the image region of the eye has been found, a second algorithm can be used to determine the amount of eyelid closure. The distinction between the left and right eye can be made based on their location in the image.

The ideal case would be that the same HOG features used for face detection are also suitable for eye detection. In this way we would be able to avoid an extra feature extraction step, as the same features would be used as input for a different classifier. This is not deemed very likely, since different objects often have different features that achieve high classification accuracy. Eyes are particularly difficult objects since they deal with common object recognition difficulties (see Section 2.2.3) in an order of magnitude and frequency rarely seen with other objects [71]. In fact, one of those difficulties is the occlusion of the eye by the eyelid, the exact thing we are looking for in this project. Looking at the difference in appearance of open and closed eyes in Figure 3.12, the difficulty of recognizing both with a single model becomes apparent.

Figure 3.12: The difference in appearance of open and closed eyes [5].

The extensive survey of [71] discusses 76 different papers that address eye detection using a video-based approach, and splits the models into shape-based approaches, appearance-based approaches and hybrid approaches. Most of these approaches, however, focus on features that describe the eye in an open state. In fact, the writers conclude that only four of the models show robustness to occlusion due to eye blinks or closed eyes. Furthermore, the survey makes no mention of HOG features used for eye detection, which is a strong indication that we would not be able to re-use the same features that were used for face detection.

The "eye-occlusion-robust" models [72] make use of template matching on the region between the eyes, as it is more constant in appearance than the eyes themselves. The eyes are searched in a small area around this "between-the-eyes" region. To initialize the "between-the-eyes" template, they detect the position of the eyes by detecting blinks based on differences in successive images in a video. This approach is not suitable for our project since our videos generally consist of a single blink, and there is much less difference between successive frames because the video is shot at a high frame rate (see Section 3.1).

Classifiers can also be trained on images, or features of images, of both open and closed eyes. The OpenCV library contains a trained classification model based on Haar features for the detection of both open and closed eyes. Furthermore, in our project a CNN was trained on roughly 120.000 images containing open and closed eyes from the RPI ISL eye training database [73], and on equally as many non-eye images. Note that this was done in a rapid-prototyping way, and therefore the CNN was not fully optimized for maximum accuracy. An example of the eye detection performed by both the Haar and CNN implementations on an image from the BioID database [74] is shown in Figure 3.13. Both methods struggle with false positives; mouths and nostrils get mistaken for eyes. This could be improved by adding specific images of these false positives to the training set of the CNN and label them as non-eye. However, due to the computationally expensive nature of these algorithms (see Section 3.2.1.3), this approach was not considered any further.

Figure 3.13: Examples of false eye detections using Haar (left) and CNN (right) algorithms. The largest square is from the face detection.

Literature on *blink* detection seems to focus on the detection of an open eye as initialization step, followed by tracking of features or template matching in subsequent frames. The authors of [75] use the same initialization method with blinks as described above and employ an updated template to track the eye region. They detect blinks based on the correlation score of the template. In [76], the author uses a Haar classifier to initialize the location of the eyes, followed by tracking of features in the eye image with a Kanade–Lucas–Tomasi (KLT, named after the authors) tracker [77]. The movement of these features indicates a blink of the subject. A similar approach is used in [78], where the variance of two subsequent frames is calculated to initialize the eye location when the subject blinks, followed by the use of KLT features for eye tracking. Haar features are used for face and eye detection in [79], followed by the fitting of a circle on the contour of the upper eyelid. Based on the properties of this circle, the eye state is predicted to be open or closed. In [80], the same Haar features are used for eye localization, but LBP histogram features are used for the binary classification of the eye images as open or closed.

While these methods are able to detect the blink of an eye, they don't concern themselves with detecting the amount of eyelid closure. However, once the location of the eye is known, Gabor filters, HOGs or optical flow features could be used to achieve an estimate of eyelid closure [81; 82]. This would also be the required extra step for the methods that only detect the eye that were described previously.

3.4.2 Landmark detection

A different approach to the eyelid closure detection is the use of face landmark detection, which is also called face alignment. In the survey of [9], 48 different methods of face alignment are discussed and divided into eight classes. Both their accuracy and speed are compared. Most of the methods discussed in the survey focus on the detection of the 68 landmarks shown in Figure 3.14. On each eye, six landmarks are located: two in the corners, two on the upper eyelid and two on the lower eyelid. If these are accurate enough, they could be used directly for the calculation of eyelid closure. In [83], these landmarks are already successfully used for blink detection.

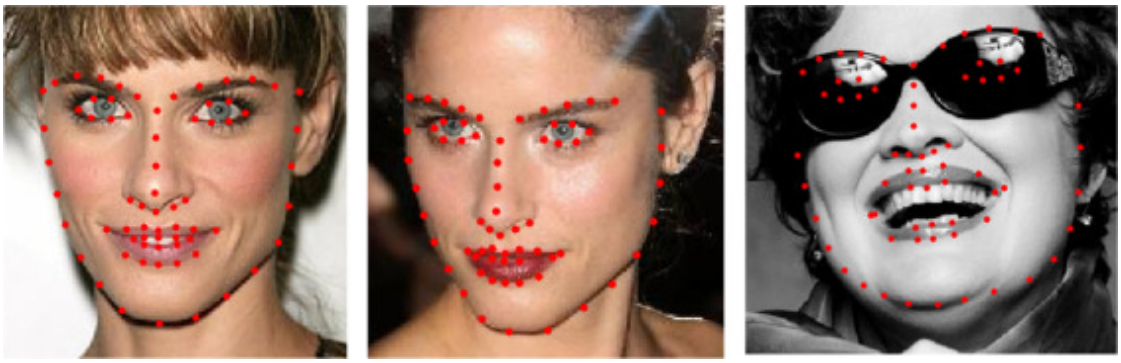


Figure 3.14: Example of landmark detection on three faces [6].

Not all the methods are evaluated on the same database. The database on which the most the most face alignment methods are evaluated, is the HELEN database [84] (18 methods). Out of the five used databases, this is also the one in which the image conditions are the most similar to the conditions in our project. The error is measured as the euclidean distance of each detected landmark to the true (annotated) landmark, divided by the inter-ocular distance for scale invariance. Results of each method can be seen in Table 3.8, which is taken directly from [9].

Table 3.8: Comparison of the performance of different landmark detection methods on the HELEN database. Number of points indicates the number of landmarks a method detects. Table taken from [9].

Method	**#Points**	**Error (%)**	**FPS**
Stacked Active Shape Model		11.10	-
Component-based ASM		9.10	-
Explicit Shape Regression		5.70	70 (C++)
Robust Cascaded Pose Regression		6.50	6 (Matlab)
Supervised Descent Method		5.85	21 (C++)
Ensemble of Regression Trees	194	4.90	1000
Local Binary Feature		5.41	200 (C++)
Fast Local Binary Feature		5.80	**1500** (C++)
Coarse-to-Fine Shape Searching		4.74	-
Coarse-to-Fine Shape Searching Practical		4.84	-
Cascade Gaussian Process Regression Trees		**4.63**	-
Tree structured Part Model		8.16	0.04 (Matlab)
Discriminative Response Map Fitting		6.70	1 (Matlab)
Robust Cascaded Pose Regression		5.93	12 (Matlab)
Supervised Descent Method	68	5.67	70 (C++)
Gauss-Newton Deformable Part Model		5.69	70
Coarse-to-fine Auto-encoder Networks		5.53	20
Deep Cascaded Regression		**4.63**	-

The table is incomplete as it does not list the speed and programming language for every method, but it is used as an indication for which landmark detection method might be suitable for this project. The method that seems the most promising is the "Ensemble of Regression Trees" (ERT) [7], as it performs fifth-best in accuracy and second-best in speed. This method belongs to the class of cascaded regressors, which refines its estimates of the landmark locations in a number of consecutive stages, and is described by the survey as "one of the most popular and state-of-the-art methods for face alignment, due to its high accuracy and speed".

To further review the accuracy of this algorithm, we test its implementation in Dlib on the BioID database. This database contains 1521 images of 384×284 pixels taken in conditions that strongly resemble the ones in our project (see Figure 3.13), and of which the location of the center of both eyes is annotated. The algorithm predicts the

location of 68 landmark. From the six landmarks on each eye the center is calculated, and compared with the annotated location. The error was calculated in the same way as in the landmark detection survey described above. For all eyes in the database, the average error was 3.66%.

Furthermore, the eyeblink response of multiple blink videos of the Erasmus Neuroscience Department was plotted using this landmark detection algorithm. An example of the landmark detections on one of the images from these videos, as well as the plotted eyeblink response of that video, can be seen in Figure 3.15 and Figure 3.16, respectively. Our neuroscience peers deemed the eyeblink response graphs satisfactory. Additionally, the potential for other landmarks to be used to research the behavior of facial muscles during eyeblink conditioning was identified.

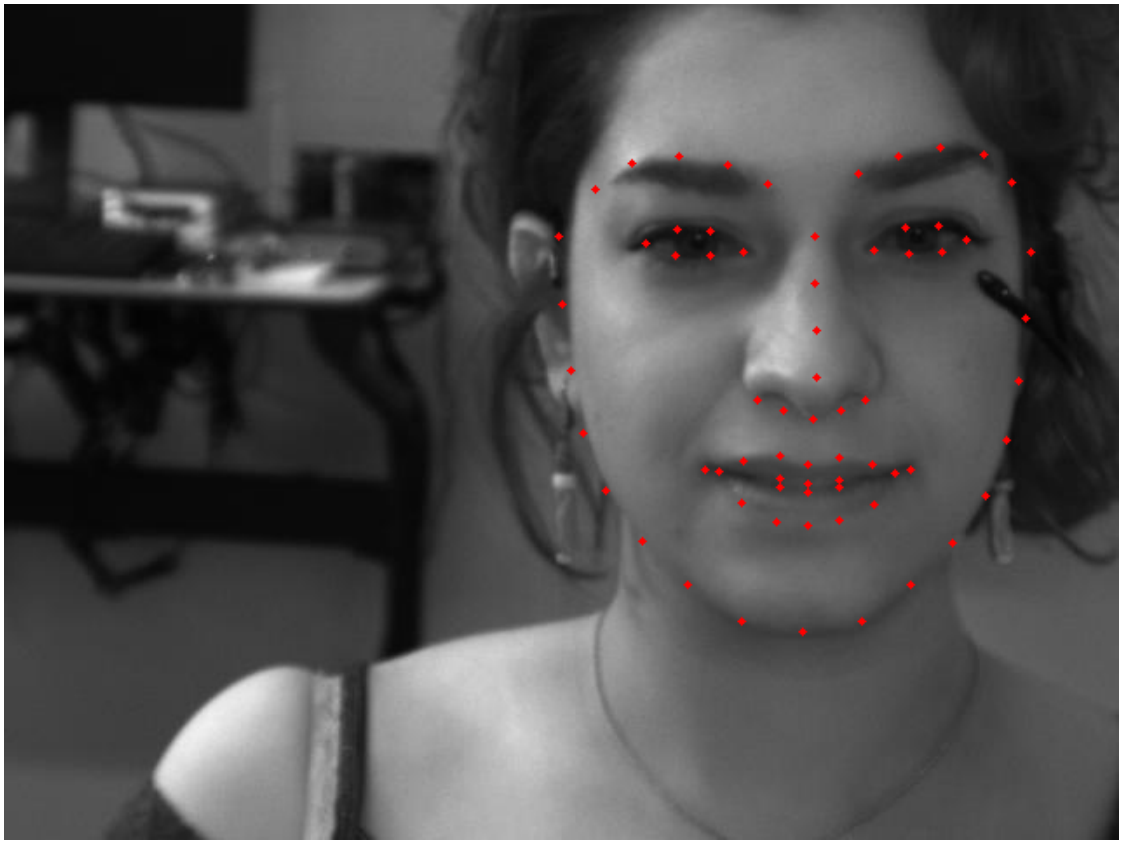


Figure 3.15: Example of the landmark detection on an image from the Erasmus eyeblink conditioning videos.

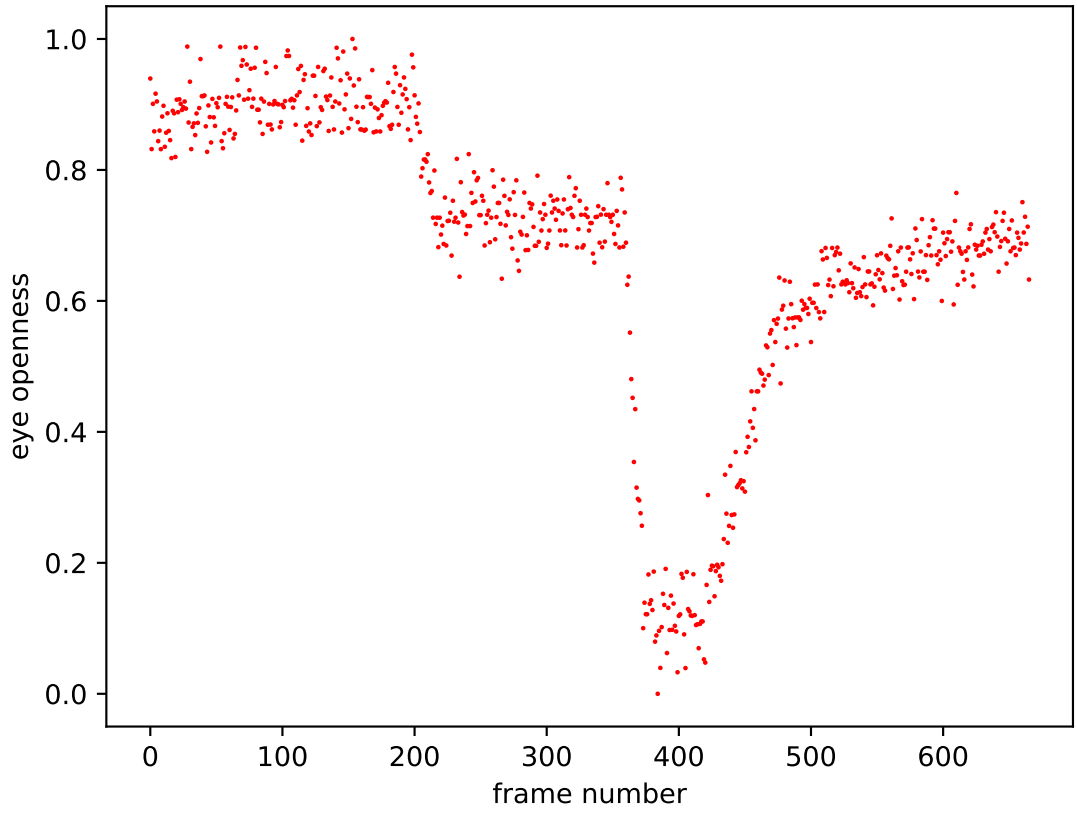


Figure 3.16: Example of an eyeblink response graph created with the landmark detection algorithm.

3.4.3 Conclusion on eyelid closure detection algorithms

The landmark detection and eye detection followed by eyelid closure detection, are two different approaches to detect the eyeblink response. The most promising method of the first approach is a sequence of algorithms that detects the eye, tracks its location and estimates its closure. In contrast, the second approach only requires one algorithm to estimate the landmarks, from which we can directly calculate the level of eyelid closure. In addition, the landmarks on other parts of the face are interesting for research on face muscle movement during eyeblink conditioning. The Ensemble of Regression Trees (ERT) algorithm [7] provides a fast and accurate way of landmark detection and is implemented in the Dlib library.

For the aforementioned reasons, the **Ensemble of Regression Trees** algorithm for landmark detectoin of [7] shall be used for eyelid closure detection in this thesis.

Details of the selected algorithms

4

In this chapter the selected algorithms of Chapter 3 will be discussed in detail. Furthermore, the algorithms will be profiled and an analysis of different acceleration methods will be conducted. Section 4.1 will discuss the face detector, while Section 4.2 will cover the landmark detector. Both algorithms are implemented in the Dlib library in the C++ language, and profiling is done on a single core of an AMD Ryzen 7 1800X CPU (3.6 GHz).

4.1 Face detector

The HOG face detector of the Dlib library is based on the feature extraction method of [4] and five classifiers trained on 3000 images of the Labeled Faces in the Wild (LFW) database [85]. The five classifiers are trained on five different facial rotations to make the face detector more rotation-invariant, and an image is scanned at multiple scales to make the face detection more scale-invariant.

4.1.1 Algorithm analysis

We will divide the algorithm into multiple steps that are also used later in the acceleration stage (see Chapter 5):

- **Scaling**: The original image is scaled down in multiple steps until either the width or height of the image is smaller than those of the detection window (80×80). This is done by a factor of $\frac{5}{6}^{th}$ per scale in the original implementation and not changed in this work. The closer this factor is to zero, the bigger the scaling steps are, and the smaller the amount of total scales to be analyzed is. Therefore, a factor closer to zero will increase the chance of missed face detections. The computation of pixel values of the downscaled images is done by ways of bilinear interpolation. Note that every step from now is done on all scaled versions of the image.

- **Gradient computation**: For every pixel in the image, the gradient is computed by applying the finite difference filter $[-1, 0, +1]$ for the x-direction at its transpose for the y-direction. The gradient orientation is computed as $tan^{-1}(\frac{grad_y}{grad_x})$ and discretized into one of eighteen signed directions. The gradient magnitude is calculated as $\sqrt{grad_x^2 + grad_y^2}$.

- **Histogramization**: The image is divided into cells of 8×8 pixels. Each cell is described by a histogram of eighteen bins, corresponding to the eighteen discrete gradient orientations of the pixels. Each pixel adds its gradient magnitude to bins, corresponding to this gradient orientation, of four surrounding histograms. Bilinear

interpolation is used to scale the contribution to each of the four histograms based on the location of the pixel.

- **Normalization**: The energy of each cell is computed by summing over the square of nine unsigned orientation bins (the two bins of opposite orientations in the signed histogram are added). The histogram bins of each cell are normalized based on the energy value of the cell and its eight neighbors.

- **Feature computation**: The final feature vector of each cell contains 31 features: 18 signed normalized histogram bins, 9 unsigned normalized histogram bins (where opposite orientations have been added together), and 4 gradient energy features, capturing the cumulative gradient energy of square blocks of surrounding cells. This completes the feature extraction phase: the original (scaled down) image is divided into cells of 8×8 pixels, and each of those cells is described by a total of 31 features. We shall refer to this converted image as the feature image.

- **Classification filter**: The classifier is trained with a face size of 80×80 pixels, or 10×10 cells. This means that the features of an area of 10×10 cells, 3100 in total, are used as input for the classifier. Each of these features is multiplied by a certain weight, and when the total exceeds a threshold, the area is classified as a face. This is done for every area of 10×10 cells in the feature image. The multiplication of the weights is done in two steps. First, a row filter is applied to every feature of every cell, which multiplies the same feature of the ten horizontally neighboring cells with a different weight. The value of each feature of each cell is now the weighted accumulation of its ten horizontal neighbors. Next, the same process is done for every cell using a column filter for the ten vertically neighboring cells using the new accumulated values. When this is done, each cell contains a total of 31 features, each representing a weighted accumulation of the same feature of its 10×10 neighboring cells. The total accumulation of these features is the detection score, and the image representing these detection scores for each cell is called the saliency image. As previously stated, there are five classifiers, for five different rotations of the face, each with its own filter values. Therefore this whole process is repeated five times; once for every classifier.

- **Detection**: The detection score of each cell is compared to the threshold of the classifier. If it is higher than the threshold, the cell is the center of an area of 10×10 cells which is classified as a face. This is done for all five saliency images.

- **Non-maximum suppression**: Since multiple detection can arise from the same face, we need to decide which detections are from different faces and which are from the same face. This is done by calculating the overlap of the detection rectangles. When two detections overlap more than 50%, the detection with the highest score is chosen as the final detection.

These steps are visualized in Figure 4.1.

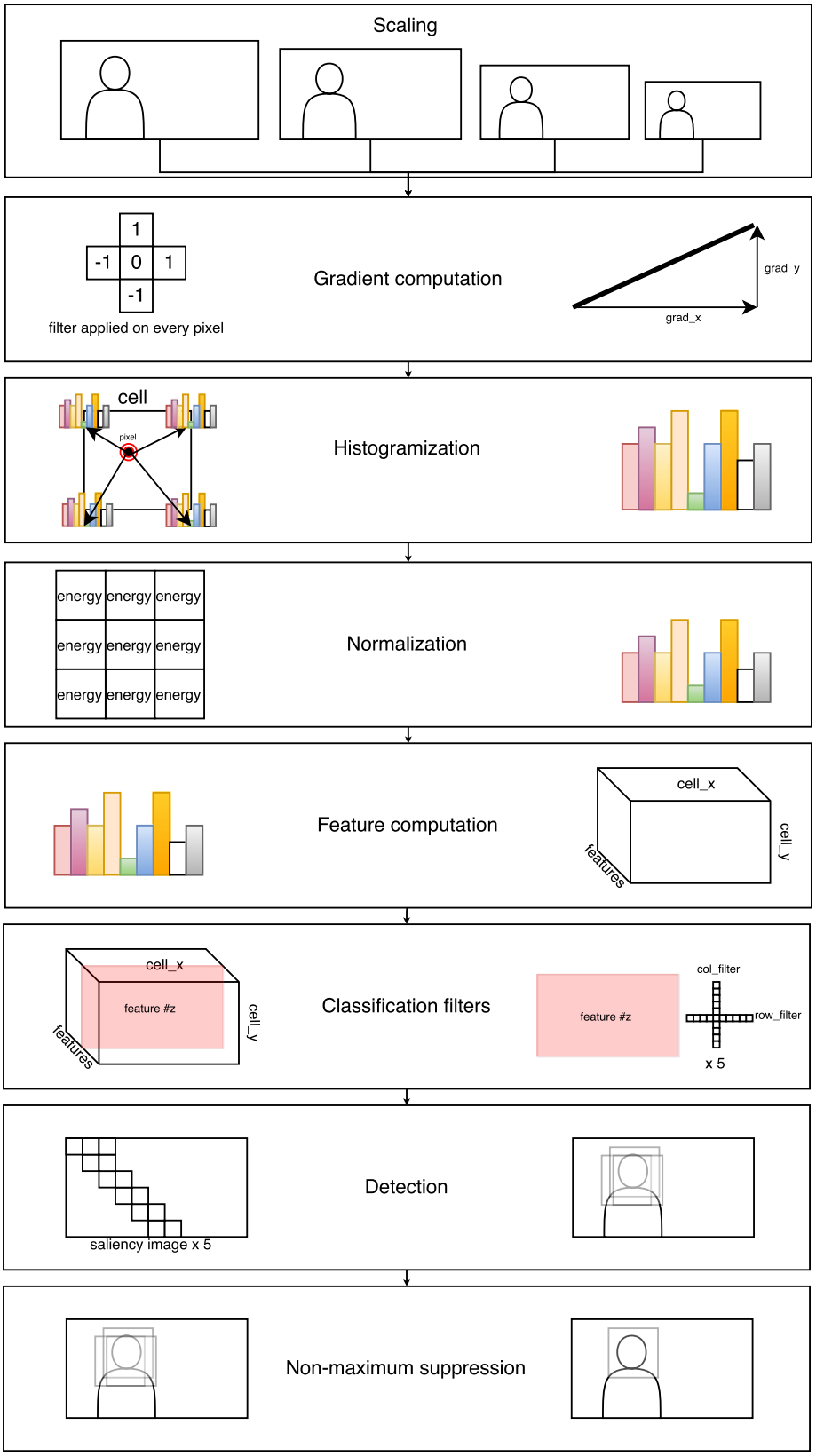


Figure 4.1: Flow graph of face-detection algorithm.

4.1.2 Algorithm profiling

To see which parts of the face-detection algorithm are the most computationally expensive, it is profiled with Valgrind [86] on 10 images from Erasmus MC eyeblink videos (640 × 480 resolution). AVX2 instructions are not enabled to compare the computational load of different functions fairly (some functions might benefit more from them than others). Figure 4.2 shows a call graph of functions of the face-detection algorithm in which more than 10% of the total face detection execution time is spent.

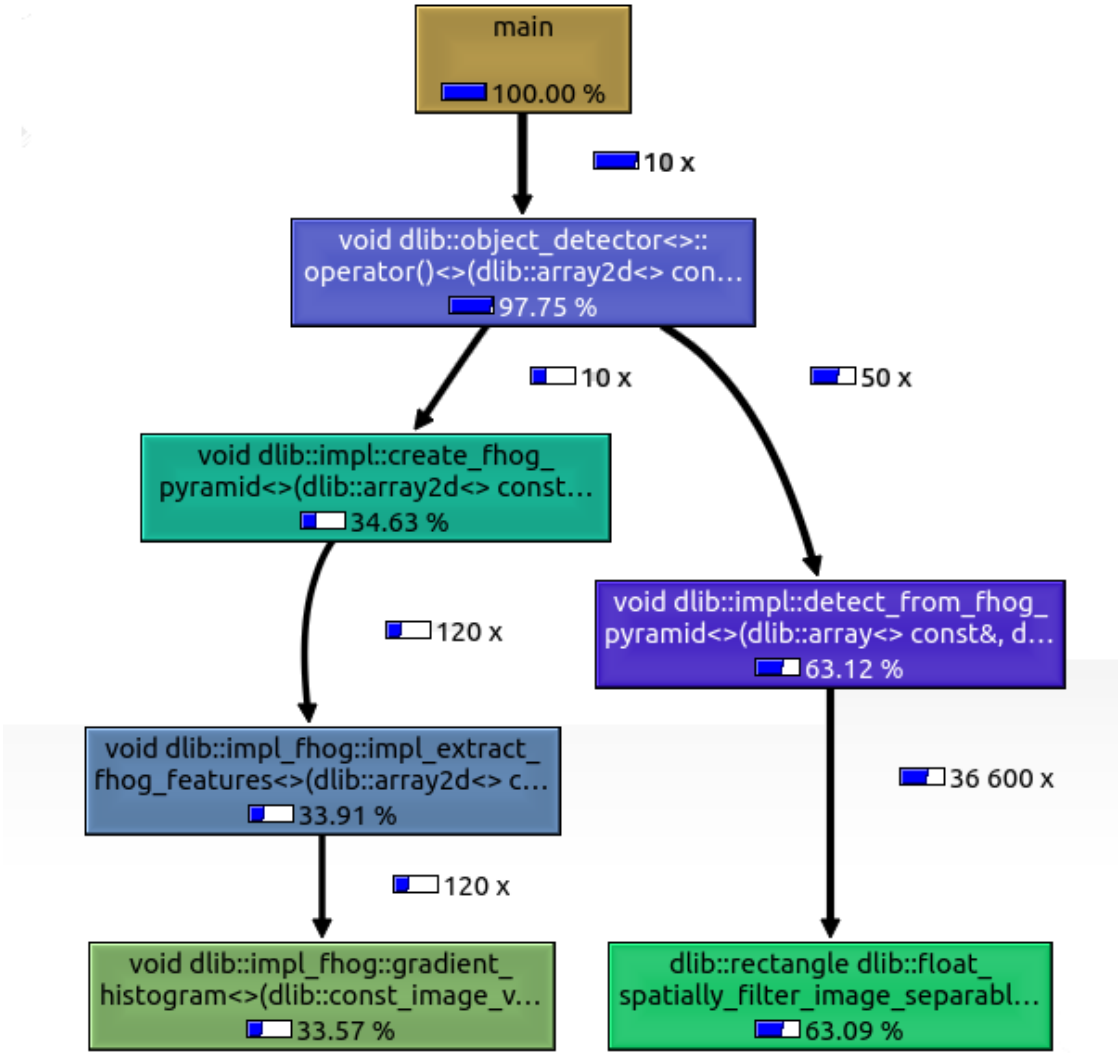


Figure 4.2: Call graph of face-detection algorithm. Percentages indicate the amount of time spent in a function compared to the total face-detection execution time. Arrow values indicate the number of times the function is called.

In `create_fhog_pyramid`, the image is scaled down, and in `impl_extract_fhog_features` features are extracted for each scale of the image

(i.e. aforementioned the gradient computation, histogramization, normalization and feature computation steps). The gradient computation and histogramization take up 99% of the feature extraction process. `Float_spatially_filter_image_separable` applies the row and column filters, `detect_from_fhog_pyramid` compares the saliency images with the detection thresholds and the non-maximum suppression is a function of the `object_detector` class. From this we can deduce that the majority of the time is spent in applying the classification filters to the feature image. Furthermore, processing the ten images takes **6.03 seconds** without AVX2, and **0.31 seconds** with AVX2, which is in line with our previous finding in Section 3.3.2.1.

4.1.3 Algorithm acceleration

To detect the eyelid closure at the required speed for this project, both the face detection and landmark detection need to be accelerated. Because we are dealing with videos of 1000 frames (assuming 500 FPS and two second videos), one of the more obvious ways to accelerate the face detection would be to make use of task parallelism. Each processing element can process its own share of frames completely independently from the others, requiring no communication or synchronization. This would require almost no alterations to the algorithm, as each core receives a private copy of the `object_detector` object and gets assigned a number of frames to perform the detection on. With current high-end consumer grade CPUs having as much as 8 cores and 16 threads (Intel Core i7 Extreme, AMD Ryzen 7), this could lead to a significant speedup. High Performance Computing (HPC) many-core processors such a the Intel Xeon Phi boast around 60 cores and over 200 threads, which further increases the task parallelism speedup potential.
Another parallel computing model well-suited for this problem is data-level parallelism, where the same operation is performed on independent data. This is a model that is frequently used for the acceleration of image processing applications, since the same operations are often performed on each individual pixel. Looking at the aforementioned steps of the HOG algorithm, we can distinguish different "levels" of data parallelism per step:

- Scaling, gradient computation and histogramization operations are performed on every pixel.
- Normalization, feature computation and detection are performed on every cell.
- Classification filters are performed on every feature of every cell.
- Non-maximum suppression is performed on a small number of detections and therefore not well-suited for data parallelism, but its computational cost is negligible.

Factors that would limit the performance of this approach are the required synchronization after each step of the algorithm, and the use of atomic operations to prevent race conditions during the histogramization step. Furthermore, the use of an external accelerator requires data transfers to and from the accelerator memory.

Modern high-end consumer GPUs consists of several thousands of cores which can be used to exploit this data parallelism (see Section 2.3. They are known for their single-precision floating-point processing power, which is the data type that is mostly used

in the HOG algorithm. Furthermore, the cores of a GPU are divided into a number of Streaming Multiprocessors (SM), which can perform their own tasks independently from other SMs by using streams (see Section 5.1.11). This allows for the possibility of task-parallelism within the GPU; different SMs can run different kernels or process different images at the same time.

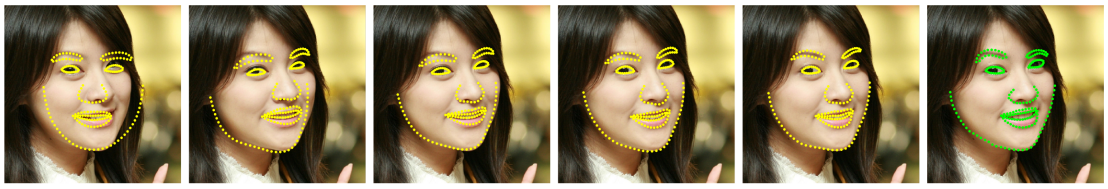
The high level of possible data-level parallelism of the algorithm, combined with the possibility of task parallelism across multiple images, seems well-suited for GPU acceleration. To achieve this, we will make use of CUDA [87], the parallel computing platform and programming model for GPGPU developed by NVIDIA. The advantages of this platform are that it is open-source, supported by a large community, comes with excellent profiling tools, and supports easy programmability compared to other solutions (such as OpenCL).

4.2 Landmark detector

The ERT landmark-detection algorithm of the Dlib library is based on the algorithm described in [7] and trained on the iBUG 300-W face landmark dataset [88].

4.2.1 Algorithm analysis

In contrast to the face-detection algorithm, the landmark-detection algorithm is an iterative process. It is also called a cascaded-regression approach. Each iteration, or level of the cascade, refines the estimate of the landmark locations, as can be seen in Figure 4.3 for a number of iterations.



(a) $T = 0$ (b) $T = 1$ (c) $T = 2$ (d) $T = 3$ (e) $T = 10$ (f) Ground truth

Figure 4.3: Landmark estimates as the number of iterations (T) progresses [7].

The algorithm starts with an initial estimate of landmark locations that is based on the mean shape of all the images it has been trained with, centered at the middle of the face image. In each of the 15 levels of the cascade, a total of 500 regression trees each calculates a shift of these landmarks to make the detection more accurate. The results of all the regression trees in a level are added to the current landmark estimation, which results in the landmark-shape estimate for the next level.

The depth of each regression tree is 5 and its number of leaves 16 (see Figure 4.4). In each level of the tree, either the right or the left child is chosen, based on the intensity-difference of two pixels. The critical point of this approach is that the location of these pixels is indexed relative to the landmark-estimation shape of the current cascade level.

At each level of the cascade, the new locations of the required pixels for the decision-splits of all 500 regression trees are calculated. This is done by calculating the similarity transform between the current-shape estimation and the original mean-shape estimation. The feature points are indexed relative to the initial mean shape and undergo the same transformation to calculate their position relative to the current shape estimate.

This process can be summarized in the following steps, which are also shown in Figure 4.4:

- **Initialization**: Initialize the landmark-shape estimate. This is the mean of all landmark shapes with which the detector has been trained.
- **Feature computation**: Calculate the similarity transform between the current shape estimate and the original mean-shape estimate. Use this transformation to calculate the new location of the feature points for the regression trees.
- **Regression-tree estimation**: Traverse each of the 500 regression trees based on pixel intensity differences and add their results to the current landmark-shape estimate.
- **Repeat**: Repeat the feature computation and regression-tree estimation step for each level of the cascade.

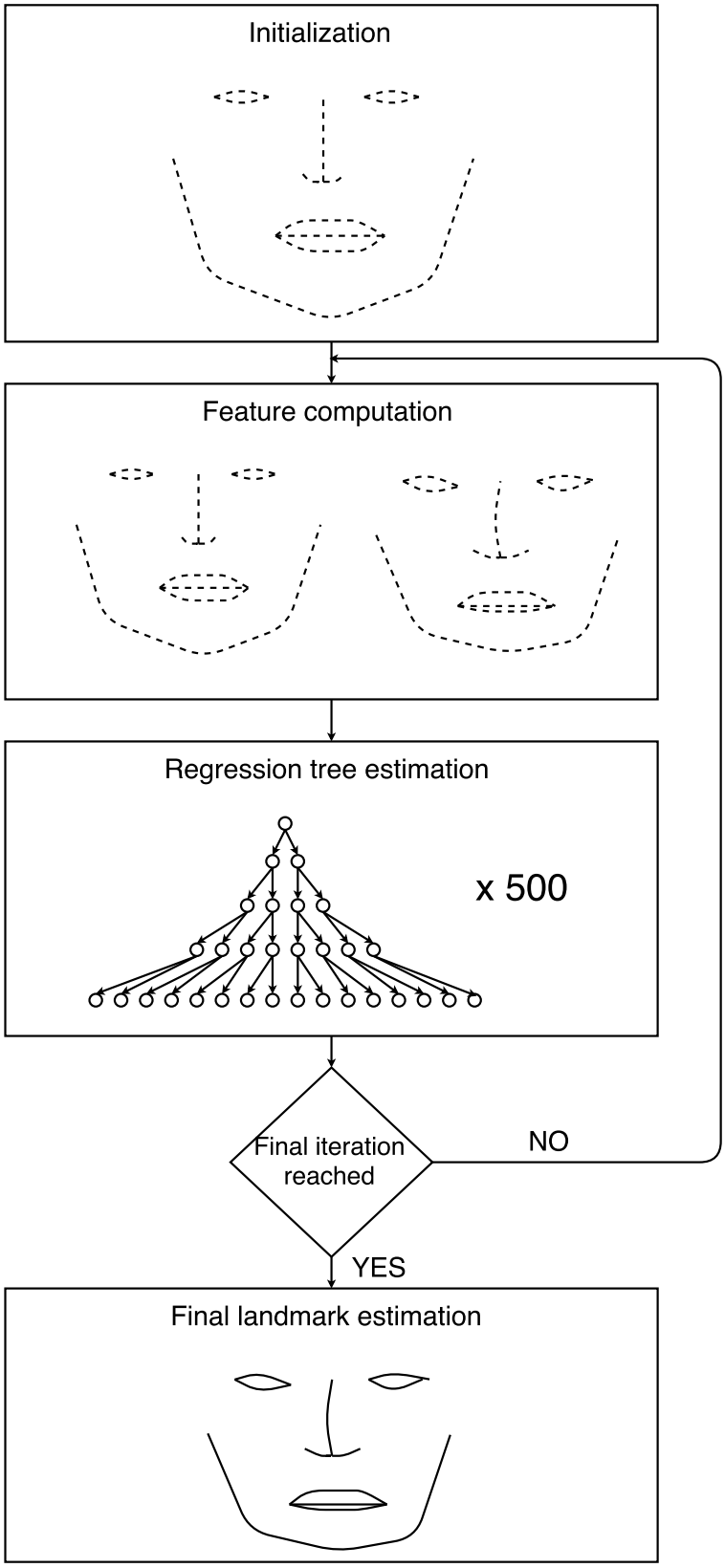


Figure 4.4: Flow graph of landmark-detection algorithm

4.2.2 Algorithm profiling

The algorithm is profiled on ten face images extracted by the face detector to identify the computationally intensive steps. The results can be seen in Figure 4.5.

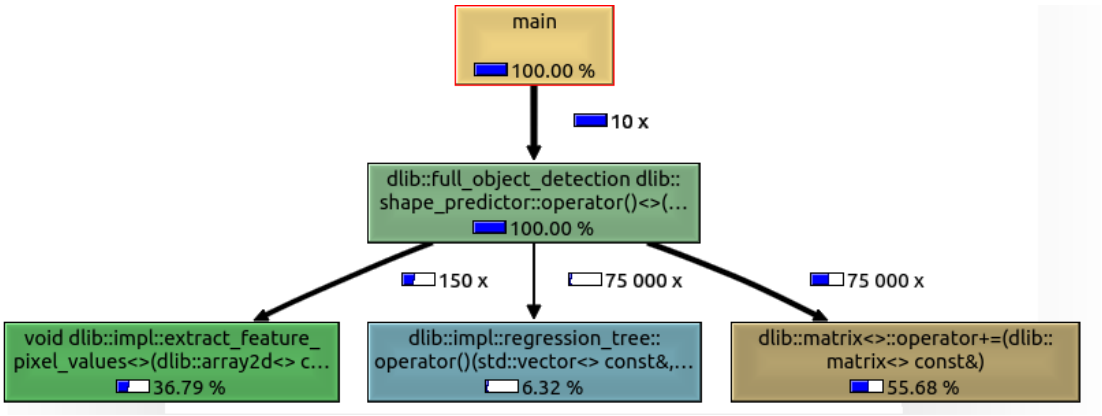


Figure 4.5: Call graph of landmark-detection algorithm. Percentages indicate the amount of time spent in a function compared to the total landmark-detection execution time. Arrow values indicate the number of times the function is called.

The `extract_feature_pixel_values` computes the similarity transform and extracts all feature pixel values required by the regression trees in the current level of the cascade. In `regression_tree()`, the regression trees are traversed and the chosen leave node is returned. Finally, in `matrix::operator+=`, the landmark-shifts estimated by each of the trees are added to the current landmark-shape estimation. The algorithm completes the detection on ten face images in **0.029 seconds**, and is therefore a factor 10 and 200 faster than the face detector with AVX2 instructions enabled and disabled, respectively. In contrast to the face detector, the landmark detector does not benefit from these instructions. Note that the detection time of roughly 3 milliseconds is three times slower than the advertised speed in [7], but the authors do not report the CPU used.

4.2.3 Algorithm acceleration

To speed up the landmark detection, we can look at a number of different parallel models. Although each regression tree can be calculated in parallel, the amount of trees (500) is rather small to benefit from data parallelism on a GPU. Furthermore, the regression trees all update the same global landmark-estimation values which would require a form of atomic additions or a parallel reduction scheme. The same applies to the similarity transform, which requires the calculation of the mean and variance of the landmarks (based on formulas 34-43 of [89]). Synchronization also forms a performance bottleneck, as this is required after each level of the cascade to make sure each regression tree has updated the landmark shape before the features of the next cascade layer are computed. Finally, the landmark-detection algorithm is already fast on a single-core CPU (less than 3 ms per image), which would would make the data transfers to and from an external hardware accelerator relatively costly compared to the potential algorithm

speedup. Therefore, we shall not make use of the data-level parallelism model. Instead, the coarser task-level parallelism model described in Section 4.1.3 is also applicable to the landmark-detection algorithm. Every processing element can perform the landmark detection on its own face image without any required communication with the other processing elements. Furthermore, detecting the landmarks on the CPU will allow for overlap with the face detection on the GPU when using pipelining. We shall implement the task-level parallel model for landmark detection with OpenMP [90], an Application Programming Interface (API) for multiprocessor programming in C, C++ and Fortran.

5 Implementation

In this chapter, we will discuss the implementation details of the HOG algorithm for face detection and ERT algorithm for landmark detection. The GPU implementation of the face-detection algorithm is discussed in Section 5.1, while the multi-core CPU implementation of the landmark-detection algorithm is covered in Section 5.2.

5.1 Face detection on GPU

To accelerate the face detection, the algorithm is modified to make use of a combination of data and task-level parallelism on the GPU. First we will describe the architecture of the used GPU, after which we will discuss the design process of optimizing different kernels and the algorithm as a whole. Kernel performance was analyzed and optimized using the NVIDIA Visual Profiler (NVVP), which is part of the NVIDIA CUDA Software-Development Kit (SDK). The reported kernel times are the average of 3 runs on an image of 640×480 pixels. After each optimization step, the accuracy was verified by manual inspection of 67 face images.

5.1.1 Titan X specifications

To understand the choices made in the implementation phase we take a closer look at the architecture of the used GPU. In this project this is an NVIDIA Titan X (Pascal architecture). Specifications of the GPU can be seen in Table 5.1, and a schematic overview of the chip layout is given in Appendix A.3.

Table 5.1: Specifications of the NVIDIA Titan X (Pascal) GPU, output by the `deviceQuery` binary of the CUDA SDK.

CUDA Driver Version / Runtime Version:	8.0 / 8.0
CUDA Capability number:	6.1
Total amount of global memory:	12187 MBytes
28 Multiprocessors, 128 CUDA Cores/MP:	3584 CUDA Cores
GPU Max Clock rate:	1531 MHz (1.53 GHz)
Memory Clock rate:	5005 Mhz
Memory Bus Width:	384-bit
L2 Cache Size:	3145728 bytes
Maximum Texture Dimension Size (x,y,z):	1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, layers:	1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, layers:	2D=(32768, 32768), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per SM:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 2 copy engine(s)
Run time limit on kernels:	Yes
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Disabled
Device supports Unified Addressing (UVA):	Yes
Device PCI Domain / Bus / location ID:	0 / 41 / 0

This table can be used as reference for certain design optimization choices. Notably, the `deviceQuery` does not output the size of the unified L1/texture cache, but from the CUDA programming guide [2] we learn that this is 48 KB for devices with Compute Capability 6.1.

5.1.2 Optimization techniques

A number of optimization techniques and fundamental ideas behind the GPU architecture will be explained that are used for the optimization of the kernels and the algorithm as a whole. They were explored during the implementation phase and therefore described in this chapter.

The most crucial aspect of maximizing GPU performance is memory optimization [91]. This concerns both the data transfers between the host (CPU) and device (GPU), as well as the memory transactions within the device itself.

Data transfers between the host (CPU) and device (GPU) are done via Peripheral Component Interconnect Express (PCIe) 3.0, which has a peak bandwidth of 15.75 GB/s, and are, therefore, relatively costly. Minimizing these data transfers is important for high performance. Even if not every step of the algorithm demonstrates a speedup on the GPU, it is often preferable to keep it on-board to prevent the extra data transfers. Pinned, or page-locked, memory can be allocated on the host by `cudaMallocHost` and attains the highest bandwidth between host and device. This memory is also required in case of asynchronous data transfers: when data is transfered between host and device while the device is executing kernels.

Minimizing data transfers within the device, to and from the off-chip global memory, is also of key importance. Arithmetic latency is short compared to memory latency (6 to 24 cycles versus 400 cycles [92]), and therefore the algorithm should be optimized in such a way that the number of global memory transactions is minimized, and each transaction contains as much useful data as possible.

A number of caches exist on the GPU that alleviate the need for global-memory transactions and should be made use of as much as possible. The L2 cache is on-chip memory of 3 MB that is shared between the 28 SMs. Each of these SMs also has a unified L1/texture cache of 48 KB. The NVIDIA Pascal tuning guide [93] states that global loads are not cached in this unified cache by default, but this can be enforced by specifying the `-Xptxas -dlcm=ca` flag at compile time. The shared memory cache is shared between the threads of a block, and transactions to and from it have a latency that is roughly 100 times lower than those to and from global memory [94]. This cache can e.g. be used as a user-managed data cache, or to facilitate communication between the threads of a block. Lastly, the constant cache is a read-only cache used to speed up reads from the constant memory, which resides in the device memory. The size of this cache is not specified by NVIDIA. Both the constant cache and shared memory cache are able to broadcast data from a single memory location to multiple threads in a warp.

When global-memory transactions *are* required, it is important to make them as efficient as possible. The concurrent memory requests of all threads in a warp will coalesce into a number of memory transactions equal to the required cache lines. In contrast to previous architectures (Kepler, Maxwell), global-memory transactions are served at a 32 B granularity regardless of whether L1 caching is enabled or not. If the threads in a warp all access adjacent memory addresses, this results in fewer required cache lines than when the addresses are scattered. Maximum efficiency is achieved when all the data loaded from memory was actually requested by the threads, e.g. when all 32 threads in a warp request adjacent 4 B words and a total of 128 B is loaded from global memory. The process of aligning the memory addresses of store and load operations

of all the threads in a warp to minimize the global memory transactions is known as memory coalescing. Vectorized memory access, using CUDA vector data types such as `float2` or `float4`, can further increase the effective memory bandwidth [95].

While consecutive elements of normal arrays are linearly ordered in device memory addresses, CUDA arrays are different data types specifically designed to exploit spatial locality [8]. A cache line fill from these arrays results in fetching data elements that are spatially local in one, two or three dimensions, depending on the dimensionality of the CUDA array. An example of this can be seen in Figure 5.1.

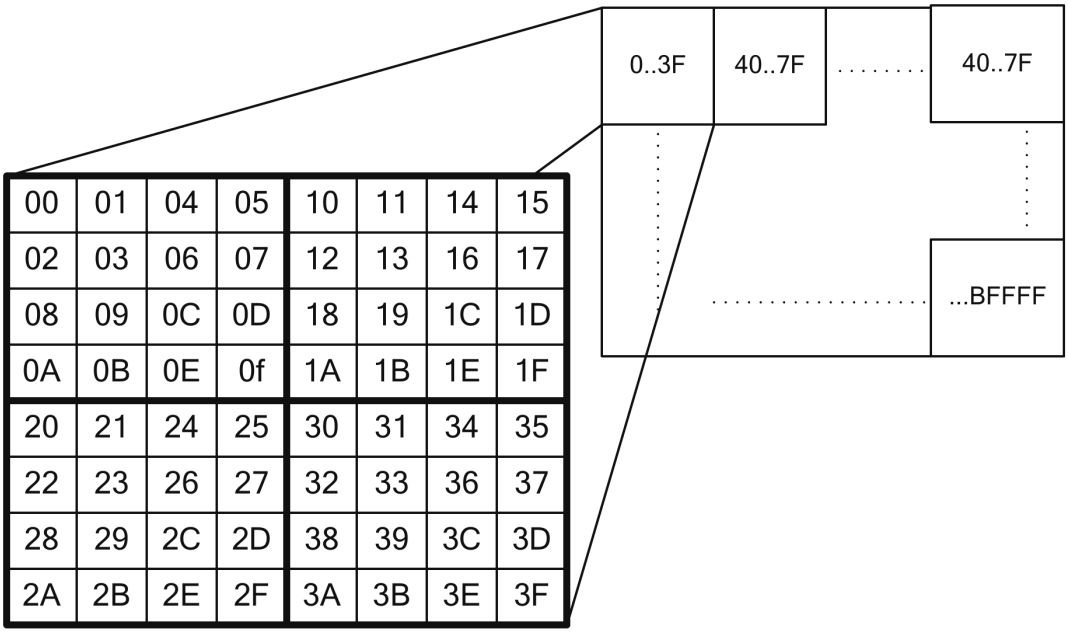


Figure 5.1: Example of the memory layout of a 2-dimensional CUDA array. A cache line fill fetches the data elements from a square block of data elements [8].

These CUDA arrays cannot be accessed by pointers but via texture objects and references. Texture objects have been introduced after the texture references, in the Kepler architecture, and allow for a more flexible way of accessing CUDA arrays while also inducing less overhead [96]. Making use of the GPU texturing hardware, memory accesses to CUDA arrays, using the texture objects and references, come with free bilinear interpolation.

As stated in Section 2.3, GPUs are latency hiding machines. To achieve this, they make use of thread and instruction-level parallelism. When a thread block is assigned to an SM, shared memory and a set of registers for each thread are allocated to it until all threads have completed execution. The warp scheduler selects a warp with instructions that are ready to execute and issues these. While they are being executed, the warp scheduler selects other warps and issues their instructions. Context switching

between warps is very fast because the shared memory and register values don't have to be saved and restored [97]. The more threads there are running on an SM, the higher the chance is that there is a warp with an instruction ready to execute and therefore hide the latency of previous instructions that are being executed. This is referred to as thread-level parallelism. A measure for the amount of possible thread-level parallelism is occupancy: the number of threads (or warps) that runs on an SM compared to the maximum possible number (2048 threads, or 64 warps in our case). The number of registers and the amount of shared memory per thread can be occupancy limiters, since they are assigned to each thread for the duration of the thread block as described above.

An additional approach to latency hiding is the use of instruction-level parallelism. The next instruction of a thread can be issued before the previous one has completed as long as they are not dependent on each other. Loop unrolling is an example of a way to increase the amount of instruction-level parallelism. The two latency-hiding techniques are described in detail in [92] and [29].
We can summarize the device memory optimizations in the following way:

- **Avoid:** Avoid unnecessary transactions to and from the off-chip memory by making optimal use of the available caches.
- **Combine:** Combine the necessary memory transactions by coalescing memory requests of the threads within a warp.
- **Hide:** Hide the high latency memory transactions by making use of thread and instruction-level parallelism.

In a sense this also applies to the data transfers between the host and device, since we should avoid data transfers even though parts of the algorithm don't demonstrate GPU speedup, combine multiple transfers to reduce overhead, and hide data transfers by executing them concurrently with kernel executions.

The following subsections will describe the optimization process of the kernels of the face-detection algorithm, as well as optimizations that affect the performance of the algorithm as a whole.

5.1.3 Image scaling kernel

The first step of the face-detection algorithm is the image downscaling. The image size of 640×480 pixels requires a total of 12 scales to detect faces of all sizes, so the image is downscaled 11 times. Memory for the images is allocated using `cudaMallocPitch` to ensure proper aligning of the image rows in the device memory.

The value of each pixel of the downscaled image is computed by means of bilinear interpolation. Two versions of the kernel were implemented and compared: one making use of the free bilinear interpolation of texture objects and one manually implementing the bilinear-interpolation algorithm.

The number of threads in the grid equals the number of pixels in the downscaled image. Blocks are two-dimensional, with the x and y dimension corresponding to the width and height of the downscaled image respectively.

Each thread computes its pixel's new location compared to the higher scale image by multiplying the x and y coordinate by the scale-factor. This location is surrounded by four pixels in the higher scale image, on which bilinear interpolation is performed to calculate the pixel value in the downscaled image.

Although the use of CUDA arrays in combination with texture objects allows for free bilinear interpolation, there is an overhead in creating the texture objects and binding them to the CUDA arrays. Also, to write to CUDA arrays directly from a kernel, we need to make use of `surface writes`, which requires the creation of Surface objects and binding them to the CUDA arrays. Not making use of the surface objects forces us to write to the linear device memory and then perform a device-to-device copy from the linear memory to the CUDA array, which induces more overhead than the creation of the surface objects. Furthermore, to make use of the bilinear interpolation, the texture object reads the CUDA array elements as normalized floats (this can be specified by a parameter when binding the texture object to the CUDA array). Since the gradient magnitude computation of the next kernel requires unnormalized floats, a conversion is needed. This can either be done "manually" by converting them in the next kernel or by binding another texture object to the CUDA array that can read its elements normally.

The results of the two implementations can be seen in Table 5.2.

Table 5.2: Results of image scaling kernels using texture objects bilinear interpolation and the manual implementation of the bilinear interpolation algorithm.

Implementation	**Time (μs)**
Texture object	**26.914**
Manual bilinear interpolation	30.572

As can be seen the texture object implementation is roughly 10% faster, making use of the free bilinear interpolation of the texturing hardware. This comes at a cost, as the creation and destruction of the texture and surface objects takes 2 to 10 μs each. The creation and destruction of two texture objects and one surface object for all 12 scales results in an overhead much bigger than the kernel execution time itself. However, whenever we are processing a sequence of images with a constant image size, as is the case for our videos, we can re-use the same CUDA arrays and texture and surface objects. This would only require the creation of the objects before the first image and the destruction after the last. Therefore, the texture object implementation for image scaling is used in this project.

Synchronization must occur after each downscaling step (until minimum image size has been reached), because each scaled image is based on the image of one scale higher. The remaining kernels are computed on all image scales, which are completely independent from each other. Synchronization only has to occur between kernels of the same image scale, not in between different image scales.

5.1.4 Gradient and histogram kernel

Once the image downscaling has completed, we can start the feature-extraction process. The first step of this process is to compute the gradient orientation and magnitude of each pixel, and the histograms of each cell.

In this kernel, the number of threads in the grid equals the number of pixels in the image. Blocks are two-dimensional, with the x and y dimension corresponding to the width and height of the image, respectively. Each thread computes the discrete gradient orientation and gradient magnitude of a pixel and contributes to four surrounding cell histograms as described in Section 4.1.

The computation of the gradient requires each thread to load the pixel values of two horizontal and vertical neighbors. There are several ways to approach this. The standard approach makes use of the linear device memory, L2 and L1 cache. However, the spatial locality of the required pixels could benefit from the use of CUDA arrays and texture memory, which is the second approach. Lastly, we can try to manage the data-caching ourselves by making use of the shared-memory cache: Each thread block first loads all the required pixel values in the shared memory before beginning gradient and histogram computations. The results of these different approaches are shown in Table 5.3.

Table 5.3: Comparison of different approaches to loading image data from memory for the gradient computation and histogramization kernel.

Implementation	**Time (μs)**
Linear device memory	211.134
Shared memory	216.766
Texture memory	**209.577**

The spatial locality of the pixels is exploited with the use of texture memory, but the difference with normal linear device memory is small. The chosen approach also depends on the previous kernel. Since the images are already in the texture memory, it makes sense to keep them there since the conversion from CUDA array to linear device memory requires an extra device-to-device memory transfer. The use of shared memory did not result in better loading performance. Therefore, the texture memory implementation is chosen to load the image data in this kernel.

The second possible optimization involves histogram data storage. The histograms of each cell are stored in memory in a *histogram bins - cell rows - cell columns* order; more specifically:

- The values of consecutive histogram bins of a cell are next to each other in memory.
- The histograms of cells in the same row are next to each other in memory.
- The rows, containing the histograms of all cells in the row, of the image are next to each other in memory.

This is visualized in Figure 5.2. This notation will also be used to explain memory layouts in the following kernels.

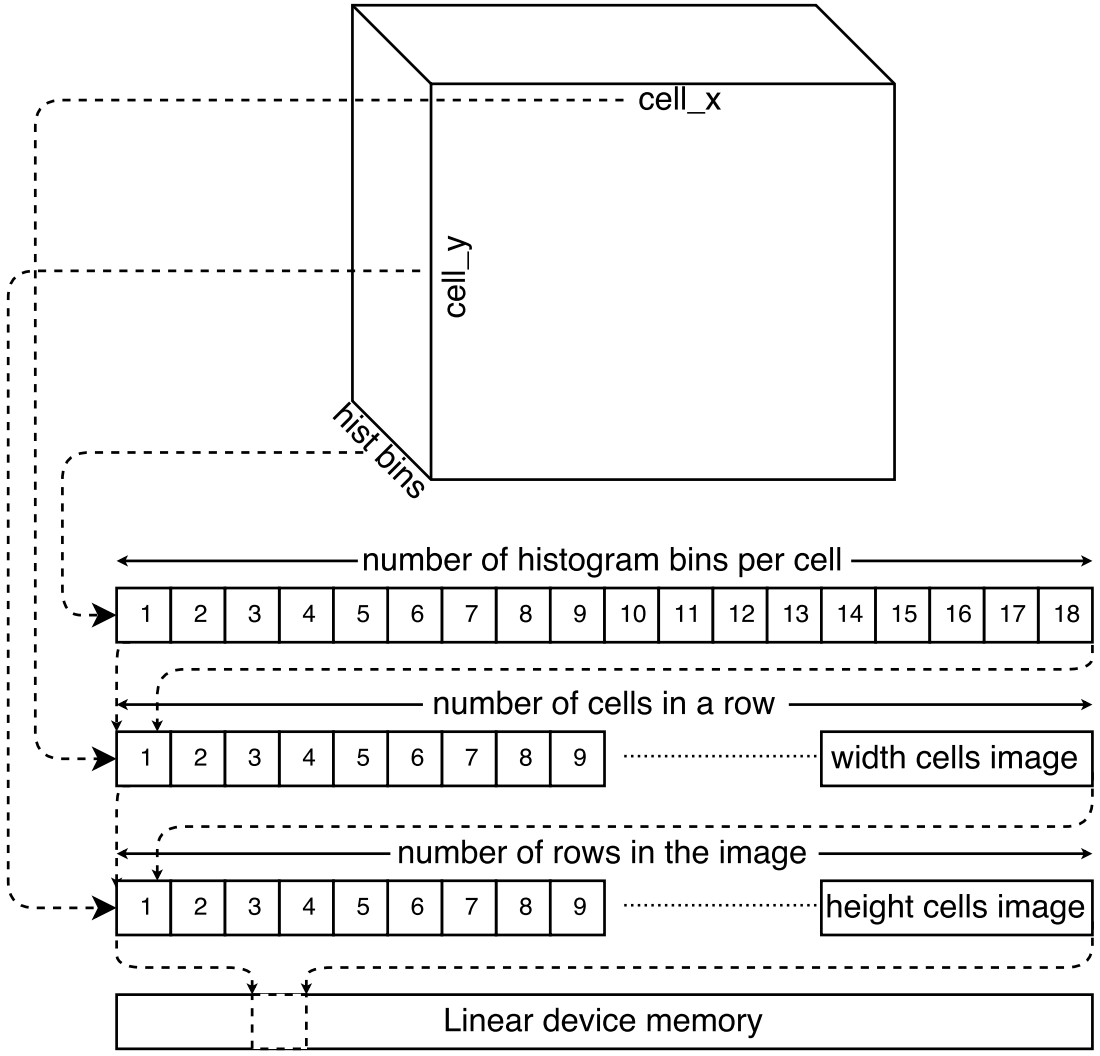


Figure 5.2: The image consists of a two-dimensional grid of cells, each cells represented by a histogram of 18 bins. This three-dimensional object is stored in device memory linearly. We describe this particular layout as a a *histogram bins - cell rows - cell columns* order.

Histogramization suffers from a number of difficulties on the GPU. Firstly, the memory location of the histogram bin to which a thread must write is not known at compile time; it depends on the gradient orientation of the pixel. This causes uncoalesced memory writes. Secondly, multiple pixels might write to the same histogram bins at the same time, which could cause race conditions. To prevent this from happening, we need to make use of atomic operations, which sequentializes writes to the same memory address.

To reduce the number of writes to global memory we can combine the results of the threads in a block in shared memory before writing to global memory [98]. The

following example clarifies this:

Assuming we have a block of 32×32 threads. The threads in this block can write to the histograms of an area of 5×5 cells, as can be seen in Figure 5.3.

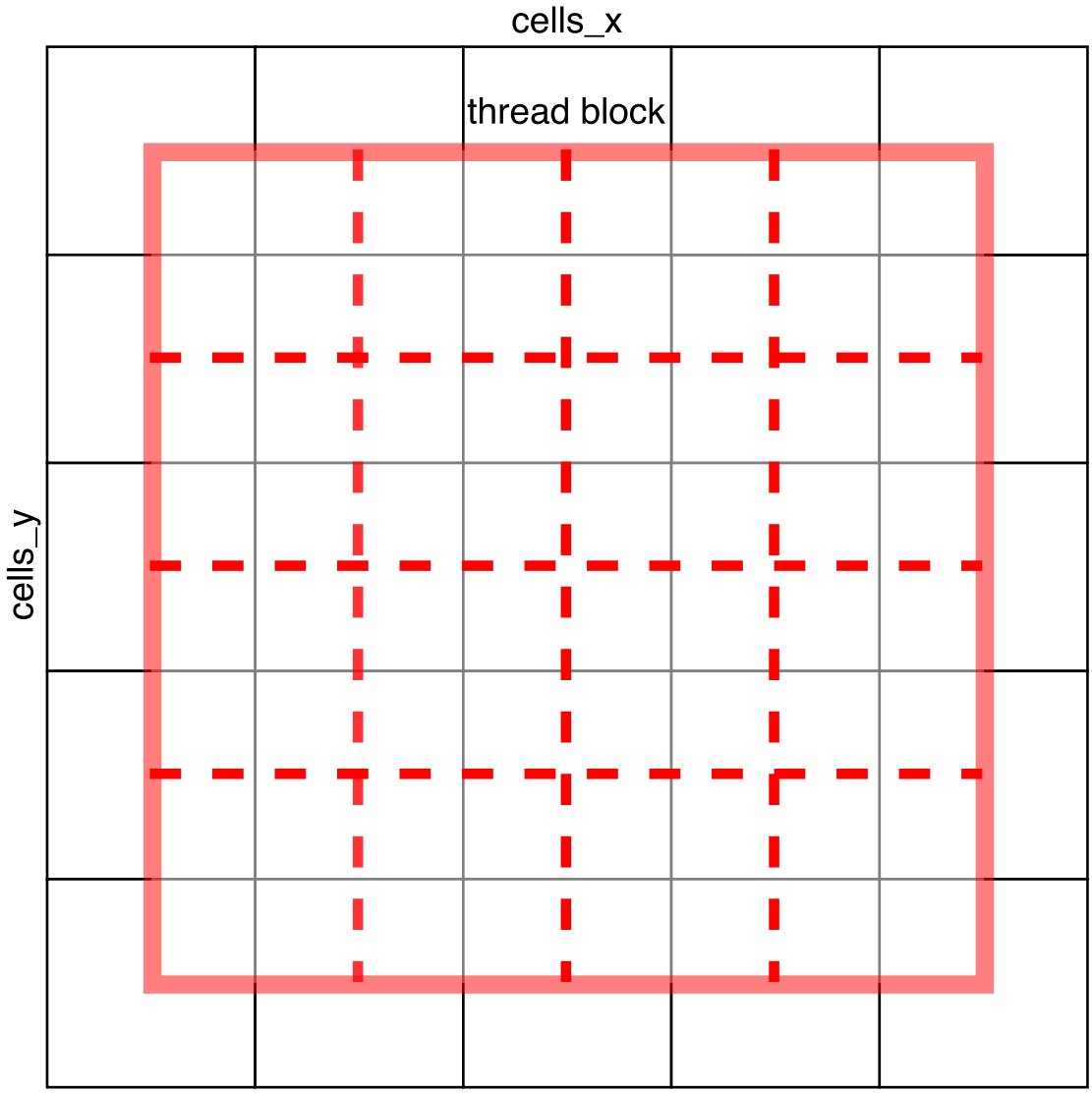


Figure 5.3: An area of 32×32 pixels (within the red borders) can contribute to the histograms of an area of 5×5 cells (the black squares). Each cell is 8×8 pixels. The pixels within the red dotted squares contribute to the four cells that connect within the red-dotted square.

If each of the 1024 threads writes directly to four histogram bins in global memory, this results in a total of 4096 global atomic writes. However, if the results are combined in a shared memory array of 5 *cells* $\times$ 5 *cells* $\times$ 18 *bins* first, a total of 450 global writes remain. Moreover, these writes can now be performed in a coalesced way. The

performance results of the two different approaches can be seen in Table 5.4.

Table 5.4: Comparison of gradient computation and histogramization kernel with histogram values written to global memory directly, or with results first combined in shared memory.

Implementation	**Time (μs)**
Global atomic write	**211.134**
Shared atomic write	374.352

These results are surprising, but can be explained by a difference in implementation of the atomic operations. Atomic operations have improved over the course of NVIDIA GPU architectures and can basically be divided into two categories: hardware implemented and software implemented. Hardware native atomic operations are implemented using single instructions and are much faster than their software counterparts, which make use of lock/unlock semantics [8]. In Pascal architecture, global atomic floating point additions are implemented in hardware, while shared atomic floating point additions are implemented in software [99]. To overcome this problem, we implement our own version of atomic floating point additions in shared memory by making use of the hardware native atomic compare-and-swap (CAS) [100]. As can be seen in Table 5.5, this only results in a small performance increase.

Table 5.5: Comparison of the shared atomic floating point addition making use of compare-and-swap with the previously discussed methods.

Implementation	**Time (μs)**
Global atomic write	**211.134**
Shared atomic write	374.352
CAS shared atomic write	369.147

Unlike floating points addition, atomic 32-bit integer addition *is* hardware implemented in shared memory. If we scale the floating point values by a power of ten and convert them into integers, we can essentially perform fixed-point atomic additions in shared memory and convert them back to floating points afterwards. To determine the maximum scaling factor that can be used without resulting in overflow, we must first determine the range of possible floating point values at this point:

- The horizontal and vertical gradient can have a maximum value of 255 and a minimum value of 0.
- Therefore, the gradient magnitude can have a maximum value of $\sqrt{255^2 + 255^2} \approx 360.62$ and a minimum value of 0.
- There are $4 \times 8 \times 8 = 256$ pixels that can contribute to a histogram. Because of the bilinear interpolation, each pixel contributes on average 0.25 times its gradient

magnitude to a histogram bin. If all 256 pixels have the maximum gradient magnitude value and contribute to the same histogram bin, this results in a maximum value of $256 \times 0.25 \times 360.62 = 23079.68$. The minimum value remains 0.

- The maximum value of a signed 32-bit integer is 4294967295. Because $\frac{4294967295}{23079.68} \approx 186093.02 > 10^5$, we can safely multiple the floating point values by a factor of 10^5 before converting them to integers without the possibility that later additions will result in overflow. This effectively results in a precision of five decimal places.

After the shared-memory atomic additions, the integers are scaled back and converted back to the floating-point data type before writing them to global memory. Table 5.6 shows that this approach results in a significant speedup compared to the approach with direct global memory atomic writes, which is why it as chosen as final implementation for histogram data storage.

Table 5.6: Comparison of the shared atomic fixed point implementation with the global atomic floating point implementation for histogram data storage.

Implementation	**Time (μs)**
Global atomic write	211.134
Shared atomic fixed-point write	**114.436**

After the histogramization, synchronization is required in order to make sure the histograms of every cell are complete before computing their energy values.

5.1.5 Energy kernel

The energy values of each cell are computed in this kernel. The number of threads in the grid equals the number of cells in the image. Blocks are two-dimensional, with the x and y dimension corresponding to the cells in the rows and columns of the image, respectively. Each thread computes the energy by adding the histogram bins of opposing directions together (making an unsigned version of the histogram), and accumulating the squares of these bins:

$$Energy = \sum_{n=0}^{8} (hist\_bin(n) + hist\_bin(n+9))^2$$

Each thread needs to load a total of 18 floating-point values, one for every histogram-bin of the cell. In combination with the *histogram bins - cell rows - cell columns* order memory layout of the histograms, this causes a problem for the coalescing of memory loads:

Assuming a block of 32×32 threads, the 32 threads in a warp compute the energy values of cells that are on the same row of the image. Each of these threads starts with loading the value of the first histogram bin. As can be seen in Figure 5.2, this causes memory loads of the threads within a warp to have a stride of 18. Since memory

transactions are served at 32 B granularity (8 floating points), this reduces the memory load efficiency to 12.5%.

Changing the histogram memory layout to *cell rows - cell columns - histogram bins* order would solve this problem. The first (and second, third, etc.) histogram bins of all the cells would be directly next to each other in the device memory. However, this would have implications for the store efficiency of the previous kernel. Two stages in the implementation process need to be distinguished in order to understand the evolution of the memory loading process of this kernel: before and after the shared-memory implementation of the previous kernel.

The store efficiency of the global atomic write implementation of the previous kernel suffers heavily when changing the memory layout, as the threads in a warp will write to very scattered memory addresses. To see if this is compensated by the gain in loading efficiency of energy kernel, we will compare kernel speed with three possible memory layouts. The feature kernel is also included in this comparison, since it benefits from the same layout as the energy kernel does. The results are shown in Table 5.7.

Table 5.7: Comparison of different layouts of the histogram data in device memory. BRC = *histogram bin - cell row - cell column* memory layout, RBC = *cell row - histogram bin - cell column* memory layout and RCB = *cell row - cell column - histogram bin* memory layout.

Kernel	**Time (μs)**		
	BRC	**RBC**	**RCB**
GradientHistogram	**206.978**	257.729	286.108
Energy	47.979	**23.926**	24.107
Feature	379.751	**344.356**	345.815
Total	634.709	**626.011**	656.030

In the table it can be seen that both the energy and feature kernel benefit from the different memory layout, while the gradient and histogram kernel speed decreases. The *cell rows - histogram bins - cell cols* order memory layout performs best: it suffers less from the scattered memory writes while benefiting fully from the coalesced memory loads. Memory writes are less scattered because the 32 threads in a warp write to four horizontally adjacent cells; writes to horizontally adjacent cells with the same histogram bin value are still combined.

The performance of the *bins – cell rows – cell cols* memory layout is somewhat surprising. Especially the energy kernel performs very poorly; it is roughly twice as slow as with the other two memory layouts. Although the memory loads are not coalesced, we would expect a very high cache hit rate since each thread fetches the values of 8 histogram bins with the first load, which it all eventually needs. Effectively, only 3 loads from off-chip global memory would be needed if the fetched memory segments can all remain in the cache. As it turns out, the problem is the block size of 1024 threads.

The unified L1/texture cache size is 48 KB per SM. If each thread loads 18 floating points, this results in $1024\frac{threads}{block} \times 18\frac{floats}{thread} \times 4\frac{bytes}{float} \approx 72\frac{KB}{block}$, which is too much to keep in the unified cache. This causes loads from the shared L2 cache, instead of the faster private unified cache. Halving the number of threads per block for the energy and feature kernels should result in a significant speedup for these kernels since 36 KB of data is able to remain in the L1 cache. The unified cache hit rate of both kernels can be seen in Table 5.8.

Table 5.8: Unified cache hit rate of energy and feature kernels with varying block sizes for all image scales. Higher scale number means smaller image.

	Blocksize 32×32			**Block size** 32×16		
Scale	**Grid size**	**Cache hit rate (%)**		**Grid size**	**Cache hit rate (%)**	
		Energy	**Feature**		**Energy**	**Feature**
0	6	40.495	27.355	12	86.781	51.747
1	6	51.983	29.413	12	86.625	51.775
2	4	61.968	31.573	6	86.791	51.806
3	4	48.511	28.501	6	86.656	51.695
4	2	42.534	26.040	4	86.570	51.832
5	1	34.589	44.111	2	86.815	51.803
6	1	86.701	51.993	2	86.701	52.015
7	1	86.722	51.81	2	86.722	51.81
8	1	86.646	51.743	1	86.646	51.743
9	1	86.428	51.983	1	86.428	51.983
10	1	86.387	51.912	1	86.387	51.893
11	1	86.237	52.158	1	86.237	52.105

Halving the block size results in the expected increase of the unified cache hit rate. To see the effect on the kernel running times we compare the three memory layouts again with halved block sizes. The results are in Table 5.9. Note however, that this is problem size dependent. The grid size is mentioned because as long as the number of blocks is less than the number of SMs in the GPU, multiple blocks will not be allocated to the same SM and therefore don't have to share their unified cache with other blocks. Increasing the image size will cause the number of blocks to increase, and from a certain point cause the cache hit rate to go down again.

Table 5.9: Comparison of different layouts of the histogram data in device memory with block size of 512 threads. BRC = *histogram bin - cell row - cell column* memory layout, RBC = *cell row - histogram bin - cell column* memory layout and RCB = *cell row - cell column - histogram bin* memory layout.

Kernel	**Time (μs)**		
	BRC	**RBC**	**RCB**
GradientHistogram	**208.521**	251.836	278.935
Energy	24.673	20.439	**20.246**
Feature	228.201	**224.618**	228.042
Total	**461.395**	496.893	527.223

This shows a more expected behavior. The performance of the energy and feature kernel of the original *bins – cell rows – cell cols* memory layout is now comparable to the other two, while the gradient and histogram kernel performance is still superior. Because we are dealing with a problem size small enough to guarantee the high unified cache rates, we shall opt for this memory layout at this stage. Furthermore, it can be noted that especially the feature kernel has benefited from the reduced block sizes, regardless of the memory layout. This is because there are more SMs are utilized that would otherwise be idle. The block size of each kernel shall be covered later in Section 5.1.13.

To see whether the histogram data loading speed can be improved any further, two additional implementations were investigated. One makes use of vector data types (`float4`, `float2`, [95]), the other one of texture memory loads. The results are shown in Table 5.10.

Table 5.10: Comparison of kernel speeds with linear device memory vector loads and texture memory loads. LDM is linear device memory, TM is texture memory.

Kernel	**Time (μs)**			
	LDM	**LDM float4**	**LDM float2**	**TM**
Energy	24.673	35.138	**21.750**	28.555
Feature	228.201	226.213	**220.575**	221.192
Total	252.874	261.351	**242.325**	249.747

Usage of the `float4` vector data type requires loading 20 (5×4) floating points per thread, and selecting the right 18, which outweighs the performance gains of increased bandwidth usage. These additional steps are not required for the `float2` data type (9×2), which is the best performing method. The performance of the texture memory approach is comparable with the linear device memory approach but induces an additional device-to-device memory transfer overhead by copying the histogram values to a CUDA array.

All these optimizations however, were performed *before* the shared memory implementation of the previous kernel. We can use this shared memory as a coalescing buffer to keep the storage efficiency of the gradient and histogram kernel, while changing the memory layout of the histogram data to allow for coalesced memory loads in the energy and feature kernels. In Table 5.11 the results of this implementation are shown for both 32×32 and 32×16 block sizes.

Table 5.11: Comparison of kernel performance for different memory layouts of the histogram data and different thread block sizes, using the shared memory in the gradient and histogram kernel as coalescing buffer. BRC = *histogram bin - cell row - cell column* memory layout, RCB = *cell row - cell column - histogram bin* memory layout.

Kernel	**Time (μs)**			
	32×16 **block size**		32×32 **block size**	
	BRC	**RCB**	**BRC**	**RCB**
GradientHistogram	**112.784**	113.710	112.858	112.975
Energy	24.332	**20.054**	46.851	23.949
Feature	229.664	**27.796**	378.768	345.263
Total	366.780	**361.560**	538.477	482.187

Note that the kernel performance of the *histogram bin - cell row - cell column* memory layout decreases more for the larger block size (32×32), because the size of the unified cache is too small to contain the histogram elements of all threads in the block. This cache-size dependency is avoided with the *cell row - cell column - histogram bin* memory layout, while maintaining gradient and histogram kernel performance by using the shared memory as coalescing buffer. Therefore, this memory layout is chosen for the histogram data.

5.1.6 Feature kernel

In this kernel, the final 31 features per cell are computed from the histogram and energy values, as described in [4]. The number of threads in the grid equals the number of cells in the image. Blocks are two-dimensional, with the x and y dimension corresponding to the cells in the rows and columns of the image, respectively. Each thread loads the energy value and 18 histogram values of its own cell, and the energy values of its 8 neighboring cells. The loading of the histogram values is already discussed in the previous section. Furthermore, energy values are stored in the same order in the energy kernel, as they are loaded in the feature kernel. Therefore, these memory transactions are coalesced.

The storage of the features in linear device memory is performed in *cell rows - cell columns - features* order for coalesced memory storage, which is visualized in Figure 5.4. This puts a constraint on the thread block layout of the classification kernel. Threads

in a warp should perform the filter computation on the same feature of horizontally neighboring cells, in order to assure coalesced loading of the features (as opposed to, for example, different features of the same cell).

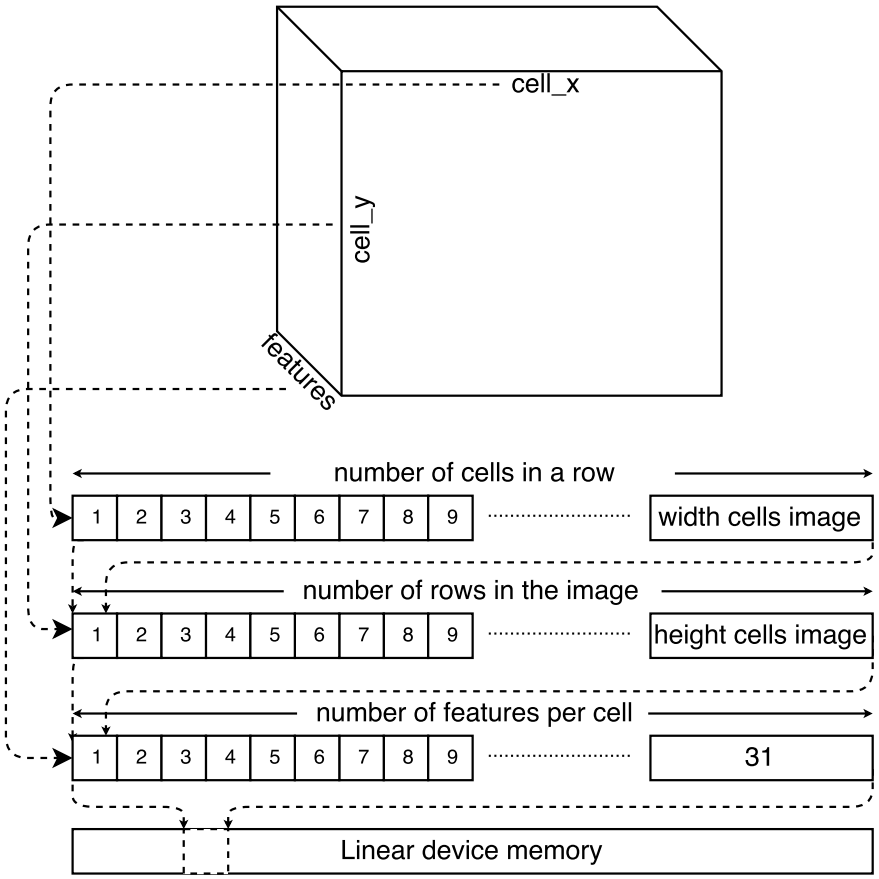


Figure 5.4: The image consists of a two-dimensional grid of cells, each cells represented by a total of 31 features. This three-dimensional object is stored in device memory linearly. We describe this particular layout as a *cell rows - cell columns - features* order.

After the feature kernel, the feature extraction process is complete. The image is divided into cells of 8×8 pixels, each of which is described by a total of 31 features. This has been done for multiple downscaled versions of the original image.

5.1.7 Classifier kernels

The classifier kernels multiply the the feature image with the classification filters to calculate the face score. This is done in two steps: row, and column filter multiplication. As mentioned before, we have five detectors for different rotation angles of the head, so this process is repeated five times for different filter values. A flow diagram of the process from feature image to saliency image (an image containing face score of every cell) is shown in Figure 5.5.

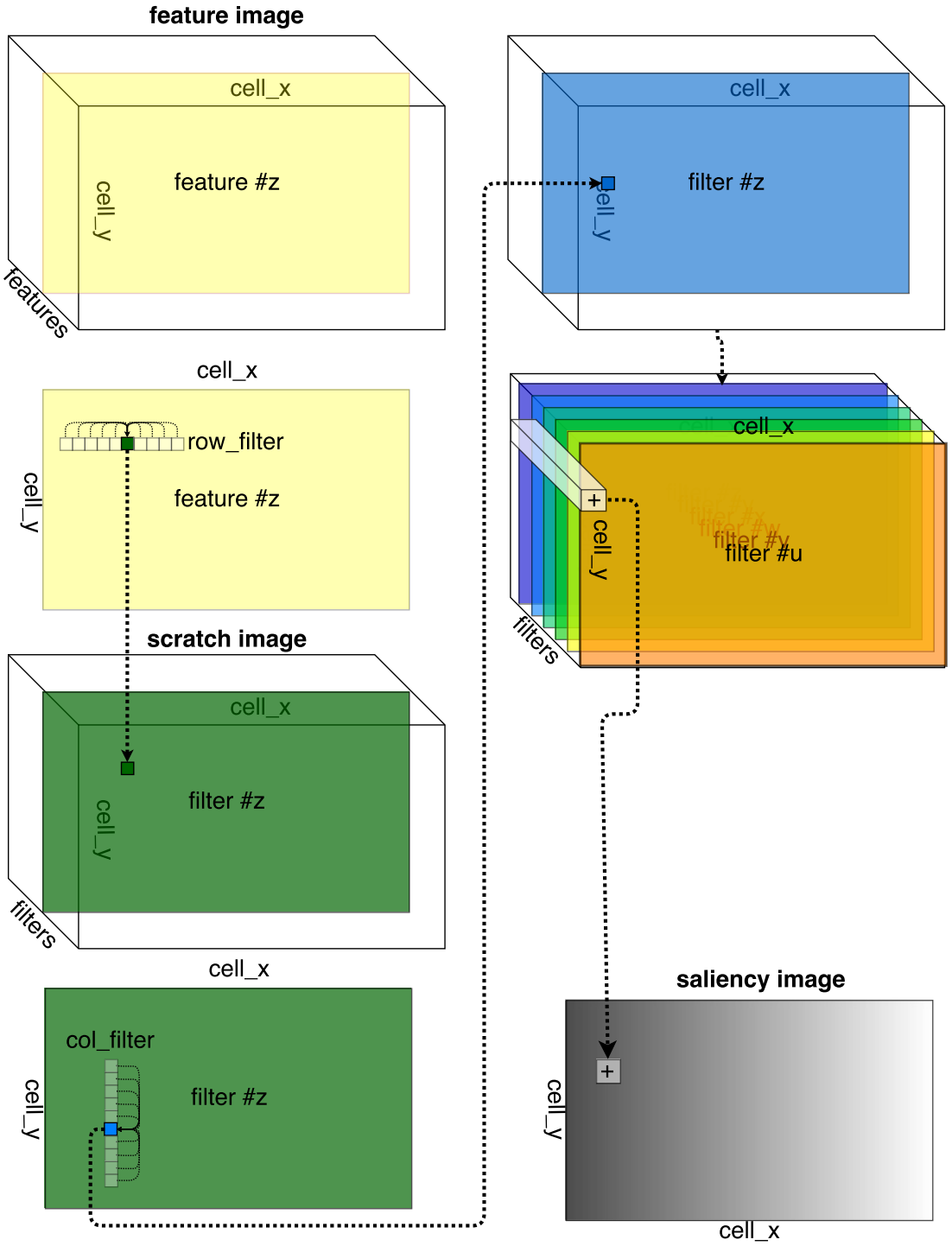


Figure 5.5: The feature image contains 31 features per image cell. A row filter is applied on each feature of every cell. It multiplies the value of a particular feature ('z' in the image) of 10 horizontally neighboring cells by a set of weights and stores the accumulated value. Next, the column filter is applied, which does the same thing for the 10 vertically neighboring cells. Each feature of every cell of the image is now the weighted accumulation of that feature of an area of 10×10 neighboring cells. The features of each cell are accumulated to result in the saliency image.

These filters are the same for every image, regardless of their size or specifics. Therefore, we only have to send them from the CPU to the GPU once, and they remain there unchanged until all the images a user wants to analyze have been completed. This calls for the use of constant memory. Constant memory resides in the linear device memory but makes use of a special read-only constant cache that has the ability to broadcast words. This is efficient when multiple threads of a warp have to access the same filter value simultaneously.

The description of the filter multiplication in Section 4.1 and Figure 5.5 has been a slight simplification of the real implementation. Instead of every feature getting multiplied by a single row and column filter, some features get multiplied by two or three row and column filters. Furthermore, this is not the same for the different face detectors. For example, while feature #17 gets multiplied by two row and column filters for face detector #1, it might get multiplied by one or three row and column filters for the other four detectors. The total number of filters for the detectors of each rotation angle is 61, 58, 54, 66 and 66 respectively. Since there are 31 different features, this results in an average of roughly 2 filters per feature (per detector).

Since synchronization is required between the multiplication of the row and column filters, we divide the process into two kernels. Only after all the row filters have been applied, we can start with the column filters.

5.1.7.1 Row filter kernel

For the filter multiplication kernels the thread grid is three-dimensional. As has been previously discussed and shown in Figure 5.4, the features are stored in *cell rows - cell columns - features* order. This means that the x and y dimensions of the grid should correspond to the cells in each row and column of the image, respectively, for coalesced memory loads. For the size of the z dimension there is a choice between the number of filters and the number of features.

Choosing the number of filters as the size of the z dimension allows for the creation of more threads and a finer level of parallelism. Each thread multiplies a single filter to the same feature of 10 horizontally adjacent cells. To see which filter corresponds to which feature, we make use of a lookup table in constant memory. The drawback of this method is the difference in number of filters that need to be applied to each feature. If a feature is multiplied by more than one filter, and each thread only computes one filter, this means the same feature values are loaded by multiple threads. This causes more expensive memory loads in total.

In contrast, choosing the number of features as the z dimension of the grid prevents these extra memory loads. Each thread applies a number of filters (one, two or three) to the same feature of 10 horizontally adjacent cells. To see how many, and which, filters are applied to each feature we make use of a lookup table in constant memory once again. This approach results in a coarser level of parallelism as each thread is given

more work.

The choice of grid layout boils down to a trade-off between extra memory loads and a coarser level of parallelism. The coarser parallelism is chosen over the extra memory loads, since memory loads are high latency operations that should be avoided as much as possible. Therefore, the number of threads in the grid equals the number of cells in the image multiplied by the number of features per cell. The x, y and z dimensions of the grid correspond to the number of cells in each row, the number of cells in each column and the amount of features per cell, respectively.

Choosing the block size correctly is important for the performance of this kernel. Each thread of a warp must request the same filter value simultaneously to exploit the broadcasting ability of the constant cache. Furthermore, if each thread of a block is working on the same feature, we can assure that they all need to apply the same amount of filters in total. This assures that the workload is balanced and divergence between threads of a warp and warps in a block is minimized. To accomplish this, the z dimension of each thread block is set to one.

In a first, naive, approach, separate kernels were launched for all five detectors. However, since the same feature values are loaded to multiply with different filters for each detector, these kernels were combined into one, to reduce the number of memory transactions. Therefore, in the current implementation, each thread loops over the five detectors. For each detector, it refers to the lookup table to see which filters need to be applied to the feature. The results for each detector are stored in three-dimensional scratch images, with a *cell rows - cell columns - filters* memory layout. Synchronization is required before the column filters can be applied to these scratch images. Results of this kernel can be seen in Table 5.12.

5.1.7.2 Column filter kernel

In this kernel, each thread multiplies ten vertically neighboring cells from the same filter in the scratch image with a column filter, and accumulates its results. In contrast to the row filter kernel, the scratch image values each thread needs to load now depend on the detector (a different scratch image has been created for each detector in the previous kernel). This means that the kernels can not be combined and need to be launched once for every detector (on every image scale). The number of threads in the grid equals the number of cells in the image multiplied by the number of filters per cell. Blocks are three-dimensional, with the x, y and z dimension corresponding to number of cells in each row, the number of cells in each column and the number of filters per cell respectively. The face score of each cell is calculated by atomically adding the filter scores for each cell.

Comparing the results in Table 5.12 with the results from previous kernels, it can be seen that this kernel is relatively computationally expensive. Above all, this is because it is invoked five times as many as the previously discussed kernels.

Table 5.12: Execution time of row and column filter kernels

Kernel	**Time (μs)**
Row filter	143.049
Column filter	484.149

5.1.7.3 Combining row and column filter kernels

The row and column filter kernels are computationally intensive kernels that take up a large part of the face detection time. The synchronization step in between the two kernels causes that the scratch image values calculated by the row filter can't be used directly, but have to be stored in, and loaded from, global memory in order to be used by the column filter. This memory transaction overhead can be avoided by making use of shared memory, block synchronization and performing redundant computations. This is presented in detail in Figure 5.6.

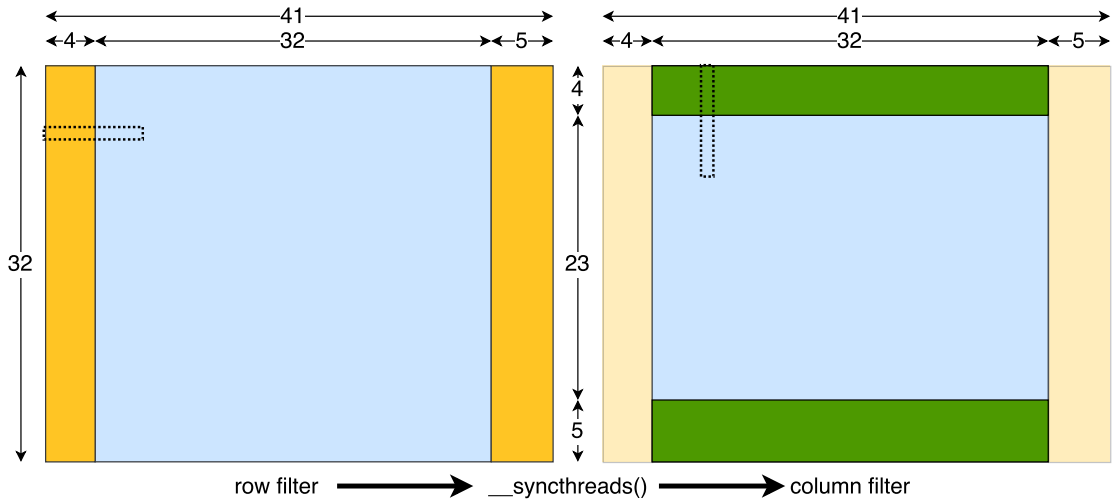


Figure 5.6: Consider a thread block of $32 \times 32 \times 1$ threads for the row filter kernel. These threads need an area of 41×32 values of a particular feature to calculate the values for the scratch image. Each thread stores its scratch image value in shared memory instead of global memory. Block synchronization with `__syncthreads()` is performed to ensure that every thread has written its value to shared memory before the row filter computation begins. We can apply the column filter to an area of 32×23 without any communication with other thread blocks.

This comes at the cost of redundant computations, as two vertically adjacent thread blocks overlap in an area of 32×9 cells in order to compute the column filters for

all the cells without inter-block communication. This means we increase column filter computations by 28.125% in order to merge the row and column filter kernels, hereby preventing excessive synchronization and global memory transactions. As can be seen in Table 5.13, this greatly reduces the classification kernel execution times. In addition, the filter values are loaded using `float2` vector data types which further improves the kernel performance.

Table 5.13: Comparison of row and filter kernel implementations. The separate implementation makes use of a separate row and column filter kernel, which requires global memory transactions and device synchronization. The combined implementation merges the row and column filter kernels, by making use of shared memory and block synchronization. The combined vector implementation optimizes the loading of filter values by making use of `float2` vector data types.

Kernel	**Time (μs)**		
	Separate	**Combined**	**Combined vector**
Row filter	143.049	-	-
Column filter	484.149	-	-
Combined filters	627.198	371.471	**340.493**

5.1.8 Detection kernel

The detection kernel compares the face scores of every cell of a saliency image with a threshold. The kernel is launched five times per image scale; once for each of the five detectors.

The total number of threads is equal to the number of cells in the image. Blocks are two-dimensional, with the x and y dimension corresponding to number of cells in each row and column of the image, respectively. If the face score is higher than the threshold, the thread writes its cell location, which corresponds to the location of a face in the image, to an array in device memory. When all the launched detection kernels have finished, this array is transferred from the GPU back to the CPU. The execution time of the detection kernel 77.344 μs.

5.1.9 Non-maximum suppression

Non-maximum suppression is a technique to reduce multiple detections that arise from the same object to one. When two detections overlap more than 50%, the highest scoring detection is selected and the other one removed. Since the number of total detections per face is often low (in the order of 1 to 10), this final step does not fit the data parallel model of the GPU and is therefore performed on the CPU.

5.1.10 Image sequence with constant size

The design until this point focuses on the face detection performed on a single image. Memory on the GPU is allocated for the image, multiple arrays containing intermediate feature representations of the image, saliency images and the final face detections. After the face detection on an image has been completed, all the allocated memory is freed. Since we are dealing with a sequence of images in this project, this means this process is repeated for every image in the sequence. However, since we know that the images in the sequence all have the same size, the allocated memory can be reused. This means that memory allocation and deallocation only have to be done once at the start and end of the image sequence, respectively. In Table 5.14, we can see the effect of the reduced number of memory allocations and deallocations on a sequence of 675 images of the same eye blink response video.

Table 5.14: Comparison of memory allocation and deallocation implementations. The variable size implementation allocates and deallocates memory for every image in a sequence. This makes it suitable for image sequences that vary in size. The constant size implementation assumes that every image in the sequence has the same size and only allocates and deallocates memory once.

Implementation	Time (s)	Time per image (ms)
Variable size	5.631	8.341
Constant size	1.904	2.820

Furthermore, all the images in the sequence are 640×480 pixels in size, while the required face size is minimally 20% of the image, as stated in Section 3.1. For the biggest image scale, this results in a minimum face size of 128×96 pixels. Scaling the image down one time results in a minimum face size of 107×80 pixels, which is still larger than the detection window of 80×80 pixels. This means that we can skip the detection process on the biggest scale of the image without risking to miss faces that should have been detected. Skipping the biggest scale reduces the total kernel running time per image as can be seen in Table 5.15.

Table 5.15: Comparison of total kernel execution time on a single image when analyzing the image on all scales, or all but the biggest scale.

Implementation	Total kernel time (μs)
All scales	865.820
Skip biggest scale	**709.764**

5.1.11 Streams

Each image is analyzed on multiple scales in order to make the detection method more scale-invariant. For smaller image scales, the number of image cells is low, which results in a low number of threads for the energy, feature and detection kernels in particular.

This could be seen previously in Table 5.8, where the number of blocks was not enough to occupy the 28 available SMs. This means during the execution of these kernels, some SMs are idle. To prevent this from happening, we can make use of multiple streams: independent queues of work on the GPU [101]. While kernels in the same stream are executed in a first in, first out order, execution of kernels in different streams can overlap.

Remember that each of the previously described feature-extraction and classification kernels are applied to each scale of the image. While kernels that are applied to the same scale should be executed sequentially, kernels from different scales are completely independent from each other. Therefore, we can place the kernels of each scale in its own GPU stream. Whenever resources are available, the GPU can schedule a kernel from one of these streams to be executed. In case of a smaller kernel, that does not employ enough threads to keep all SMs of the GPU occupied, the execution of multiple kernels from different streams can be combined. The results of this implementation are compared to the single stream implementation in Table 5.16.

The improvement over the single stream implementation is not as much as was expected. To see why, we take a closer look at the NVVP output in Appendix A.4. It turns out that concurrent execution of multiple kernels only happens sparsely, even though there are plenty of smaller kernels to be combined. The reason for this is that the CPU is not able to add work to the streams fast enough, in order to build up the available work for when GPU resources are available. Launching a kernel takes roughly 5 to 10 μs, while some of the smaller kernels have an execution time as little as 1 or 2 μs. During the execution of larger kernels, the CPU is able to build up the available work, which can be seen by the concurrent kernel execution after some of the larger kernels. Furthermore, the CPU is also busy issuing `memset` commands to the GPU. `Memset` is used to set all the values of an array to 0.

The first step towards more kernel concurrency is, therefore, to free the CPU from any tasks besides launching kernels. To achieve this, all the arrays in the device memory, except the ones with the original (scaled) images, are combined into one, larger array. A lookup table is kept in constant device memory, which states at which address each of the original smaller arrays begins. This combining of arrays is done so that all the arrays can be set to 0 at the start of each new image by one single `memset`. The disadvantage of this single array implementation is that we can not make use of `cudaMallocPitch` for pitched device memory anymore, which is the reason this is not covered in the previous kernel implementation sections.

Furthermore, to reduce the total number of kernel launches, each thread in the detection kernel now loops over the five different detectors inside the kernel, instead of launching a separate kernel for each detector. In Table 5.16, we can see a big improvement over the previous implementation with streams. However, judging from the NVVP output, the CPU is still not able to launch kernels fast enough to fully exploit the concurrent kernel execution potential of the GPU.

Table 5.16: Comparison of single-stream and multiple-stream implementations on a single image of 640 × 480 pixels. The multiple-stream implementation uses a total of 11 streams, corresponding to the number of scales on which the face detection is performed. The reported time is the time from the start of the first kernel to the completion of the last kernel. This is done because the NVVP accumulates the individual kernel times even though they overlap in different streams.

Implementation	**Time (ms)**
Single stream	1.744
Multiple streams	1.527
Multiple stream, memset and detection combined	**0.520**

5.1.12 Image combinations

Another approach to increase the concurrent kernel execution is to let the GPU process multiple images at the same time. If one CPU thread is not able to launch the kernels fast enough, we can employ more CPU threads to increase the kernel launch throughput. The `-default-stream per-thread` NVIDIA CUDA Compiler (NVCC) option lets every CPU thread add work to its own default GPU stream. Each of the CPU cores loads its own image from memory, adds work to its own GPU stream and collects the face detection results afterwards. The implementation is visualized in Figure 5.7.

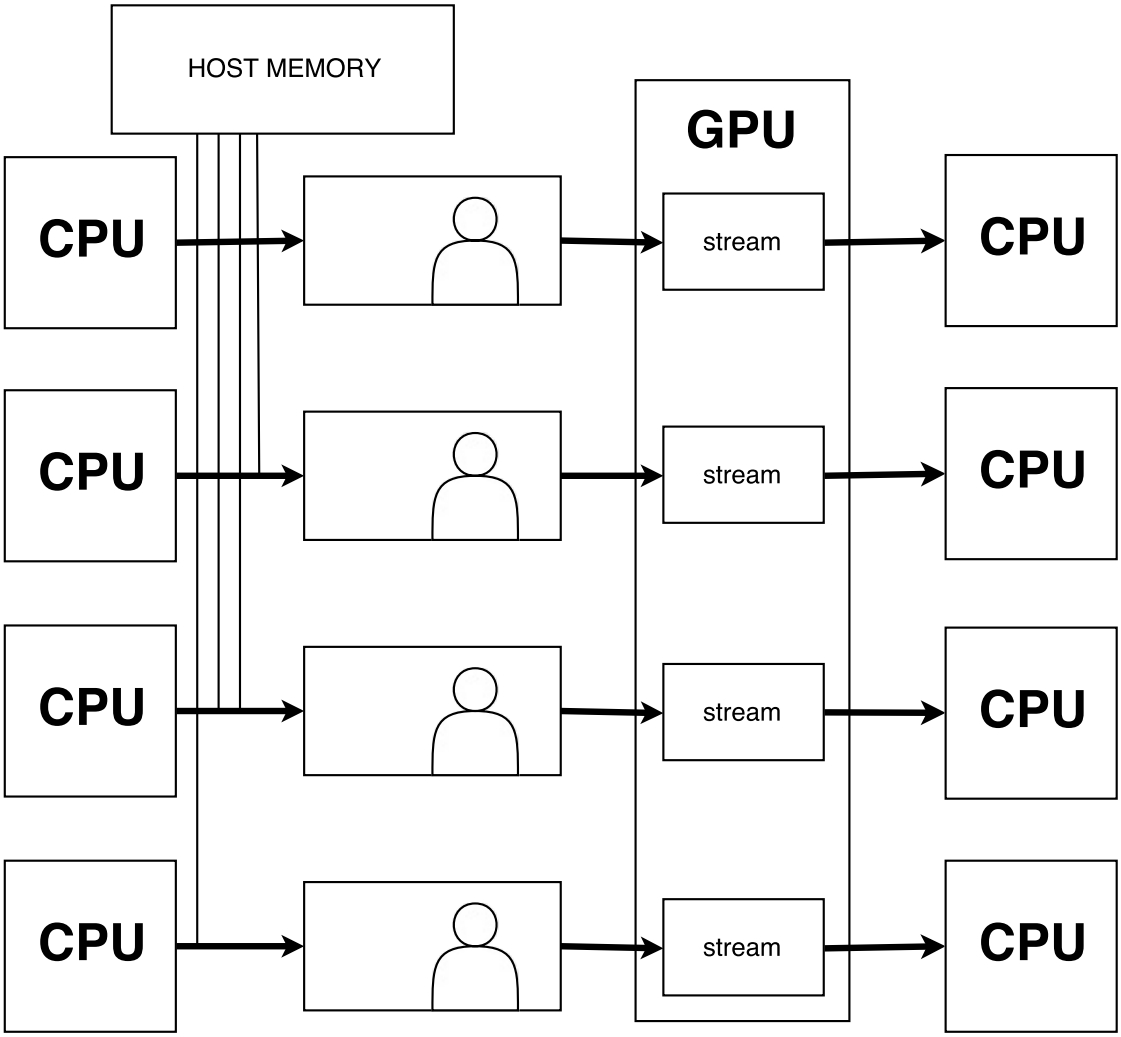


Figure 5.7: Face detection implementation where each CPU thread launches kernels to its own GPU stream.

The results for 4 and 16 CPU threads are shown in Table 5.17.

Table 5.17: Time from the start of the first kernel until completion of the last kernel, for implementation where multiple CPU threads launch kernels in their own GPU stream.

CPU threads	**Total time (ms)**	**Time per image (ms)**
4	3.580	0.895
16	18.607	1.163

Surprisingly, the execution time per image has increased compared to the single-CPU-thread implementation. From the NVVP output we can read that this approach increases the kernel launching time on each CPU thread. For example, the kernel launching times

of the 4-CPU-threads approach increase to approximately 50 μs, compared to the 5 to 10 μs of the original single-thread approach. If we employ more CPU threads to launch kernels, but each launch takes much longer to complete, effectively nothing has been gained. Therefore, we stick with the single-CPU-thread implementation.

If the kernel-launch throughput can not be increased any further, it means we must increase the execution time of each kernel. If the ratio of kernel launch time to kernel execution time becomes smaller, the amount of available work on the GPU can build up and kernels can be executed concurrently if resources are available. There are two ways to achieve this:

- Increase the amount of work of each thread while maintaining the same number of threads.
- Increase the amount of threads while maintaining the amount of work of each thread.

The first approach is achieved in the following way: A number of CPU threads, W, load an image from memory in an array that is shared between them. One CPU thread sends the array, which contains multiple images, to the GPU. Each of the GPU kernels is modified with an outer `for-loop`, which loops over the complete original kernel code, and executes it once for every image. Since each kernel now performs computations on W images, the total number of kernel launches for the complete image sequence is divided by W, while each thread is given more work. This greatly reduces the kernel-launch time to kernel-execution time ratio. A schematic overview of this implementation is shown in Figure 5.8.

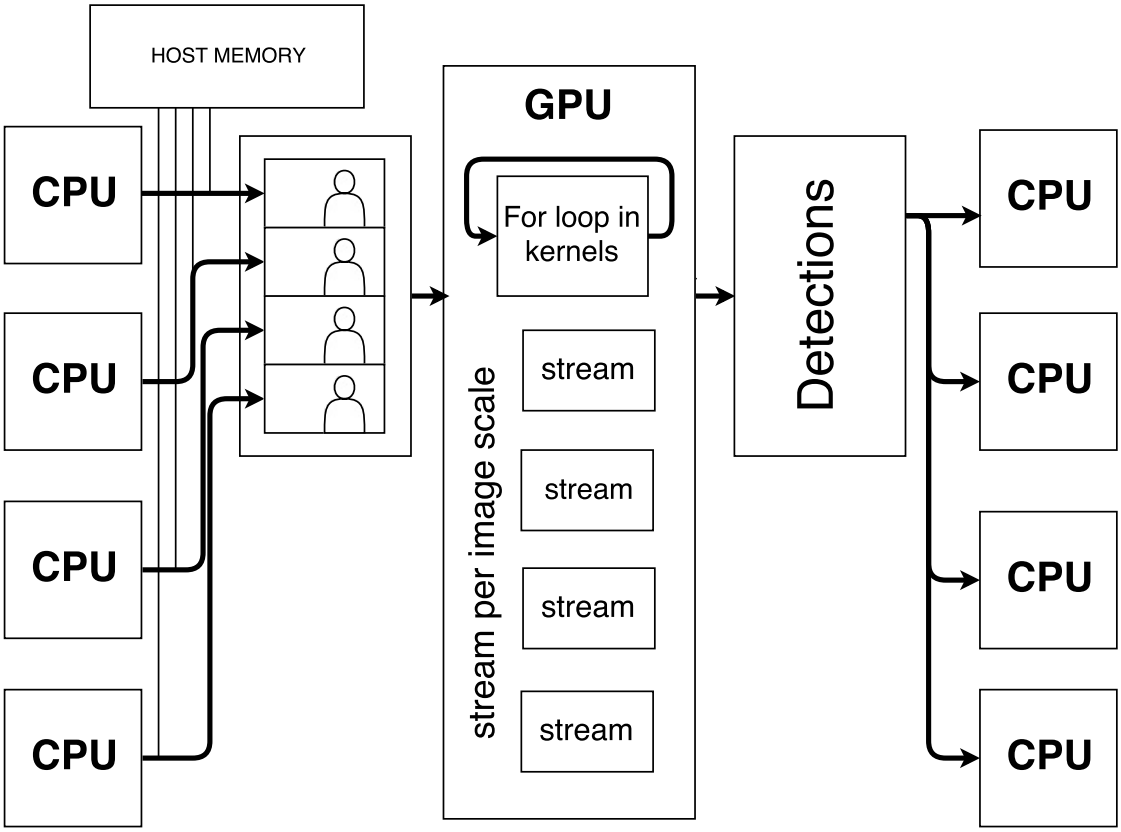


Figure 5.8: Face detection implementation where each kernel in the GPU loops over multiple images.

The downside of this method is that it increases the number of used registers for each kernel, which limits the occupancy of especially the feature and classification kernels. The results for 4 and 16 CPU threads can be seen in Table 5.18.

Table 5.18: Time from the start of the first kernel until completion of the last kernel, for implementation where each GPU thread loops over multiple images inside the kernel.

Number of images	**Total time (ms)**	**Time per image (ms)**
4	1.690	0.423
16	6.365	0.398

To increase the amounts of threads while maintaining the same amount of work per thread, two different solutions have been investigated. The first solution combines multiple images into one, larger image containing multiple faces. A number of CPU threads, W (a power of two), loads an image into a shared array, which is send to the GPU by a single thread. The GPU performs face detection on this combined image as if it is any other image. The only difference is, that it is now a factor $\frac{W}{2}$ wider and higher, which is also the reason the amount of threads is increased. The GPU returns an

array of detections, and based on their location in the combined image, we can can tell to which of the original images they belong. This approach is visualized in Figure 5.9.

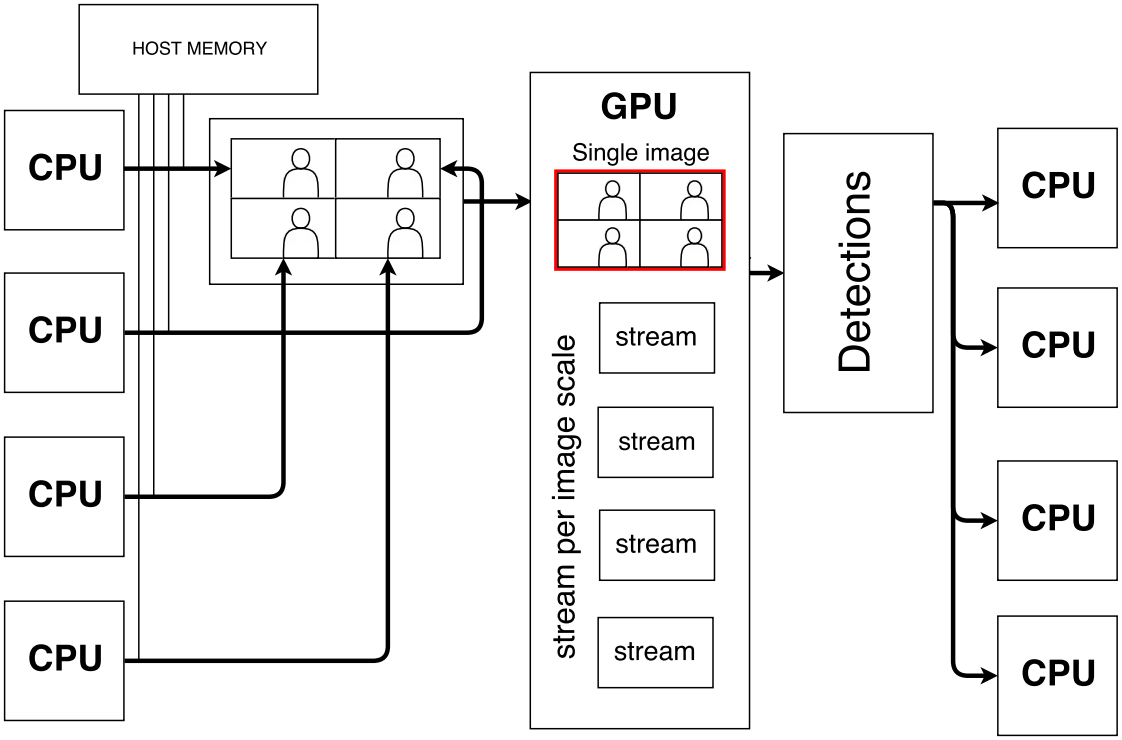


Figure 5.9: Implementation where face detection is performed on one larger image that consists of multiple combined original images.

However, this approach results in a loss of face detection accuracy whenever a face in the combined image is near the border of an original image, as can be seen in Figure 5.10. The pixels of a neighboring image can reduce the classification score of a face near a border. Because of the loss in accuracy, this method is not investigated any further.

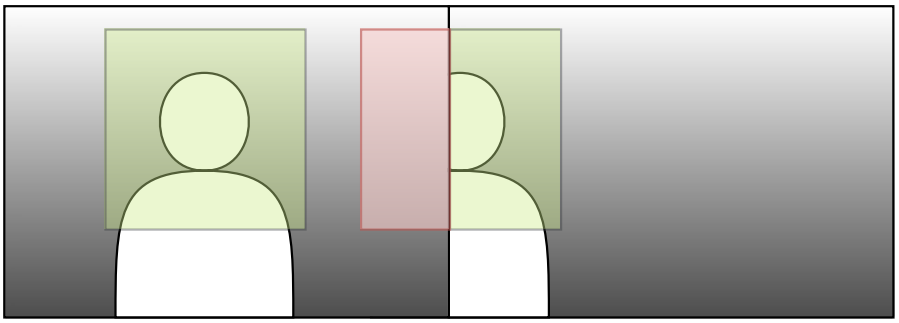


Figure 5.10: The pixels of the left image may negatively contribute to the face detection score of the detection window in the right image.

The second approach to increase the amount of threads per kernel launch, is a slight variation on the `for-loop` approach. A number of images is still combined by W CPU threads and sent to the GPU, but instead of looping over multiple images inside the kernel, we now change the number of dimensions of the thread grid. The grids of two-dimensional kernels are now changed to three-dimensional, with the third dimension corresponding to the number of different images on the GPU. The dimension of the classification thread grid can not be increased any further, since it already is three-dimensional (which is the maximum). Therefore, the number of threads in the x dimension shall be made W times bigger. Each thread checks on which image it should perform computations by dividing its x coordinate in the grid by the width of the original image. To distinguish which detection belong to which original image, the detections on each image are all placed in a separate part of the array that is sent back to the CPU. The implementation is shown in Figure 5.11.

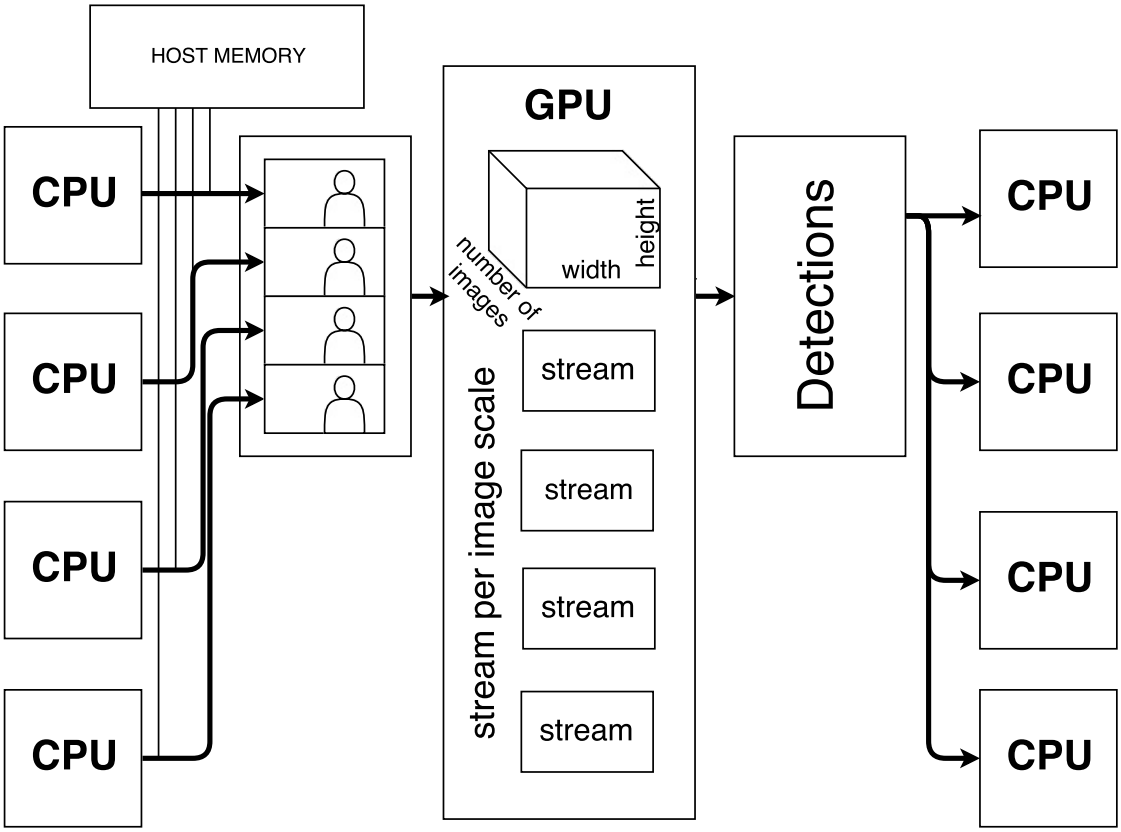


Figure 5.11: Face detection implementation where the grid of each kernel increases in the z dimension, for multiple images analyzed per kernel launch.

The performance results of this implementation are shown in Table 5.19. From these results, and the NVVP output in Appendix A.5, we can conclude that this method results in concurrent kernel execution and the fastest face detection per image until this point in the design. Since it does not have any of the drawbacks of the other image combination

methods, this approach is the clear winner and shall be used for the rest of the project. Furthermore, the use of 16 CPU threads and, therefore, 16 images simultaneously on the GPU, results in the best performance.

Table 5.19: Time from the start of the first kernel until completion of the last kernel, for implementation where the thread grid size is increased by putting multiple images in the z dimension.

Number of images	**Total time (ms)**	**Time per image (ms)**
4	1.231	0.308
16	4.613	0.289

5.1.13 Block sizes

Because the optimization of each individual kernel is finished, we can now empirically determine which block sizes result in the best performance for each kernel. The z dimension of the thread block, which corresponds to the image on which a thread performs it computation (as described in the previous section), is kept at 1 since threads in the same block should perform computations on the same image. The z dimension is also kept at 1 in the classification kernel for reasons discussed in Section 5.1.7.1.

Results of block sizes of $16 \times 8 \times 1$, $8 \times 16 \times 1$, $16 \times 16 \times 1$, $32 \times 16 \times 1$, $16 \times 32 \times 1$ and $32 \times 32 \times 1$ for each kernel are shown in Table 5.20.

Table 5.20: Comparison of kernel execution times for different x and y dimension block sizes. The z dimension is kept fixed at 1. The reported times are from face detection performed on a total of 16 images.

Kernel	**Time (ms)**					
	16×8	8×16	16×16	32×16	16×32	32×32
Scaling	**0.304**	0.384	0.311	0.318	0.325	0.349
GradientHistogram	1.114	1.177	0.993	**0.937**	0.984	1.077
Energy	0.174	0.207	0.173	0.147	0.159	**0.101**
Feature	0.495	0.683	0.468	0.347	0.397	**0.277**
Classifier	-	5.954	5.215	5.227	**2.364**	2.442
Detection	0.135	0.092	0.097	0.082	0.082	**0.068**

For each kernel, the block size that results in the best performance is boldfaced and used in the final implementation.

5.1.14 Data transfers between host and device

Each image is approximately 0.3 MB in size, which results in a total host-to-device data-transfer size of 4.915 MB for 16 images, which takes 536 μs to complete. The 16

images are all sent in one transfer to increase the used bandwidth and minimize the overhead cost. The array that is returned from the device to the host (which contains the face locations) is approximately 2 MB is size and takes 150 μs to transfer. In comparison, the total kernel execution time is 4613 μs, so the data transfers between host and device take up 12.9% of the total face detection time.

Speeding up transfer time, and overlapping kernel executions with memory transfers, requires the use of page-locked memory, which is more expensive to allocate and deallocate [102]. Because we are dealing with relatively small and infrequent data transfers, it is expected that effectively little is gained from the use of page-locked memory. Therefore, we shall not investigate this any further at this point.

5.1.15 Summary of optimizations for GPU face-detection implementation

Because a multitude of different optimizations for GPU face detection have been investigated in this work, the final optimizations are summarized:

- The image and downscaled versions of the image are stored in CUDA arrays, in order to make use of the free bilinear interpolation of the GPU texturing hardware in the image scaling kernel.
- The gradient and histogram kernel converts the histogram values to a fixed-point representation, in order to make use of hardware-native atomic operations in shared memory. This way, the histograms of a thread block are combined in shared memory first, which reduces the number of global-memory atomic transactions. The shared memory is also used as coalescing buffer, to store the histogram data efficiently in the gradient and histogram kernel, while also ensuring coalesced loads in the energy and feature kernels.
- The row and column filter kernels have been combined by making use of shared memory and block-wide synchronization. This greatly reduced the number of global memory transactions, at the cost of 28.125% more column filter computations.
- In order to ensure that all SMs of the GPU are utilized, we make use of streams to combine kernels that employ a small number of threads.
- Because the CPU was not able to issue kernel launches fast enough to make effective use of the GPU streams (work was not added to the streams fast enough to enable concurrent kernel execution), we increased the kernel execution times by analyzing multiple images with each kernel launch. This improves the kernel-launch-time to kernel-execution-time ratio, and, therefore, allows work to build up in the GPU streams fast enough to enable concurrent kernel execution.

5.2 Landmark detection on multi-core CPU with OpenMP

Once the face has been detected, we can continue with the blink response detection. A total of 68 landmarks is detected on each face, 6 of which are on the eye, which are used to estimate the closure of the eyelid. The OpenMP API (also OMP) shall be used for multi-core CPU acceleration of the landmark detection algorithm. Since there is no dependency between different frames of a video, each frame can be processed individually by a separate CPU thread.

We shall investigate two different approaches to achieve this: OMP tasks and OMP work-sharing constructs.

5.2.1 Work-sharing construct

The OMP work-sharing construct makes use of `#pragma omp parallel for` to share the number of iterations of a `for-loop` between the available CPU threads. In our case the total amount of frames is divided between the CPU threads. In Table 5.21 the total landmark detection time can be seen for 1, 2, 4, 8 and 16 CPU threads.

Table 5.21: Multi-threaded CPU approach to landmark detection, using OMP work sharing constructs on a sequence of 672 frames.

CPU threads	**Time per image (ms)**
1	2.723
2	1.459
4	0.788
8	0.449
16	**0.249**

5.2.2 Tasks

The use of OMP tasks is based on a *farmer-worker* model, where one thread adds work (tasks) to a pool of worker threads. Threads that are idle can pick up these tasks, execute them, and rejoin the pool of available worker threads. After the 'farmer' thread has finished scheduling all the tasks, it joins the pool of worker threads to complete the execution of the remaining tasks. Synchronization barriers can be added to ensure all tasks up to a certain point are finished before the next ones are started. Furthermore, one can specify dependencies between tasks or groups of tasks to ensure execution in a particular order.

The results of the OMP tasks implementation for landmark detection employing 1, 2, 4, 8 and 16 threads are shown in Table 5.22.

Table 5.22: Multi threaded CPU approach to landmark detection, using OMP tasks on a sequence of 672 frames.

CPU threads	**Time per image (ms)**
1	2.866
2	1.429
4	0.799
8	0.433
16	**0.248**

5.2.3 Pipelining

The two different OMP implementations of landmark detection show little difference, as there are no dependencies or synchronization requirements between the frames. However, when we include the face detection step on the GPU, one of the methods might prove itself better suited for the project as a whole.

We can distinguish three time-consuming components of the total process of eye blink response detection:

- Image loading and decoding: Images are stored in the JPEG format on a Solid State Drive. They are fetched and decoded into an array of pixel intensity values. This is done by 16 CPU threads concurrently and takes approximately 2.659 ms per 16 images.
- Face detection on GPU: One CPU thread sends the image data to the GPU, which performs face detection on 16 images and returns an array with detection locations. This takes approximately 5.299 ms for 16 images, including data transfers between host and device.
- Landmark detection: The landmark detection is also done by 16 CPU threads concurrently, and takes approximately 3.989 ms to complete per set of 16 images.

If we execute these steps sequentially, this should result in a total blink response detection time of 11.947 ms per 16 images, or 0.747 ms per image. A test on 672 images shows that the average blink response detection time is **0.821** μ**s**, which is close to this estimate. Some additional steps, such as the non-maximum suppression and the conversion from face coordinates to detection rectangles, could account for the remaining difference.

However, there is a possibility to partly overlap the computations done on CPU and GPU by making use of a pipeline model. This is shown in Figure 5.12.

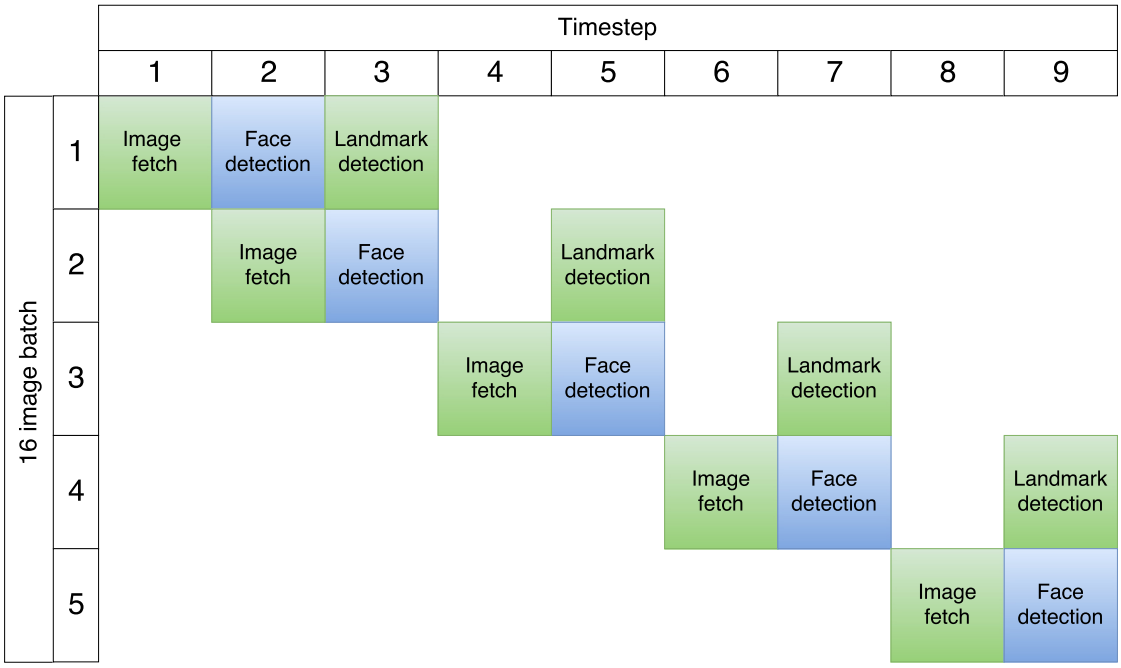


Figure 5.12: Performing the computations on CPU and GPU simultaneously by making use of a pipeline model. The landmark and face detection execution times (excluding host-device memory transfers) are comparable. Items shown in green are performed on CPU, while items in blue are performed on GPU.

This approach lets each CPU thread perform landmark detection on the previous batch of images, while the GPU performs the face detection on the next. This has implications for the choice of OpenMP model. Each CPU thread loads an image into a shared array that is sent to the GPU. Three time steps later, landmark detection is performed on that same image.

The idea of OMP tasks is that any available thread can pick a task up and perform it. This requires that the previously loaded images are shared variables. It also means that we are not able to exploit the fact that the image data is already loaded into lower level cache layers three time steps earlier. In contrast, the use of OMP work sharing-constructs allows us to keep the images private variables, and by making use of the `OMP_PROC_BIND` environment variable, we can ensure that each logical CPU thread is bound to the same physical CPU core for the duration of the program, consequently enabling the possible reuse of previously cached data.

The two pipelined models are compared in Table 5.23. From these results we can conclude that we have effectively hidden the face detection execution time, and that the work-sharing constructs implementation outperforms the tasks implementation.

Table 5.23: Comparison of total eyeblink-response detection execution time, using different OMP methods in combination with a pipeline that overlaps computations on CPU and GPU. The time per image is averaged over a sequence of 672 images.

Pipeline implementation	**Time per image (ms)**
Work sharing constructs	**0.554**
Tasks	0.599

6 Evaluation

In this chapter, the final implementations of the face- and landmark-detection algorithms are evaluated, as well as their combined pipelined implementation for blink-response detection. The experimental setup that is used for testing is discussed in Section 6.1. In Section 6.2, the final implementation of the algorithms is discussed and compared to the original to review the achieved speedup. Section 6.3 will cover how the algorithm performance scales with different CPU and GPU hardware, while Section 6.4 discusses the scalability with respect to the problem size. Finally, Section 6.5 provides a discussion of the results.

6.1 Experimental set-up

The implementations in this chapter are evaluated on a set of 10 eyeblink-response videos, consisting of a total of 6720 grayscale images, recorded by the Neuroscience department of Erasmus MC. Each image is 640×480 pixels in size. The GPU that is used for evaluation has been previously described in Section 5.1.1 and its specifications are summarized in the device column of Table 6.1. The CPU is an AMD Ryzen 7 1800X, of which the specifications are summarized in the host column of Table 6.1, along with those of the host memory.

Table 6.1: Specifications of device (NVIDIA Titan X (Pascal)) and host (AMD Ryzen 7 1800X + Corsair Vengeance DDR4 DRAM)

Specification	**Device**	**Host**
Clock speed	1.53 GHz	3.60 GHz
Number of cores	3584	8 (16 threads)
DRAM size	12 GB	32 GB
DRAM bandwidth	480 GB/s	34 GB/s
L1 data cache size	28×48 KB	8×32 KB
L2 cache size	3 MB	8×512 KB
L3 cache size	-	16 MB

Timing measurements on the GPU are performed with the NVVP, while those on the host are done with the OMP wall-clock timer.

6.2 Acceleration results

In this section, we shall discuss the acceleration results of the face detection, landmark detection and combined implementation for eyeblink-response detection.

6.2.1 Face detection on GPU

Face detection is performed by using the Histogram of Oriented Gradients algorithm for feature extraction in combination with a linear classifier. The original implementation in the Dlib machine-learning library is already sped up with SIMD AVX2 instructions, to exploit a small degree of data-level parallelism and to make use of specialized instructions such as fused multiply-add (FMA). We maximize the data-level parallelism potential by deploying the algorithm on a GPU, while also making use of task-level parallelism by executing multiple kernels concurrently in streams and performing the face detection on multiple images at the same time.

Four different face-detection implementations are evaluated:

- The original Dlib implementation without AVX2 instructions (see Section 4.1);
- The original Dlib implementation with AVX2 instructions (see Section 4.1);
- The unoptimized naive GPU implementation (see Section 5.1);
- The fully optimized GPU implementation (including and excluding the memory transfer time between host and device) (see Section 5.1).

The results are shown in Table 6.2.

Table 6.2: Performance comparison of different implementations of the face-detection algorithm. Speedup is calculated relative to the slowest implementation.

Implementation	**Time/image (ms)**	**Speedup**
Original serial version	583.623	-
Original version AVX2	30.208	19
Unoptimized naive GPU version	16.624	35
Optimized GPU version incl. memory transfer	0.333	1753
Optimized GPU version excl. memory transfer	0.289	2018

Compared to the original sequential version, a final speedup of 1753× is achieved with the optimized GPU accelerated version. If we exclude the memory transfers from host to device and back, and purely look at the kernel execution time, the achieved speedup is 2018×.

The accuracy of the GPU-accelerated face detector in the recording setting of this project, was verified on 1062080 images with faces from the Erasmus MC eyeblink-response videos. In only 184 of those images, the face detector was unable to find a face in the image, which is less than 0.02%. In each of these 184 images, the subjects showed

a clear violation (too much head rotation) of the face-detection requirements described in Section 3.1, as can be seen in Figure 6.1. On the remaining 1061896 images, exactly one face was detected, which indicates the algorithm has a low chance of false positives.

Figure 6.1: Example of two images where the face-detection algorithm was unable to identify a face. Note that both subjects show a high degree of yaw rotation.

6.2.2 Landmark detection on multi-core CPU

The landmark-detection algorithm uses an ensemble of regression trees to refine the estimation of the landmark locations in an iterative process. The iterative nature and large number of reductions of the algorithm make it ill-suited for data-level parallelism. A task-level parallel approach is, thus, implemented with OMP, making use of 16 CPU cores (see Section 5.2). This implementation is compared to the original sequential implementation in Table 6.3.

Table 6.3: Performance comparison of original sequential landmark-detection algorithm with the OMP-accelerated version. Speedup is calculated relative to the slowest implementation.

Implementation	**Time per image (ms)**	**Speedup**
Original serial version	2.878	-
OMP 16-thread accelerated version	0.251	11.49

We achieve a speedup of 11.49× compared to the sequential implementation by making use of 16 CPU threads. The eyeblink-response graphs generated with the landmark locations extracted with the algorithm were reviewed and deemed satisfactory by the neuroscientists of Erasmus MC for which this project is carried out.

The standalone accelerated face-detection (0.289 ms) and landmark-detection (0.251 ms) algorithms achieve similar execution times. Due to the fact that they are decoupled components that are executed on different platforms, pipelining (interleaving) of the tasks of the two components (across CPU and GPU) is possible. The fact that the

execution times are well-matched means that the amount of time the CPU or GPU is idle, is minimal.

6.2.3 Combined implementation for eyeblink-response detection

The complete solution, computing an eyeblink-response graph from a sequence of images, involves loading the image files from memory and decoding them, detecting the faces, and detecting the landmarks. These steps were originally executed in a sequential order. However, because we are dealing with multiple images, we can make use of pipelining to simultaneously perform computations on the CPU and GPU.

We compare four versions of the complete eyeblink-response detection implementation in Table 6.4:

- The original sequential Dlib implementation without AVX2 instructions for face detection;
- The original Dlib implementation with AVX2 instructions for face detection;
- The accelerated implementation executing the image load and decode step, face-detection step and landmark-detection step sequentially;
- The accelerated implementation overlapping the steps that are performed on CPU (image load and decode, and landmark detection) and GPU (face detection) by making use of a pipelined model.

Table 6.4: Comparison of different implementations of the complete eyeblink-response detection solution, which includes the image loading and decoding, face detection and landmark detection steps. Speedup is calculated relative to the slowest implementation.

Implementation	**Time per image (ms)**	**Speedup**
Original serial version	586.906	-
Original version AVX2	32.674	18
Accelerated non-pipelined version	0.801	732
Accelerated pipelined version	0.533	1101

By comparing Table 6.2 and Table 6.3, we can see that the original implementation is heavily dominated by the face-detection component. Acceleration results in comparable face-detection and landmark-detection times, which are overlapped in the final implementation. This results in a final speedup for the complete solution of 1101×, compared to the original sequential implementation. This enables an eyeblink-response detection speed of roughly 1876 FPS, which more than satisfies the original requirement of 500 FPS. Furthermore, it is noticed that although the face- and landmark-detection algorithms have comparable execution times, overlapping their execution by making use of a pipelined model does not lead to virtual halving of the execution time. This is due to memory I/O and image-decode operations.

6.3 Hardware scalability

To see how the algorithm scales with different hardware, we must investigate what limits the performance of the current implementation. The three different steps of the process (face detection, landmark detection and image-loading and -decoding) will be discussed separately.

6.3.1 Face detection on GPU

The face-detection process is split up into 6 different kernels. The NVVP indicates for each of these kernels whether the performance is limited by computing resources, memory bandwidth or latency. The results are shown in Table 6.5. Since none of the kernels are compute bound, the amount of single-precision floating-point units that is utilized in each kernel is also reported (specified by the profiler on a 10-step scale from 0 to 1).

Table 6.5: Limiters of kernel performance of GPU-accelerated face detection. SP = single-precision floating point.

Kernel	**% of total time**	**Performance limiter**	**SP unit utilization**
Classifier	44.5	L2-cache bandwidth	0.4
Gradient histogram	27.9	Latency	0.7
Scaling	6.4	Latency	0.2
Feature	5	Bandwidth	0.2
Energy	1.8	Bandwidth	0.1
Detection	1.3	Latency	0.2
Memset	13.1	-	-

From this table we can conclude that the kernels are either bandwidth or latency bound. Although the gradient histogram, scaling and detection kernels each employ more than enough threads to ensure maximum theoretical occupancy on each of the 28 SMs (2048 threads per SM), there is still not enough thread- and instruction-level parallelism to effectively hide the latency of high-latency operations. Future versions of GPU architectures, that will allow for more than 2048 resident threads per SM, would alleviate this problem and allow for a more effective use of the available bandwidth and computing resources. This number has been increased in previous versions before, as can be seen in Table 6.6 [2].

Table 6.6: Maximum amount of resident threads per SM for different versions of CUDA Compute Capability.

CUDA Compute Capability	**Maximum number of threads per SM**
1.0 - 1.1	768
1.2 - 1.3	1024
2.x	1536
3.0 - 7.0 (most recent)	2048

Unfortunately, instruction latencies are not specified, or made public, by NVIDIA, so little is known about how they have evolved or improved over the course of new architectures, and whether to expect improvements in that area.

The other kernels are bound by the bandwidth of either the off-chip device memory or on-chip L2 cache. Since the device-memory bandwidth has been increasing at a rapid

pace over the course of recent NVIDIA GPU architectures (see Figure 6.2), we can expect significant increases in Feature- and Energy-kernel execution times. Unfortunately, NVIDIA does not release the specifications of the L2-cache bandwidth of any of its architectures. In case it scales in a similar fashion as the device memory bandwidth, this would allow for a significant speedup of the classifier kernel and, considering the relative importance of the kernel, the whole face-detection execution time as a whole.

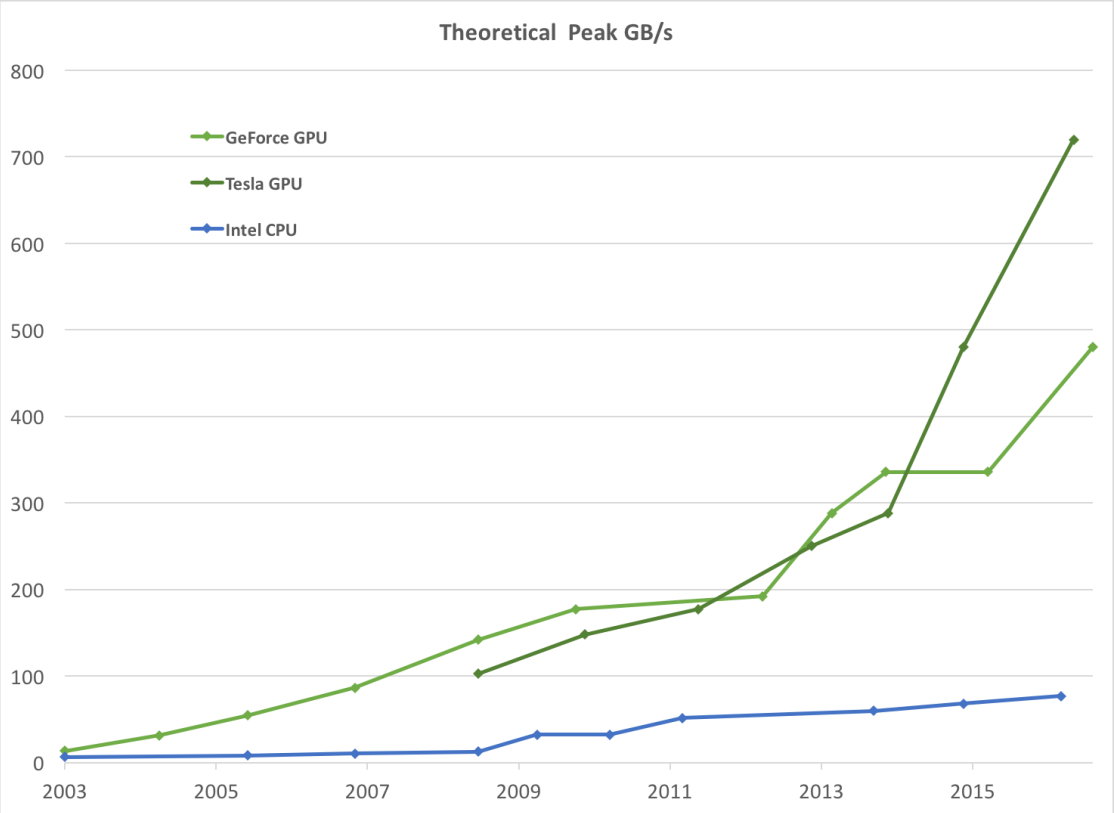


Figure 6.2: Evolution of device memory bandwidth of different NVIDIA GPU architectures in time [2].

A weighted average of the kernels' single-precision floating-point unit utilization is approximately 46% (excluding `memset`). If the increased bandwidth of future architectures allows this utilization to go up, we might reach a point where the kernels become compute-bound. However, as can be seen in Figure 6.3, the relative increase in computational power is comparable to the relative increase of device-memory bandwidth in recent years. If this trend continues, the kernels are likely to remain bandwidth- (and latency-) bound in future architectures.

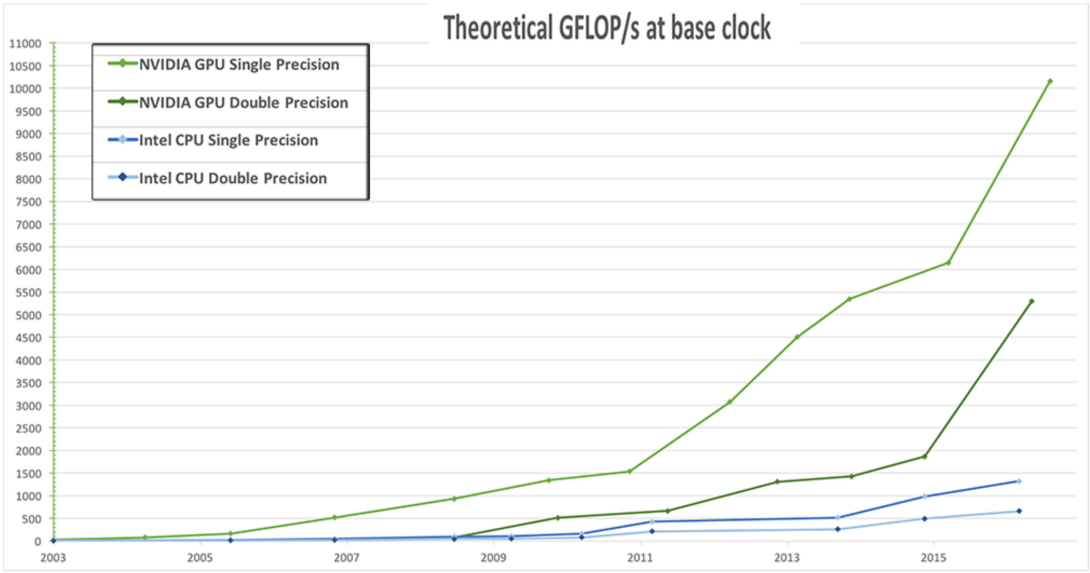


Figure 6.3: Evolution of maximum theoretical FLOPS of different NVIDIA GPU architectures in time [2].

Another way to increase the face-detection speed is to increase to amount of task-level parallelism by increasing the number of GPUs. For example, using twice the amount of GPUs would allow each GPU to perform the face-detection algorithm on 16 images concurrently, therefore doubling the face-detection throughput.

The data transfers between host and device take up approximately 12.7% of the total face-detection time. These transfers are done via PCIe 3.0, which has a theoretical peak bandwidth of 15.75 GB/s. As the kernel execution time decreases with improved GPU architectures, this data-transfer time could eventually become the dominant factor of the total time. However, the more recent 4.0 version of the PCIe interconnect already doubles the bandwidth compared to its predecessor [103]. In addition, NVIDIAs new high-speed NVLINK interconnect between CPU and GPU (or between multiple GPUs) boasts a bandwidth of 80 GB/s and is already being used in current HPC platforms [104].

6.3.2 Landmark detection on multi-core CPU

The scalability of the landmark-detection speed with the number of CPU threads has been discussed previously in Section 5.2. There appears to be a linear relation between the number of CPU threads and the achieved speedup. If we extrapolate the trend line that described this relation, we can see that for 32 threads, a speedup of approximately 19.6× would be achieved. This can be seen in Figure 6.4.

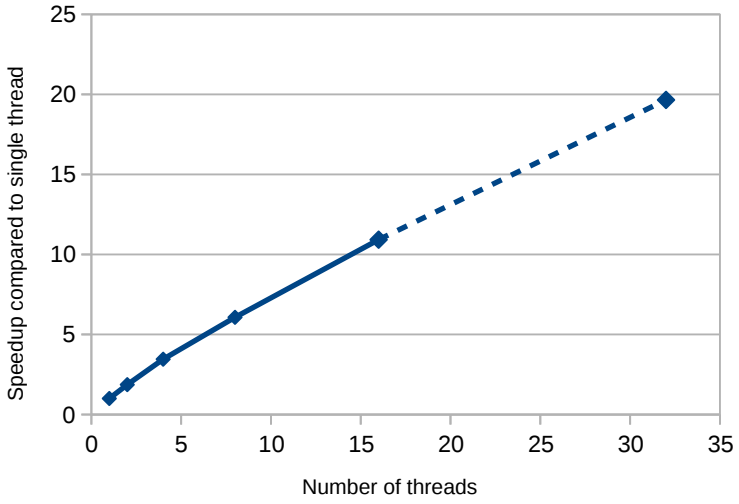


Figure 6.4: Speedup of landmark detection with multiple CPU threads. The dashed line indicates the possible speedup for a 32-threaded accelerated implementation.

6.3.3 Image loading and decoding

The image-loading and -decoding step is performed by 16 threads simultaneously. Execution times for implementations using 1, 2, 4 ,8 and 16 threads are shown in Table 6.7. An explanation for the marginal speedup going from 1 to 2 CPU threads could be the additional required step of combining the images before sending them to the GPU, which is included in the timing. Obviously, this step is not required for a single-threaded implementation.

Table 6.7: Execution time of the image loading and decoding step for a varying number of CPU threads.

CPU threads	Time per image (ms)
1	1.271
2	0.935
4	0.515
8	0.313
16	0.166

Similar to the landmark detection, there appears to be a linear relation between the number of threads and the speedup. Extrapolating the line in Figure 6.5 predicts a total speedup of 14.39 when making use of 32 threads.

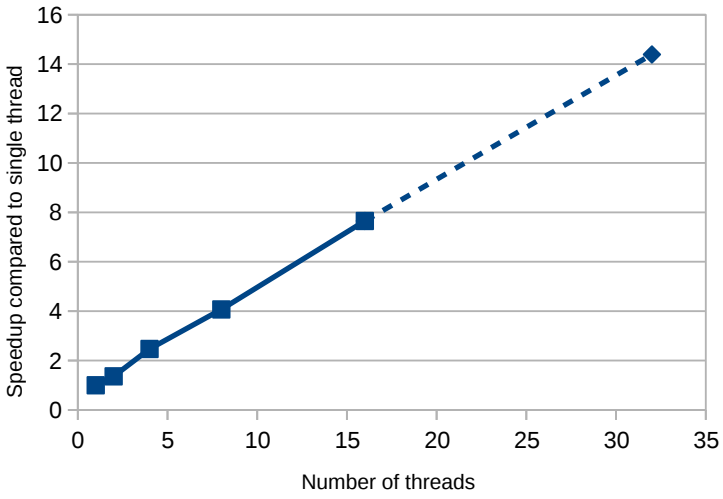


Figure 6.5: Speedup of image loading and decoding with multiple CPU threads. The dashed line indicates the possible speedup for a 32-threaded accelerated implementation.

6.3.4 Host-memory bandwidth

In the previous sections, a linear performance increase for both the landmark detection and image-loading and -decoding steps is estimated when increasing the amount of CPU threads from 16 to 32. This estimation assumes that the current performance of the complete solution is compute-bound and can benefit from extra computing resources. In case the performance of the algorithm is memory-bound, i.e. the host memory bandwidth is already fully utilized, adding the extra CPU threads will not result in extra speedup. Unfortunately, no profiler or appropriate tool was found to monitor the host-DRAM bandwidth utilization of the application. However, a rough estimation of the amount of memory I/O traffic can give an indication of how much of the peak bandwidth is currently being used.

The filter values of the linear classifier of the face detector only have to be read from the host memory once, and are stored in the GPU device memory for the duration of the program. Therefore, these filter values will not contribute to the amount of utilized host-memory bandwidth during the eyeblink-response detection process. The same holds for the regression trees, which only have to be read once from host memory and are small enough to remain in the last-level cache from that point on. Since, for each analyzed image, only a single value describing the eyelid closure has to be written back to memory, the only significant remaining contributing factor to the host-memory bandwidth utilization is the loading of images from memory. These images are stored in JPEG format, and approximately 31 KB in size. At the achieved detection speed of

$\frac{1}{0.533*10^{-3}} \approx 1876$ frames per second, this results in only 58.161 MB of memory traffic per second. Although this is a very rough estimate, this amount is so much smaller than the peak theoretical host-memory bandwidth of 34 GB/s, that we consider it very unlikely that the performance of the current implementation is memory-bound.

6.3.5 Minimum hardware to meet the requirements

The current implementation is more than able to achieve the required detection speed of 500 frames per second. Because possible future work for this project could be to investigate the possibilities for a more mobile solution, an estimation is made of the minimum required hardware to achieve a detection speed of 500 FPS.

Because the implementation overlaps the face detection on GPU and image loading, image decoding and landmark detection on CPU, we can define the total eyeblink-response detection time as:

$$Time = \min\left((LD + ILD),\ FD\right)$$

$$LD = landmark\ detection,\ ILD = image\ load\ and\ decode,\ FD = face\ detection$$

A detection speed of 500 FPS equals a maximum detection time of 2 ms. If we combine Table 6.7 and Table 5.21, we can calculate the total amount of execution time spent on the CPU for the image loading and decoding and landmark detection steps combined, for a varying number of threads. This can be seen Table 6.8.

Table 6.8: Total combined execution time of the steps performed on the CPU.

CPU threads	**Time per image (ms)**
1	3.995
2	2.395
3	1.640
4	1.303
8	0.761
16	0.416

From this table we can deduce that, to satisfy the execution time requirement of < 2 ms, no more than 3 CPU threads need to be utilized. Note that this estimation requires a CPU with comparable performance per thread as the one used in our experimental set-up.

The scaling of the execution time on the GPU is somewhat harder to assess, since each of the 6 kernels will scale differently with changes in the GPU architecture. However, the kernel limiters in Table 6.5 indicate that a reduction in bandwidth would have more severe implications on the face-detection time than a reduction in peak singe-precision floating-point computing performance. Assuming the same interconnect between host and device, the total data-transfer time remains constant at 0.044 ms per image. This leaves 1.956 ms for kernel execution time. This is a factor $\frac{1.956}{0.289} \approx 6.77$

more than the kernel execution time of the current implementation.

Furthermore, it should be noted that with different hardware, the execution time of the face- and landmark-detection steps could be less well-matched than in the current implementation. This would result in a larger idle-time of either the GPU or CPU, depending on which of the steps takes longer to complete.

6.4 Problem-size scalability

In the current eyeblink conditioning set-up, videos are recorded at a resolution of 640×480 pixels. Because future experiments may be recorded at a higher resolution, it is important to assess the performance with relation to the input image size.

The different feature-extraction and classification steps of the face-detection algorithm all have linear time complexity ($O(n)$, where n is the problem size, i.e. the number of pixels in the image). However, the pyramidal image-scaling structure of the face detector results in a greater number of scales to be analyzed as the input image size increases. While the image size increases, the detection-window size remains 80×80 pixels wide, therefore allowing the detection of faces that are relatively smaller compared to the total image size. However, as described in the requirements of Section 3.1, the size of the face is always more than 20% of the image, no matter the input image size. We can use this information to skip the face detection on bigger image scales, which is already done in the current implementation as described in Section 5.1.10. This effectively results in **a feature-extraction and classification time independent of the input image size**. The only two operations that scale (linearly) with the image size are the host-device memory-transfer time and the scaling-kernel time (which needs to scale the image down more, in order to get to the smaller scales of which the features are extracted and classified). Furthermore, in future work, the algorithm could be slightly adjusted to scale down from the original image size to the largest image size that needs to be analyzed in one step, instead of multiple steps of $\frac{5}{6}^{th}$ (the scaling-factor, see Section 4.1.1), before proceeding with the downscaling in the smaller steps for all image scales that need to be analyzed. This would result in constant time complexity for the image downscaling step.

The landmark-detection algorithm (ensemble of regression trees) has a runtime complexity that does not depend on the input size, but on the number of layers in the cascade (T), the number of regression trees (K) in each layer and the depth of each tree (F) ($O(TKF)$, as described in [7]).

Finally, the image decoding from JPEG to an array of pixels has linear time complexity ($O(n)$).

The results for an image sizes of 640×480, 1280×960 and 2560×1920 pixels are shown in Table 6.9.

Table 6.9: Execution time of different steps of the blink response detection for different image sizes. IFD = image fetch and decode, FD = face detection, LM = landmark detection.

Blink response detection step	**Time per image (ms)**		
	640×480	1280×960	2560×1920
IFD	0.166	1.230	5.066
FD	0.333	0.734	1.436
LD	0.251	0.268	0.300

In the table above, the image size increases by a factor 4 per column. The image fetch and decode step is almost 8 times slower for the 1280×960 input size compared to the original size. A possible explanation for this, is that for the smaller image size, each of the 8 cores can make effective use of simultaneous multi-threading (SMT) while this is not the case for the increased image sizes. The same step scales as expected for the 2560×1920 image size, as it is about 4 times slower than the 1280×960 size. The face- and landmark-detection steps show the expected scaling behavior. In case the data-transfer time between host and device become the dominant factor of the face detection, it is worth re-evaluating the use of pinned memory (see Section 5.1.14) for increased transfer speed and concurrent data transfers with kernel executions.

6.5 Discussion

The implemented solution for eyeblink-response detection achieves super-real time speed by making use of a combination of multi-core CPU and GPU acceleration. The HOG algorithm is used for face detection while an ensemble of regression trees is used to estimate a total of 68 landmarks on the face. Six of these landmarks (per eye) are on the eyelid and are used to estimate the amount of eyelid closure.

While Chapter 3 discusses the selection process of the algorithms in detail, the field of computer vision and face detection is so extensively studied that it was not possible to investigate and empirically compare all of the different approaches in detail. Based on the studied literature and carried out experiments, the algorithms that were chosen were thought to be the most suitable for this particular project.

While the accuracy of the face and landmark detection could be verified with the use of annotated databases, the final result, the eyeblink-response graph containing the amount of closure of the eyelid over time, is harder to objectively assess. Although the results were manually inspected and positively reviewed by the neuroscientists of the eyeblink-conditioning project, the amount of samples that could be evaluated in this way was limited. In any case, the landmark detection is able to accurately determine the location of the eye. If it turns out that the landmarks on the eyelid are eventually not deemed accurate enough for the eyeblink-response graph, an extra algorithm can be

performed on the image of the eye to estimate the amount of eyelid closure in a different way.

7 Case study

The accelerated implementation for eyeblink-response detection enables the fast generation of new eyeblink-conditioning data. In this chapter we showcase an interesting use of a newly-acquired database, by trying to answer whether the asynchrony of the eyelids during the conditioning experiment is related to the performance of the eyeblink response.

7.1 Problem description and hypothesis

Eyeblink conditioning is a powerful model to measure multiple aspects of how we learn in our daily life. During the experiment, we can make a distinction between two types of blinks, which involve the use of two different types of memory:

- **Conscious blinks**: the subject actively focuses on the sound (CS), and responds by quickly closing its eyes after hearing it. This involves the use of explicit memory: knowledge of facts and events ("knowing that").
- **Unconscious, or automatic, blinks**: the subject is not focused on the eyeblink-conditioning experiment. The use of implicit memory ("knowing how") lets it perfectly time the closure of its eye without even realizing it.

The use of explicit and implicit memory involve different parts of the brain. While the cerebellum is known to be involved in unconscious learning and the use of implicit memory, the hippocampus and prefrontal cortex are used for explicit memory (see Figure 7.1) [12].

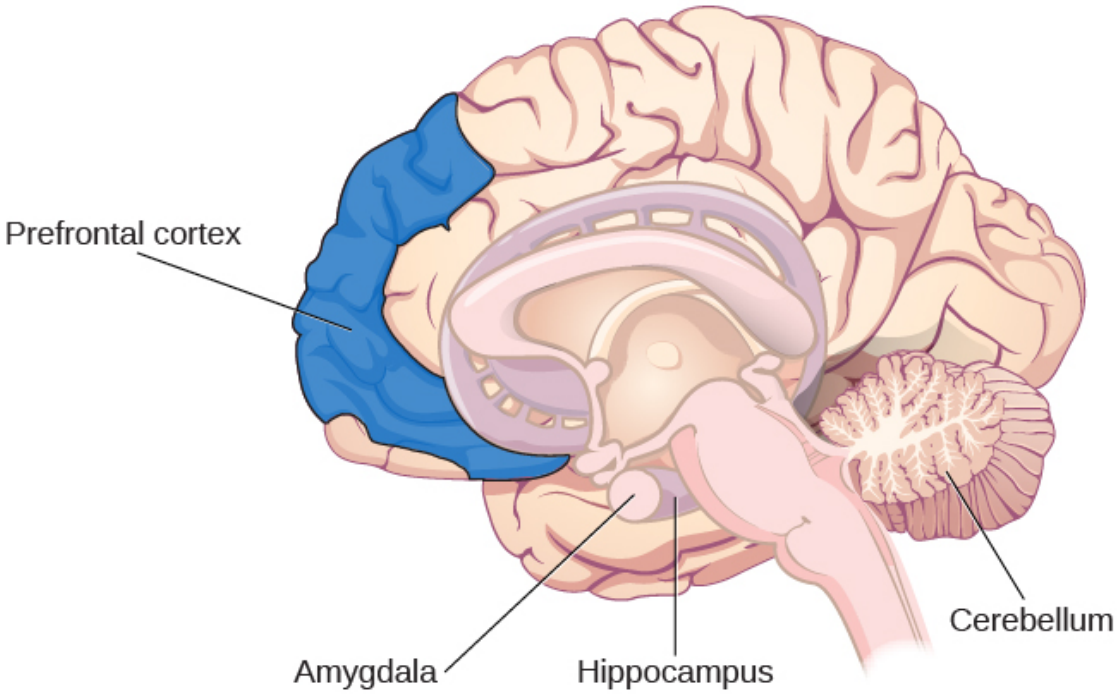


Figure 7.1: Different parts of the brain involved with memory [1].

Since the interval between the CS and US in our experiments is 500 ms, it is very hard to perfectly time a conscious blink. However, this timing is handled automatically by the implicit memory in case of an unconscious blink. To make a distinction between conscious and unconscious blinks in video data, we can make use of the asynchrony of the eyelids. It is thought that conscious blinks are done with two eyes synchronously, while (a slight) asynchrony between the eyelids indicates an automatic blink (because the air puff is only applied to the left eye, which is therefore the only eye that effectively needs to be closed). Therefore, the following hypothesis is formulated by one of the neuroscientists of Erasmus MC:

"The asynchrony of the eyelids of both eyes during the conditioned response is related to the performance of the eyeblink response."

What defines the performance of an eyeblink response is covered later in Section 7.3.3. To investigate this hypothesis, a dataset of 1440 eyeblink-response videos has been recorded. The relation between the asynchrony of the eyelids in the video and the eyeblink-response performance is analyzed by using a variety of machine-learning regression techniques in Section 7.3. These techniques learn a model that tries to predict the eyeblink-response performance based on a set of input features that represent the asynchrony of the eyelids. The idea is that if these features allow us to make more accurate predictions than a baseline model, which does not have the asynchrony information, we

[1] Image taken from *https://courses.lumenlearning.com/wsu-sandbox/chapter/parts-of-the-brain-involved-with-memory/*

are able to confirm the hypothesis.

7.2 Background

In machine learning, a regression problem concerns itself with estimating the relation between n input variables X_0, X_1 ... X_n and a continuous output variable Y. A linear regressor assumes a linear relation between the variables in the form of:

$$Y = b_0X_0 + b_1X_1 + ... + b_nX_n + b_{n+1},$$

where b_0, b_1 ... b_n are the scaling coefficients and b_{n+1} is the bias term. These are determined during model training. A non-linear regressor assumes no such form. Both linear and non-linear regression techniques estimate a function that best models the training data by minimizing a cost function. This cost function is usually defined as:

$$\sum_{n=0}^{m} |\hat{Y}_n - Y_n|$$

or

$$\sum_{n=0}^{m} (\hat{Y}_n - Y_n)^2 \ ,$$

where $\hat{Y}_n - Y_n$ is the difference between the value of $\hat{Y}$ predicted by the regressor and the actual value of Y (the error) of sample n, and m is the number of samples in the training set.

The two non-linear regression techniques that are used in this work are Support Vector Regression (SVR) and Multilayer Perceptrons (MLP). The exact details of how they function are beyond the scope of this work, but the global idea is that they map the input features to an other (higher dimensional) feature space on which linear regression is performed. For a more detailed understanding, we refer the reader to [105] and [106] for SVRs, and [107] and [108] for MLPs.

7.3 Implementation

7.3.1 Dataset

For this case study, 120 blink responses were recorded from 12 subjects each, resulting in a total dataset of 1440 eyeblink videos. The length of each video is 2 seconds (666 frames at 333 FPS); the CS and US are applied after 500 ms and 1000 ms, respectively.

7.3.2 Feature extraction

The first step towards feature extraction from the videos is to convert the videos to sequences of landmark locations in time, by using the accelerated implementation of the face and landmark detector described in the previous chapters. For each of the 68 landmarks, the location is stored for all 666 frames in the video, which represents their

movement in time. The landmarks corresponding to the left and right eye (see Figure 7.2 are used to plot the closure of each eye in time.

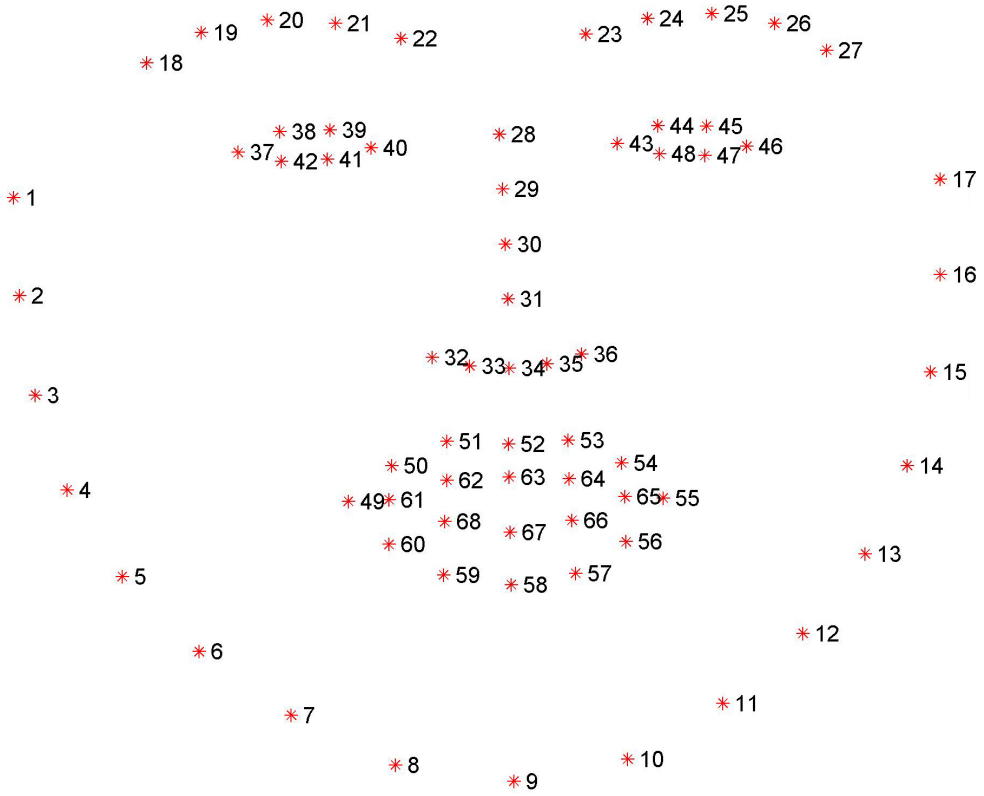


Figure 7.2: Landmark numbering of the landmark detection algorithm [2].

The landmarks on the eye are converted to a measure of eyelid closure by means of the Eye Aspect Ratio (EAR) [83], which, for the left eye, is defined as:

$$EAR = \frac{||LM_{44} - LM_{48}|| + ||LM_{45} - LM_{47}||}{2 \times ||LM_{43} - LM_{46}||}$$

$$LM = landmark$$

The lower the EAR value, the more the eye is closed. Additionally, from the videos it was noticed that during blinking, the landmarks of the eye move downwards on the face, towards the nose. This continues for a number of frames even while the eyelid is already fully closed, and further highlights the asynchrony of the blinks. Therefore, the euclidean distance from the center of the eye to the center of the nose is incorporated as scaling factor in the EAR values. This distance is divided by the length of the nose (also deduced from the landmark detection locations) for scale invariance:

[2]Image taken from *https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/*

$$EAR = \frac{||LM_{44} - LM_{48}|| + ||LM_{45} - LM_{47}||}{2 \times ||LM_{43} - LM_{46}||} \times \frac{||EC - LM_{34}||}{||LM_{28} - LM_{34}||}$$

$$LM = landmark,\ EC = eye\ center$$

A graph of the eyelid closure over time (as the frames in the video progress) that is extracted in this way is shown in Figure 7.3.

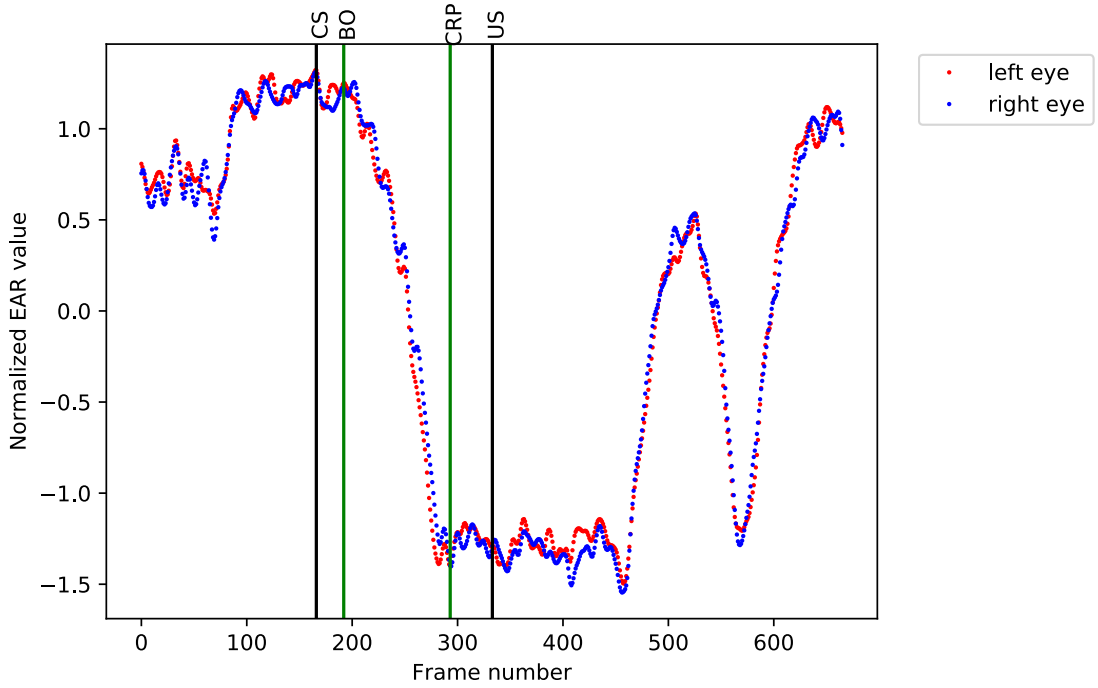


Figure 7.3: Eyelid closure of left and right eye in a period of 2 seconds. The EAR value is scaled to zero mean and unit variance. Lower values correspond to more eyelid closure. CS = Conditioned Stimulus, BO = Blink Onset, CRP = Conditioned Response Peak, CR = Conditioned Response.

The asynchrony between the eyes is calculated by subtracting the left-eye EAR values from the right-eye EAR values. Furthermore, only the difference-signal between the CS and US is used, because this is the only *conditioned* behavior in the video. To filter out high-frequency noise, a second-order low-pass Butterworth filter is used. An example of an asynchronous blink from the dataset is shown in Figure 7.4, and the corresponding asynchrony graph in Figure 7.5.

Figure 7.4: Snapshot of the eyes during an asynchronous blink in a video from the dataset. This screenshot is frame 317 from the video (see (Figure 7.5).

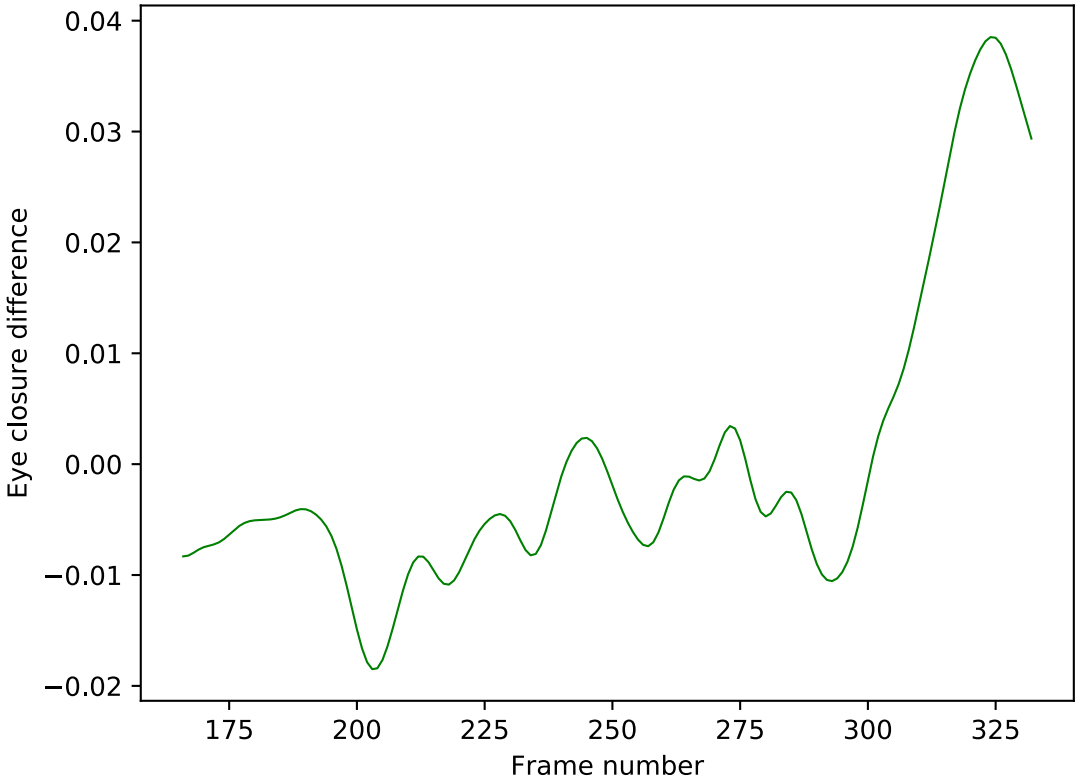


Figure 7.5: The eyelid closure difference is defined as the EAR values of the right eye subtracted by those of the left eye. Notice the peak around frame 320, which corresponds to the asynchronous blink screenshot of Figure 7.4.

However, visual inspection of the videos and difference graphs suggests that only taking the euclidean distance from the center of the eye to the center of the nose (the

EAR scaling factor) is a better representation of the asynchrony of the eyes during blinking. Figure 7.6 shows an eyelid closure difference graph that is generated in this way (from the same video as the graph of Figure 7.5).

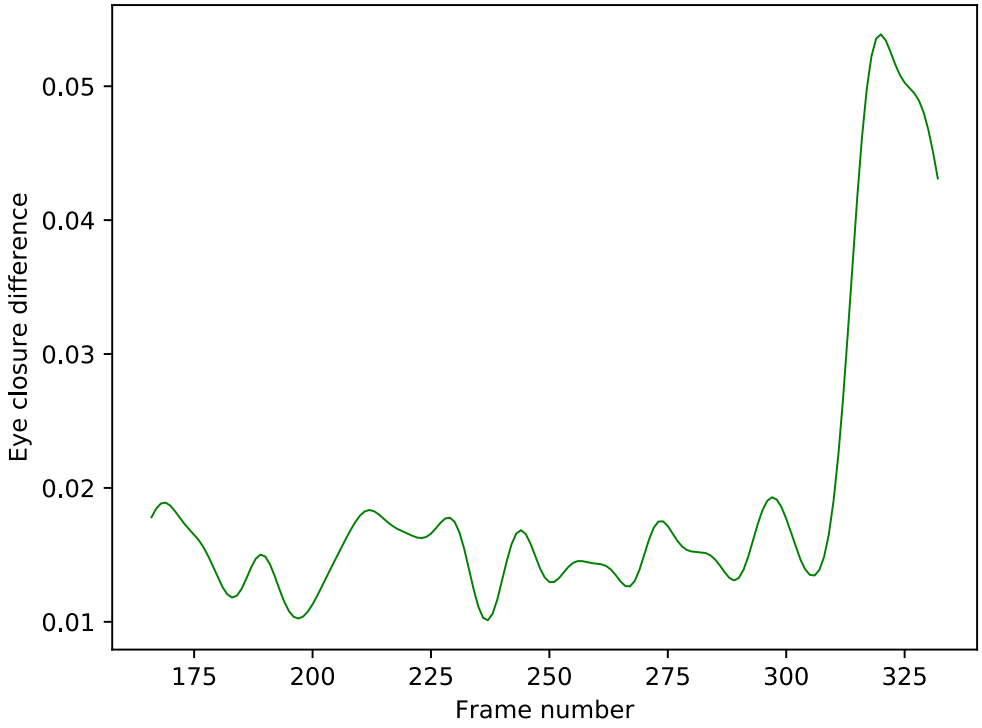


Figure 7.6: The level of eyelid closure of each eye is now defined as the euclidean distance from the center of the eye to the center of the nose. The eyelid closure difference is calculated by subtracting the values of eyelid closure of the right eye by those of the left eye.

Extra features are added by computing the first- and second-order derivative of the eyelid-closure difference signal (with respect to the frame number, which represents time). These features contain information about the difference in the speed and acceleration of the two eyelids during the conditioned response.

Since there are 167 frames between the CS and US, and we have a value of the eyelid-closure difference and its first- and second-order derivative for each of these frames, this results in a total of $167 \times 3 = 501$ input features per sample, or a 501-dimensional feature space. We choose to perform dimensionality reduction on these features by describing each of the 3 signals by 11 statistical features (so 33 features in total), because the dimensionality of the feature space (501) is very high compared to the number of samples in the dataset (1440), and the features are highly correlated (and therefore contain

redundant information). These redundant features do not only increase the regressor's execution time but also affect its robustness: the ability to correctly predict previously unseen data [109]. A large training data set would allow the regression techniques to "filter out" the (relation between the) important features, but unfortunately we are working with a small number of samples. Therefore, this filtering is now done by applying the dimensionality reduction in a preprocessing step.

The 11 statistical features we choose to describe the signals with, are based on the work of [110], which also focuses on time-series analysis and regression problems. For each eyelid-closure difference signal, its derivative and its second derivative, we select the following descriptive statistics:

- Arithmetic mean;
- Standard deviation;
- Skewness;
- Kurtosis;
- Minimal and maximal value;
- The 1^{st}, 25^{th}, 50^{th}, 75^{th} and 99^{th} percentile.

This results in a final set of **33 features** that is used as input for the regressors.

7.3.3 Labels

While the input features have been defined, we also need a measure for the performance of an eyeblink response; the label Y (see Section 7.2). Together with a neuroscientist of Erasmus MC, a set of characteristics is defined that describe the performance of an eyeblink response:

- The moment of the onset of the blink in the CR, relative to the CS and US. Closer to the US is better.
- The moment of the maximum amount of eyelid closure in the CR, relative to the CS and US. Closer to the US is better.
- The amount of eyelid closure at the time of the US, relative to all other frames of the video. Higher is better.

Each of the three characteristics gets values between 0 and 1, in the following way:

$$OB = \frac{time_{blink\ onset} - time_{CS}}{time_{US} - time_{CS}}$$

$$CRP = \frac{time_{peak\ CR} - time_{CS}}{time_{US} - time_{CS}}$$

$$CSA = \frac{amplitude_{CS}}{\max(amplitude_0,\ amplitude_1\ ...\ amplitude_{last\ frame})}$$

$$OB = onset\ blink,\ CRP = CR\ peak,\ CSA = CS\ amplitude$$

The final eyeblink-response performance score is calculated by multiplying the three factors: $OB \times CRP \times CSA$. Note that this score is only calculated for the eyeblink response of the left eye, since this is the only eye that the US is applied to. An example of an eyeblink response and the corresponding calculation of the performance score is shown in fig. 7.7.

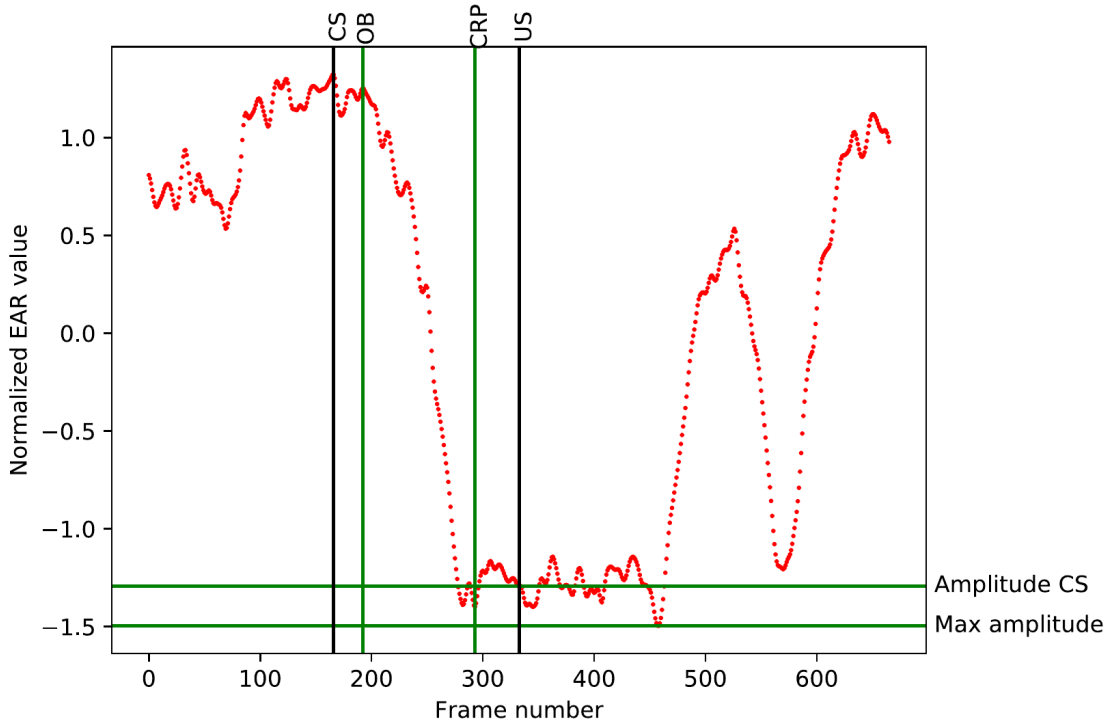


Figure 7.7: Example of eyeblink reponse graph with performance score calculation. $OB = 0.16,\ CRP = 0.76,\ CSA = 0.93$, total score $= 0.11$.

7.4 Evaluation

Three different models were trained to predict the eyeblink-response performance based on the aforementioned 33 features. We compare these models with a baseline model to see whether the asynchrony features with which the models are trained improve prediction performance (and therefore the asynchrony of the eyelid of both eyes during the conditioned response is related to the performance of the eyeblink response).

The prediction performance of each model is verified by 5-fold cross-validation. This means that we train the model 5 times, each time with a different 80% of the dataset (the training set), and review its prediction performance on the remaining 20% (the test set). The accuracy of each model is defined by the mean prediction error of the samples in the test set:

$$\frac{\sum_{n=0}^{m} |\hat{Y}_n - Y_n|}{m}$$

For model training and testing we make use of the Scikit-learn [111] and Keras [112] Python libraries.

7.4.1 Baseline

The baseline model takes the average eyeblink-response performance of the training set as prediction value for each sample of the test set. The mean prediction error for each of the 5 steps of cross validation is shown in Table 7.1.

Table 7.1: Mean prediction error of baseline model for eyeblink-response performance prediction.

Cross validation number	**Mean prediction error test set**
1	0.207
2	0.208
3	0.215
4	0.212
5	0.203

Averaged over all 5 steps of cross validation, the mean prediction error is **0.209**.

7.4.2 Linear regression

Linear regression is performed in order to map the relation between the asynchrony features and the eyeblink response performance into a linear function. Table 7.2 shows the mean prediction error for each of the 5 steps of cross validation.

Table 7.2: Mean prediction error of linear regression model for eyeblink response performance prediction based on asynchrony features.

Cross validation number	Mean prediction error test set
1	0.179
2	0.184
3	0.190
4	0.182
5	0.186

Averaged over all 5 steps of cross validation, the mean prediction error is **0.184**.

7.4.3 Support Vector Regression

For the non-linear SVR, we can tune two parameters C and γ, that determine the influence of each sample in the training set on the function that describes the relation between the input features and the output variable, and the complexity of that function, respectively [113]. To determine which parameters result in the best prediction performance, we make use of the `GridSearchCV` function of Scikit-learn. This function tries all combination of C and γ in a given parameter space and returns the best performing one. The parameter space is described in Table 7.3.

Table 7.3: Parameter space of SVR that was examined by `GridSearchCV`.

Parameter	Values
C	0.00001, 0.0001, 0.001, 0.01, **0.1**, 1, 10, 100, 1000
γ	0.00001, 0.0001, 0.001, 0.01, **0.1**, 1, 10, 100, 1000, 10000, 100000

The best-performing combination of parameters is 0.1 for both C and γ. Cross-validation results for the SVR regressor with these parameters are shown in Table 7.4.

Table 7.4: Mean prediction error of non-linear SVR model for eyeblink-response performance prediction based on asynchrony features.

Cross validation number	Mean prediction error on test set
1	0.163
2	0.174
3	0.175
4	0.164
5	0.175

Averaged over all 5 steps of cross validation, the mean prediction error is **0.170**.

7.4.4 Multi-layer perceptron

Multi-layer perceptrons also map the relation between the input features and the output value into a non-linear function. In contrast to the SVR, there are a multitude of parameters to tune that affect the performance of the prediction model. How each of these parameters affects the model performance is beyond the scope of this work. We employ the same `GridSearchCV` function to explore a large parameter space for both a single-hidden (Table 7.5) and double-hidden (Table 7.6) layer MLP.

Table 7.5: Parameter space of single-hidden layer MLP that was examined by multiple runs of `GridSearchCV`.

Parameter	**Values**
Batch size	4, **8**
Training epochs	100, 300, **500**, 700
Optimizer	SGD, **Adagrad**, Adam
Weight initialization	**uniform**, normal
Activation function	**relu**, tanh, sigmoid
Regularization	**L2(0.01)**, L2(0.05), L2(0.1)
Neurons in hidden layer	4, 8, 16, 32, **64**, 96, 128

Table 7.6: Parameter space of double-hidden layer MLP that was examined by multiple runs of `GridSearchCV`.

Parameter	**Values**
Batch size	4, **8**
Training epochs	100, 300, 500, **700**, 1000
Optimizer	SGD, **Adagrad**, Adam
Weight initialization	uniform, **normal**
Activation function	**relu**, tanh, sigmoid
Regularization	**L2(0.01)**, L2(0.05), L2(0.1)
Neurons in 1^{st} hidden layer	4, 8, 16, 32, 64, 96, **128**
Neurons in 2^{nd} hidden layer	4, 8, 16, 32, 64, 96, 128, 160, 192, 256, **384**, 512

The best-performing combination of parameters is boldfaced in both tables. The double-hidden layer MLP outperformed the single-hidden layer one. The results of the cross-validation test with the best-performing double-hidden layer MLP are shown in Table 7.7.

Table 7.7: Mean prediction error of double-hidden layer MLP model for eyeblink-response performance prediction based on asynchrony features.

Cross validation number	Mean prediction error on test set
1	0.154
2	0.169
3	0.164
4	0.148
5	0.175

Averaged over all 5 steps of cross validation, the mean prediction error is **0.162**.

7.4.5 Summary of prediction methods

The three different regression models for eyeblink performance prediction that are employed in this case study all outperform the baseline model on prediction accuracy, as can be seen in table 7.8.

Table 7.8: Comparison of mean prediction error on test data of different prediction models.

Prediction model	Mean prediction error
Baseline	0.209
Linear regression	0.184
Non-linear SVR	0.170
Double-hidden layer MLP	**0.162**

The most accurate regression model, a double-hidden layer MLP, achieves a reduction of the mean prediction error of more than 22% compared to the baseline model. The difference in information that these two models use for the prediction of the eyeblink-response performance, lies in the features that describe the asynchrony of the eyelids during the conditioned response. Therefore, we conclude that:

The asynchrony of the eyelids of both eyes during the conditioned response is related to the performance of the eyeblink response,

and therefore, the hypothesis is confirmed.

7.4.6 Discussion

The dimensionality-reduction technique that is employed to reduce the number of features per sample from 501 to 33, is just one of many techniques to convert data into a sparser representation and lessen the amount of redundant information. There is no certainty that these 33 features represent the original data in the most optimal

way. Other dimensionality-reduction techniques, such as Principal Component Analysis (PCA), could be investigated to see whether they enable better prediction results.

7.5 Conclusion

In this case study, we have investigated whether there is a relation between the asynchrony of the eyelids during a conditioned response, and the performance of the eyeblink response (in the setting of an eyeblink-conditioning experiment). To this end, 1440 eyeblink-response videos were recorded, from which we extracted 33 asynchrony features each. Furthermore, we developed a method to objectively quantify the performance of an eyeblink response as a value between 0 and 1. The relation between the asynchrony features and the eyeblink-response performance was modeled with three different machine-learning regression techniques. Their accuracy is evaluated by their prediction performance: how well they are able to predict the eyeblink-response performance of previously unseen data, based on the 33 asynchrony features. A double-hidden layer Multi-layer Perceptron (MLP) achieved the best prediction performance, reducing the mean prediction error by more than 22% compared to the baseline model, which does not make use of the asynchrony features. Therefore, we conclude that there is a statistically significant relation between the asynchrony of the eyelids during a conditioned response, and the performance of the eyeblink response.

Future work

Although we have concluded that there is a relation between the asynchrony of the eyelids of both eyes during the conditioned response, and the performance of the eyeblink response, the specifics of this relation are not known at this point. The downside of the non-linear SVR and MLP regression techniques is that they are both gray-box models and it is now known how they map the relation between the input features and output variable. A possible direction for future work would therefore be to investigate the nature of this relation. Furthermore, acquiring a bigger dataset that can be used to better train the regression models will almost certainly improve the prediction results.

8 Conclusions

This chapter concludes the thesis work. In Section 8.1 the contributions of the thesis will be discussed, while Section 8.2 describes possible directions for future work.

8.1 Contributions

The goal of this thesis has been to select a combination of algorithms that is able to detect the amount of human left eyelid closure in video data (eyeblink-response detection), and to accelerate these algorithms in order to achieve real-time processing speeds (500 FPS). This is the first step towards an on-line implementation that would not only alleviate the need for large off-line data storage, but also enable the neuroscientists of Erasmus MC to dynamically adjust eyeblink-conditioning experiments based on immediately available feedback on the subject's performance.

The detection process is split up in two different parts: the first algorithm detects the location of the face in an image, which is then used as the input for a second algorithm that determines the amount of eyelid closure.

To select the appropriate face-detection algorithm, first a set of requirements was specified that describes under which conditions it must be able to detect a face. Three face-detection algorithms were evaluated on a subset of images of the Annotated Facial Landmarks in the Wild database that meets these requirements. While the state-of-the-art Convolutional Neural Network (CNN) was the most accurate algorithm, the Histogram of Oriented Gradients (HOG) achieves nearly the same accuracy in our constrained setting while being more than five times as fast. The Haar face detector, which is even faster, was not considered suitable for this project because it is significantly less accurate than the other two methods. Ultimately, the HOG face detection algorithm is chosen because of its combination of accuracy and speed.

The algorithm analyzes each image on multiple scales in order to be able to detect faces of varying sizes (scale invariance). It is implemented on a GPU to exploit its data-level parallelism potential. Additionally, in order to maximize the GPU utilization, multiple images and multiple scales of the same image are analyzed concurrently by making use of streams, hereby also making use of task-level parallelism. A final face-detection time of 333 μs is achieved, which is a speedup of 1753$\times$ compared to the original sequential implementation.

Once the face in an image has been detected, we can proceed with the second step and determine the amount of closure of the eyelid. The original focus was to approach

this problem in a similar way as face detection: features are extracted from the face image, and a (pre)trained classification model is used to determine the location of the eyes. A second algorithm would be needed to determine the amount of eyelid closure in each eye image. However, the choice was made to switch to landmark detection, because it allowed for a simpler and computationally less expensive solution. The landmark-detection algorithm, an Ensemble of Regression Trees (ERT), detects 68 landmarks on the face, 6 of which are on each eye, which can be used directly to compute the amount of eyelid closure. This approach has the additional advantage that the other landmarks can be used to analyze the movement of other face muscles during the eyeblink-conditioning experiment, which was indicated as an area of interest by the neuroscientists of Erasmus MC. While the algorithm is not well-suited for data-level parallelism acceleration because of its iterative nature, it is already fast sequentially (2.878 ms per image). The algorithm was accelerated with OpenMP by exploiting the inherent task-level parallelism, hereby dividing the total number of frames of a video between 16 CPU threads. This resulted in a landmark detection time of 251 μs, and therefore a speedup of 11.49× compared to the original sequential implementation.

Because both algorithms are being executed on different platforms (CPU and GPU), their executions were overlapped by making use of a pipeline model. Including the time it takes to load the image from memory and decode the JPEG file format (166 μs per image), this ultimately resulted in a total eyeblink-response detection time of 533 μs, which is a speedup of more than 1101× compared to the original sequential implementation. Furthermore, this translates to a detection speed of 1876 FPS, which is well above the required real-time processing speed of 500 FPS.

Furthermore, we have also proceeded to evaluate a specific case study. In this case study, we have investigated whether there is a relation between the asynchrony of the two eyelids during a conditioned response, and the performance of the eyeblink response (in the setting of an eyeblink-conditioning experiment). To this end, 1440 eyeblink-response videos were recorded, from which we extracted 33 asynchrony features each. A method was developed to objectively quantify the performance of an eyeblink response as a value between 0 and 1. The relation between the asynchrony features and the eyeblink-response performance was modeled with three different machine-learning regression techniques. Their accuracy is evaluated by their prediction performance. A double-hidden layer Multi-layer Perceptron (MLP) achieved the best prediction performance, reducing the mean prediction error by more than 22% compared to the baseline model, which does not make use of the asynchrony features. Therefore, we conclude that there is a relation between the asynchrony of the eyelids during a conditioned response, and the performance of the eyeblink response.

Concisely, the following contributions have been made by this thesis:

- Different face detectors have been compared on a database of 24384 face images. The Histogram of Oriented Gradients was selected as the algorithm that best-suited the project requirements.

- The face-detection algorithm was accelerated on a GPU by making use of data and

task-level parallelism, achieving a speedup of 1753×.

- A landmark-detection algorithm was selected to estimate the amount of closure of the eyelid.
- The landmark-detection algorithm was accelerated with OpenMP on a 16-threaded CPU by making use of a task-level parallelism. This resulted in a speedup of 11.49×.
- The accelerated face and landmark-detection algorithms were combined and their execution was overlapped by making use of a pipeline model. This final eyeblink-response detection algorithm is 1101× faster than the original sequential version and achieves a detection speed of 1876 FPS, which is more than the 500 FPS required for real-time processing.
- The accelerated implementation for eyeblink-response detection was used to generate a database of 1440 eyeblink-response videos during the conditioning experiment. This database was analyzed with multiple machine-learning regression techniques, and the conclusion was drawn that there is a relation between the asynchrony of the eyelids and the eyeblink-response performance.

8.2 Future work

The work that is presented in this thesis can be considered as the first step towards an on-line solution for eyeblink-response detection. Consequently, one of the directions for future work is to investigate and carry out the required steps to go from the current implementation to an on-line implementation. Globally, this would require the recording equipment to directly send the video data as batches of images to a worker node over Ethernet cables. The worker node processes the images and sends the results to an additional node that shows and stores the results to allow for immediate analysis of the subject's performance.

Another direction that has been discussed is the implementation of the algorithm on a mobile platform. The current implementation achieves a detection speed of 1876 FPS, and therefore shows that the required detection speed of 500 FPS can also be achieved with less powerful hardware. However, it is considered unlikely that the implementation in its current form is able to run on a mobile platform. Therefore, it can be investigated if the accuracy of the implementation can be reduced to an acceptable extent, in order to speed up the eyeblink-response detection. Globally, this would mean that we exploit the high correlation between subsequent images in a video. For example, if we limit the amount of scales and locations on which each image is analyzed, this would already greatly reduce the face-detection execution time. Additionally, the face detector could be applied to only a part of the frames (and, for example, a tracking algorithm could be used for the frames in between), since the movement of the face in subsequent frames is expected to be limited. Furthermore, a landmark-detection algorithm could be trained that reaches its final landmark estimation in less iterations, uses less regression trees per

iteration or reduces the depth of each regression tree in order to speed up the detection process.

The intermediate steps of face and landmark detection that are used to achieve the eyeblink-response detection, allow for interesting new research possibilities with relation to the eyeblink-conditioning experiment. Currently, only the left eye is analyzed, but the neuroscientists of Erasmus MC have also expressed their interest in the movement of other facial muscles during the conditioning experiment. The landmark-detection algorithm detects a total of 68 landmarks of the face, which can be used to analyze the movement of other regions of the face. Additionally, different landmark detectors boast the detection of up to 192 landmarks, and can be used if a more detailed model of the face is required for this direction of future work. Another possibility is that instead of using an available trained model, a shape-prediction model is trained that focuses on a particular part of the face.

Furthermore, an interesting possible area of research is to see whether the emotion of subjects affects their performance during the eyeblink-conditioning experiment. Convolutional Neural Networks have been successfully employed for automatic emotion recognition in humans [114], and could be used to automatically record the emotional state of the subject during the experiment. These networks require the image of the face as input, which is already extracted by our face-detection algorithm.

Bibliography

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[2] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2017.

[3] S. Albelwi and A. Mahmood, "A framework for designing the architectures of deep convolutional neural networks," *Entropy*, vol. 19, no. 6, p. 242, 2017.

[4] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.

[5] K. Hundal, A. Mearza, and N. Joshi, "Lacrimal gland prolapse in blepharochalasis," *Eye*, vol. 18, no. 4, pp. 429–430, 2004.

[6] Y. Chen, J. Yang, and J. Qian, "Recurrent neural network for facial landmark detection," *Neurocomputing*, vol. 219, pp. 26–38, 2017.

[7] V. Kazemi and J. Sullivan, "One millisecond face alignment with an ensemble of regression trees," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1867–1874.

[8] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[9] X. Jin and X. Tan, "Face alignment in-the-wild: a survey," *arXiv preprint arXiv:1608.04188*, 2016.

[10] I. P. Pavlov and G. V. Anrep, *Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex*. Oxford University Press: Humphrey Milford, 1927.

[11] H. Cason, *The Conditioned Reaction, by Hulsey Cason ...*, 1922. [Online]. Available: https://books.google.nl/books?id=ULcUnQAACAAJ

[12] H.-J. Boele, *Neural Mechanisms underlying Motor Learning*, May 2014. [Online]. Available: http://hdl.handle.net/1765/51759

[13] J. P. Welsh and J. T. Oristaglio, "autism and classical eyeblink conditioning: Performance changes of the conditioned response related to autism spectrum disorder diagnosis," *Frontiers in psychiatry*, vol. 7, 2016.

[14] S. Brown, P. Kieffaber, C. Carroll, J. Vohs, J. Tracy, A. Shekhar, B. O'Donnell, J. E. Steinmetz, and W. Hetrick, "Eyeblink conditioning deficits indicate timing and cerebellar abnormalities in schizophrenia," *Brain and cognition*, vol. 58, no. 1, pp. 94–108, 2005.

[15] J. P. Lewis, "Fast template matching," in *Vision interface*, vol. 95, no. 120123, 1995, pp. 15–19.

[16] H. Du Nguyen, Z. Al-Ars, G. Smaragdos, and C. Strydis, "Accelerating complex brain-model simulations on gpu platforms," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 974–979.

[17] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. de Zeeuw *et al.*, "Brainframe: A node-level heterogeneous accelerator platform for neuron simulations," *Journal of Neural Engineering*, 2017.

[18] Y. Ma, G. Smaragdos, Z. Al-Ars, and C. Strydis, "Towards real-time whisker tracking in rodents for studying sensorimotor disorders," in *Proc. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, July 2017.

[19] A. Karapatis, R. M. Seepers, M. van Dongen, W. A. Serdijn, and C. Strydis, "Balancing accuracy, delay and battery autonomy for pervasive seizure detection," in *Engineering in Medicine and Biology Society (EMBC), 2016 IEEE 38th Annual International Conference of the*. IEEE, 2016, pp. 6343–6348.

[20] C. Strydis, R. M. Seepers, and A. Karapatis, "Trading detection accuracy for battery autonomy in a wearable seizure-detection device."

[21] L. Kros, O. H. Eelkman Rooda, J. K. Spanke, P. Alva, M. N. Dongen, A. Karapatis, E. A. Tolner, C. Strydis, N. Davey, B. H. Winkelman *et al.*, "Cerebellar output controls generalized spike-and-wave discharge occurrence," *Annals of neurology*, vol. 77, no. 6, pp. 1027–1049, 2015.

[22] M. N. van Dongen, A. Karapatis, L. Kros, O. E. Rooda, R. M. Seepers, C. Strydis, C. De Zeeuw, F. Hoebeek, and W. Serdijn, "An implementation of a wavelet-based seizure detection filter suitable for realtime closed-loop epileptic seizure suppression," in *Biomedical Circuits and Systems Conference (BioCAS), 2014 IEEE*. IEEE, 2014, pp. 504–507.

[23] B. G. Schreurs, A. R. Mcintosh, M. Bahro, P. Herscovitch, T. Sunderland, and S. E. Molchan, "Lateralization and behavioral correlation of changes in regional cerebral blood flow with classical conditioning of the human eyeblink response," *Journal of Neurophysiology*, vol. 77, no. 4, pp. 2153–2163, 1997.

[24] V. Bracha, L. Zhao, D. Wunderlich, S. Morrissy, and J. Bloedel, "Patients with cerebellar lesions cannot acquire but are able to retain conditioned eyeblink reflexes." *Brain: a journal of neurology*, vol. 120, no. 8, pp. 1401–1413, 1997.

[25] E. J. Kehoe, E. A. Ludvig, J. E. Dudeney, J. Neufeld, and R. S. Sutton, "Magnitude and timing of nictitating membrane movements during classical conditioning of the

rabbit (oryctolagus cuniculus).” *Behavioral Neuroscience*, vol. 122, no. 2, p. 471, 2008.

[26] M. Ivarsson and P. Svensson, “Conditioned eyeblink response consists of two distinct components,” *Journal of Neurophysiology*, vol. 83, no. 2, pp. 796–807, 2000.

[27] M. S. Raza and U. Qamar, “Understanding and using rough set based feature selection: Concepts, techniques and applications,” 2017.

[28] A. Chandarr and F. Gaisser, “2d image processing with opencv,” 2017.

[29] V. Volkov, “Better performance at lower occupancy,” 2010.

[30] S. Zafeiriou, C. Zhang, and Z. Zhang, “A survey on face detection in the wild: past, present and future,” *Computer Vision and Image Understanding*, vol. 138, pp. 1–24, 2015.

[31] P. Viola and M. J. Jones, “Robust real-time face detection,” *International journal of computer vision*, vol. 57, no. 2, pp. 137–154, 2004.

[32] F. C. Crow, “Summed-area tables for texture mapping,” *ACM SIGGRAPH computer graphics*, vol. 18, no. 3, pp. 207–212, 1984.

[33] Y. Freund and R. E. Schapire, “A desicion-theoretic generalization of on-line learning and an application to boosting,” in *European conference on computational learning theory*. Springer, 1995, pp. 23–37.

[34] R. Meir and G. Rätsch, “An introduction to boosting and leveraging,” in *Advanced lectures on machine learning*. Springer, 2003, pp. 118–183.

[35] T. Ojala, M. Pietikainen, and T. Maenpaa, “Multiresolution gray-scale and rotation invariant texture classification with local binary patterns,” *IEEE Transactions on pattern analysis and machine intelligence*, vol. 24, no. 7, pp. 971–987, 2002.

[36] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.

[37] M. Köstinger, P. Wohlhart, P. M. Roth, and H. Bischof, “Robust face detection by simple means,” in *DAGM 2012 CVAW workshop*, 2012.

[38] R. Benenson, M. Mathias, T. Tuytelaars, and L. Van Gool, “Seeking the strongest rigid detector,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2013.

[39] Q. Zhu, M.-C. Yeh, K.-T. Cheng, and S. Avidan, “Fast human detection using a cascade of histograms of oriented gradients,” in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 2. IEEE, 2006, pp. 1491–1498.

[40] F. Suard, A. Rakotomamonjy, A. Bensrhair, and A. Broggi, "Pedestrian detection using infrared images and histograms of oriented gradients," in *Intelligent Vehicles Symposium, 2006 IEEE*. IEEE, 2006, pp. 206–212.

[41] K. Levi and Y. Weiss, "Learning object detection from a small number of examples: the importance of good features," in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, vol. 2. IEEE, 2004, pp. II–II.

[42] A. Koschan, "A comparative study on color edge detection," in *Proceedings of the 2nd Asian Conference on Computer Vision*, vol. 3, 1995, pp. 574–578.

[43] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[44] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[45] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," *Computer vision–ECCV 2006*, pp. 404–417, 2006.

[46] R. S. Stanković and B. J. Falkowski, "The haar wavelet transform: its status and achievements," *Computers & Electrical Engineering*, vol. 29, no. 1, pp. 25–44, 2003.

[47] "Opencv documentation on surf," http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html, accessed: 10-05-2017.

[48] M. Mathias, R. Benenson, M. Pedersoli, and L. Van Gool, "Face detection without bells and whistles," in *European Conference on Computer Vision*. Springer, 2014, pp. 720–735.

[49] R. Lienhart and J. Maydt, "An extended set of haar-like features for rapid object detection," in *Image Processing. 2002. Proceedings. 2002 International Conference on*, vol. 1. IEEE, 2002, pp. I–I.

[50] P. Dollár, Z. Tu, P. Perona, and S. Belongie, "Integral channel features," 2009.

[51] C. A. Waring and X. Liu, "Face detection using spectral histograms and svms," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 35, no. 3, pp. 467–476, 2005.

[52] B. Wu and R. Nevatia, "Detection of multiple, partially occluded humans in a single image by bayesian combination of edgelet part detectors," in *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, vol. 1. IEEE, 2005, pp. 90–97.

[53] P. Sabzmeydani and G. Mori, "Detecting pedestrians by learning shapelet features," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007, pp. 1–8.

[54] S. Baluja, M. Sahami, and H. A. Rowley, "Efficient face orientation discrimination," in *Image Processing, 2004. ICIP'04. 2004 International Conference on*, vol. 1. IEEE, 2004, pp. 589–592.

[55] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[56] J. Bruna and S. Mallat, "Invariant scattering convolution networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1872–1886, 2013.

[57] S. S. Farfade, M. J. Saberian, and L.-J. Li, "Multi-view face detection using deep convolutional neural networks," in *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*. ACM, 2015, pp. 643–650.

[58] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[59] J. R. Uijlings, K. E. Van De Sande, T. Gevers, and A. W. Smeulders, "Selective search for object recognition," *International journal of computer vision*, vol. 104, no. 2, pp. 154–171, 2013.

[60] C. Szegedy, A. Toshev, and D. Erhan, "Deep neural networks for object detection," in *Advances in Neural Information Processing Systems*, 2013, pp. 2553–2561.

[61] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[62] E. D. Karnin, "A simple procedure for pruning back-propagation trained neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 2, pp. 239–242, 1990.

[63] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[64] X. Zhu and D. Ramanan, "Face detection, pose estimation, and landmark localization in the wild," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012, pp. 2879–2886.

[65] S. Wu, M. Kan, Z. He, S. Shan, and X. Chen, "Funnel-structured cascade for multi-view face detection with alignment-awareness," *Neurocomputing*, vol. 221, pp. 138–145, 2017.

[66] A. Neubeck and L. Van Gool, "Efficient non-maximum suppression," in *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, vol. 3. IEEE, 2006, pp. 850–855.

[67] P. M. R. Martin Koestinger, Paul Wohlhart and H. Bischof, "Annotated Facial Landmarks in the Wild: A Large-scale, Real-world Database for Facial Landmark Localization," in *Proc. First IEEE International Workshop on Benchmarking Facial Image Analysis Technologies*, 2011.

[68] G. Bradski, "Opencv library," *Dr. Dobb's Journal of Software Tools*, 2000.

[69] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, pp. 1755–1758, 2009.

[70] ——, "Max-margin object detection," *CoRR*, vol. abs/1502.00046, 2015. [Online]. Available: http://arxiv.org/abs/1502.00046

[71] D. W. Hansen and Q. Ji, "In the eye of the beholder: A survey of models for eyes and gaze," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 3, pp. 478–500, 2010.

[72] S. Kawato and N. Tetsutani, "Detection and tracking of eyes for gaze-camera control," *Image and Vision Computing*, vol. 22, no. 12, pp. 1031–1038, 2004.

[73] P. Wang and Q. Ji, "Learning discriminant features for multi-view face and eye detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 373–379.

[74] O. Jesorsky, K. J. Kirchberg, and R. W. Frischholz, "Robust face detection using the hausdorff distance," in *International Conference on Audio-and Video-Based Biometric Person Authentication*. Springer, 2001, pp. 90–95.

[75] M. M. Chakka, A. Anjos, S. Marcel, R. Tronci, D. Muntoni, G. Fadda, M. Pili, N. Sirena, G. Murgia, M. Ristori *et al.*, "Competition on counter measures to 2-d facial spoofing attacks," in *Biometrics (IJCB), 2011 International Joint Conference on*. IEEE, 2011, pp. 1–6.

[76] P. Polatsek, "Eye blink detection."

[77] C. Tomasi and T. Kanade, "Detection and tracking of point features," 1991.

[78] T. Morris, P. Blenkhorn, and F. Zaidi, "Blink detection for real-time eye tracking," *Journal of Network and Computer Applications*, vol. 25, no. 2, pp. 129–143, 2002.

[79] M. Wang, L. Guo, and W.-Y. Chen, "Blink detection using adaboost and contour circle for fatigue recognition," *Computers & Electrical Engineering*, vol. 58, pp. 502–512, 2017.

[80] K. Selvakumar, J. Jerome, K. Rajamani, and N. Shankar, "Real-time vision based driver drowsiness detection using partial least squares analysis," *Journal of Signal Processing Systems*, vol. 85, no. 2, pp. 263–274, 2016.

[81] K. Minkov, S. Zafeiriou, and M. Pantic, "A comparison of different features for automatic eye blinking detection with an application to analysis of deceptive behavior," in *Communications Control and Signal Processing (ISCCSP), 2012 5th International Symposium on*. IEEE, 2012, pp. 1–4.

[82] Z. Boukhers, T. Jarzyński, F. Schmidt, O. Tiebe, and M. Grzegorzek, "Shape-based eye blinking detection and analysis," in *Proceedings of the 9th International Conference on Computer Recognition Systems CORES 2015*. Springer, 2016, pp. 327–335.

[83] T. Soukupova and J. Cech, "Real-time eye blink detection using facial landmarks," in *21st Computer Vision Winter Workshop*, 2016.

[84] V. Le, J. Brandt, Z. Lin, L. Bourdev, and T. S. Huang, "Interactive facial feature localization," in *European Conference on Computer Vision*. Springer, 2012, pp. 679–692.

[85] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database for studying face recognition in unconstrained environments," Tech. Rep.

[86] J. Weidendorfer, "Sequential performance analysis with callgrind and kcachegrind," *Tools for High Performance Computing*, pp. 93–113, 2008.

[87] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[88] C. Sagonas, G. Tzimiropoulos, S. Zafeiriou, and M. Pantic, "300 faces in-the-wild challenge: The first facial landmark localization challenge," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2013, pp. 397–403.

[89] S. Umeyama, "Least-squares estimation of transformation parameters between two point patterns," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 13, no. 4, pp. 376–380, 1991.

[90] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[91] "Cuda c best practices guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations, accessed: 19-10-2017.

[92] V. Volkov, "Understanding latency hiding on gpus," Ph.D. dissertation, University of California, Berkeley, 2016.

[93] "Cuda c best practices guide," http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html, accessed: 20-10-2017.

[94] "Using shared memory in cuda c/c++," https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/, accessed: 20-10-2017.

[95] "Cuda pro tip: Increase performance with vectorized memory access," https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/, accessed: 20-10-2017.

[96] "Cuda pro tip: Kepler texture objects improve performance and flexibility," https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility//, accessed: 21-10-2017.

[97] J. Luitjens and S. Rennich, "Cuda warps and occupancy," *GPU Computing Webinar*, vol. 11, pp. 2–19, 2011.

[98] "Gpu pro tip: Fast histograms using shared atomics on maxwell," https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/, accessed: 22-10-2017.

[99] "Inside pascal: Nvidia's newest computing platform," https://devblogs.nvidia.com/parallelforall/inside-pascal/, accessed: 22-10-2017.

[100] "Cuda c programming guide: Compare and swap atomic operation," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions, accessed: 22-10-2017.

[101] "Cuda streams, best practices and common pitfalls," http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf, accessed: 23-10-2017.

[102] "Choosing between pinned and non-pinned memory," https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html, accessed: 25-10-2017.

[103] "Pci sig, what is pci express (pcie) 4.0)," https://pcisig.com/faq?field_category_value%5B%5D=pci_express_4.0&keys=, accessed: 25-10-2017.

[104] "Nvidia nvlink high-speed interconnec," http://www.nvidia.com/object/nvlink.html, accessed: 25-10-2017.

[105] "Support vector machine regression," kernelsvm.tripod.com/, accessed: 25-10-2017.

[106] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and computing*, vol. 14, no. 3, pp. 199–222, 2004.

[107] "Neural networks, manifolds, and topology," http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/, accessed: 25-10-2017.

[108] D. F. Specht, "A general regression neural network," *IEEE transactions on neural networks*, vol. 2, no. 6, pp. 568–576, 1991.

[109] A. Maier and D. Rodríguez-Salas, "Fast and robust selection of highly-correlated features in regression problems," in *Machine Vision Applications (MVA), 2017 Fifteenth IAPR International Conference on*. IEEE, 2017, pp. 482–485.

[110] C. Katsimerou, J. A. Redi, and I. Heynderickx, "A computational model for mood recognition," in *International Conference on User Modeling, Adaptation, and Personalization*. Springer, 2014, pp. 122–133.

[111] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[112] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[113] "Rbf svm parameters," http://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html, accessed: 06-11-2017.

[114] S. E. Kahou, C. Pal, X. Bouthillier, P. Froumenty, Ç. Gülçehre, R. Memisevic, P. Vincent, A. Courville, Y. Bengio, R. C. Ferrari *et al.*, "Combining modality specific deep neural networks for emotion recognition in video," in *Proceedings of the 15th ACM on International conference on multimodal interaction*. ACM, 2013, pp. 543–550.

Appendix A

A.1 False negatives

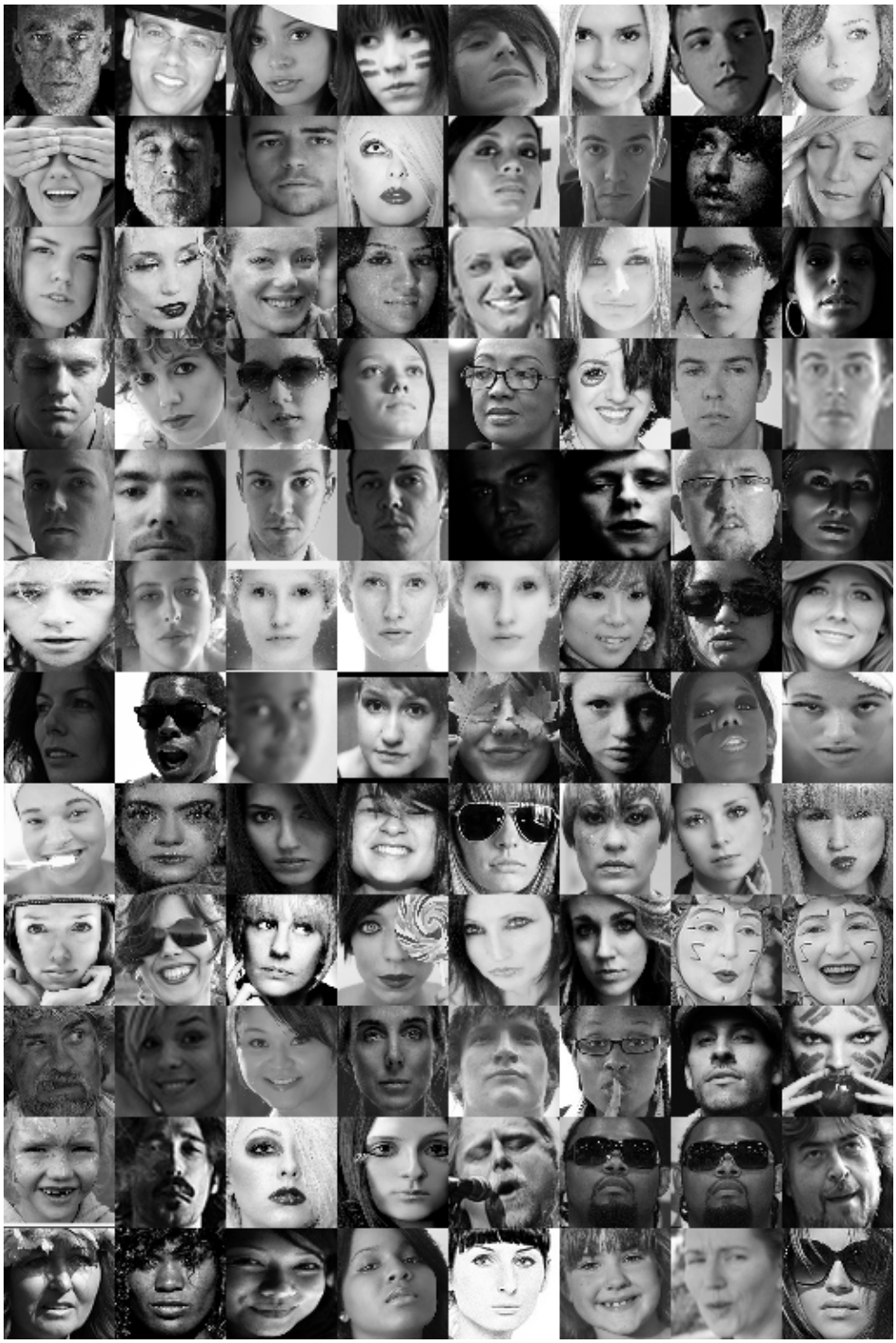


Figure A.1: 96 faces in the AFLW database that the Haar facedetector failed to identify

Figure A.2: 96 faces in the AFLW database that the HOG facedetector failed to identify

Figure A.3: 96 faces in the AFLW database that the CNN facedetector failed to identify

A.2 False positives

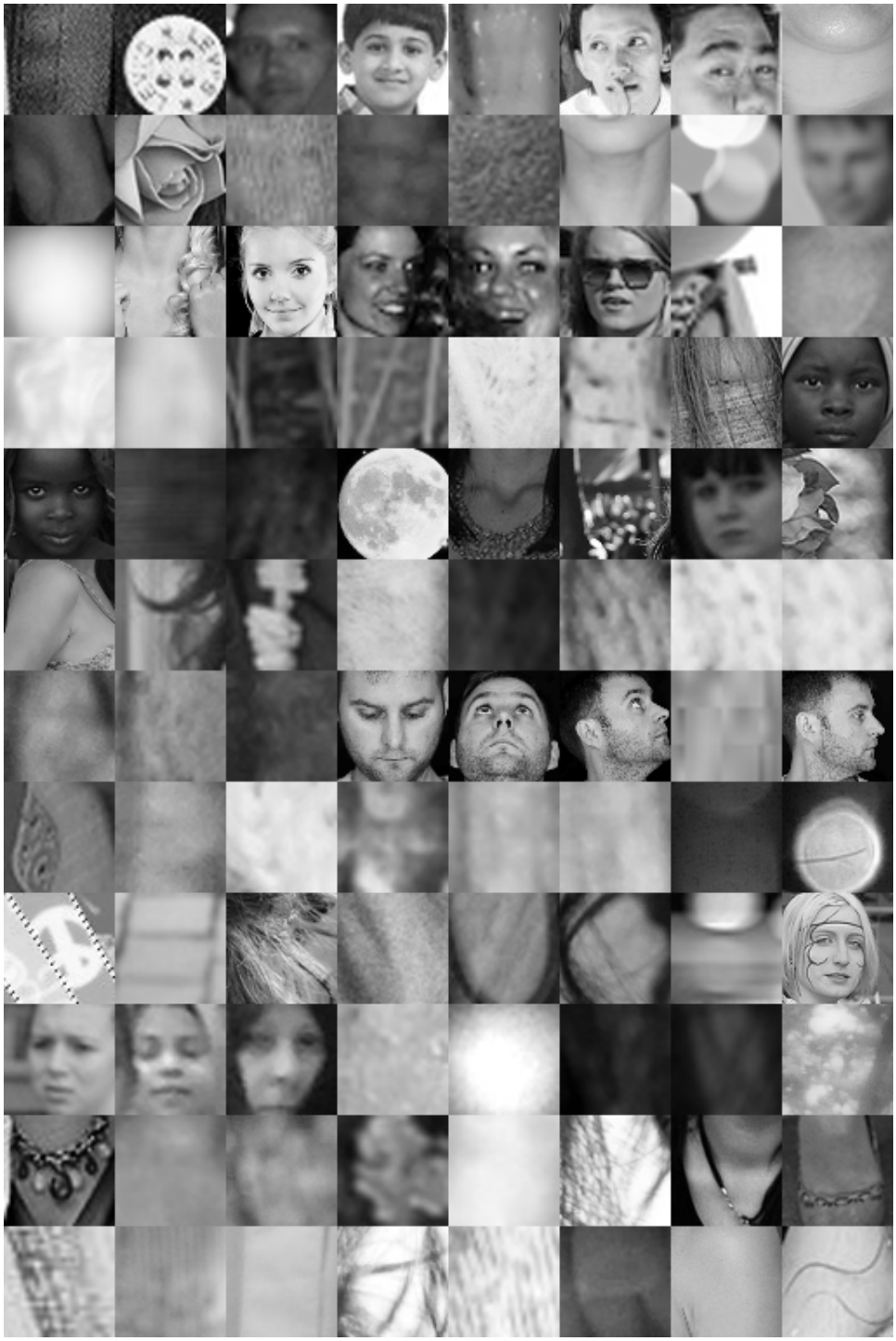


Figure A.4: 96 images that where classified as face by the Haar face detector but not annotated in the AFLW database

Figure A.5: 96 images that where classified as face by the HOG face detector but not annotated in the AFLW database

Figure A.6: 96 images that where classified as face by the CNN face detector but not annotated in the AFLW database

A.3 NVIDIA Titan X (Pascal) block diagram

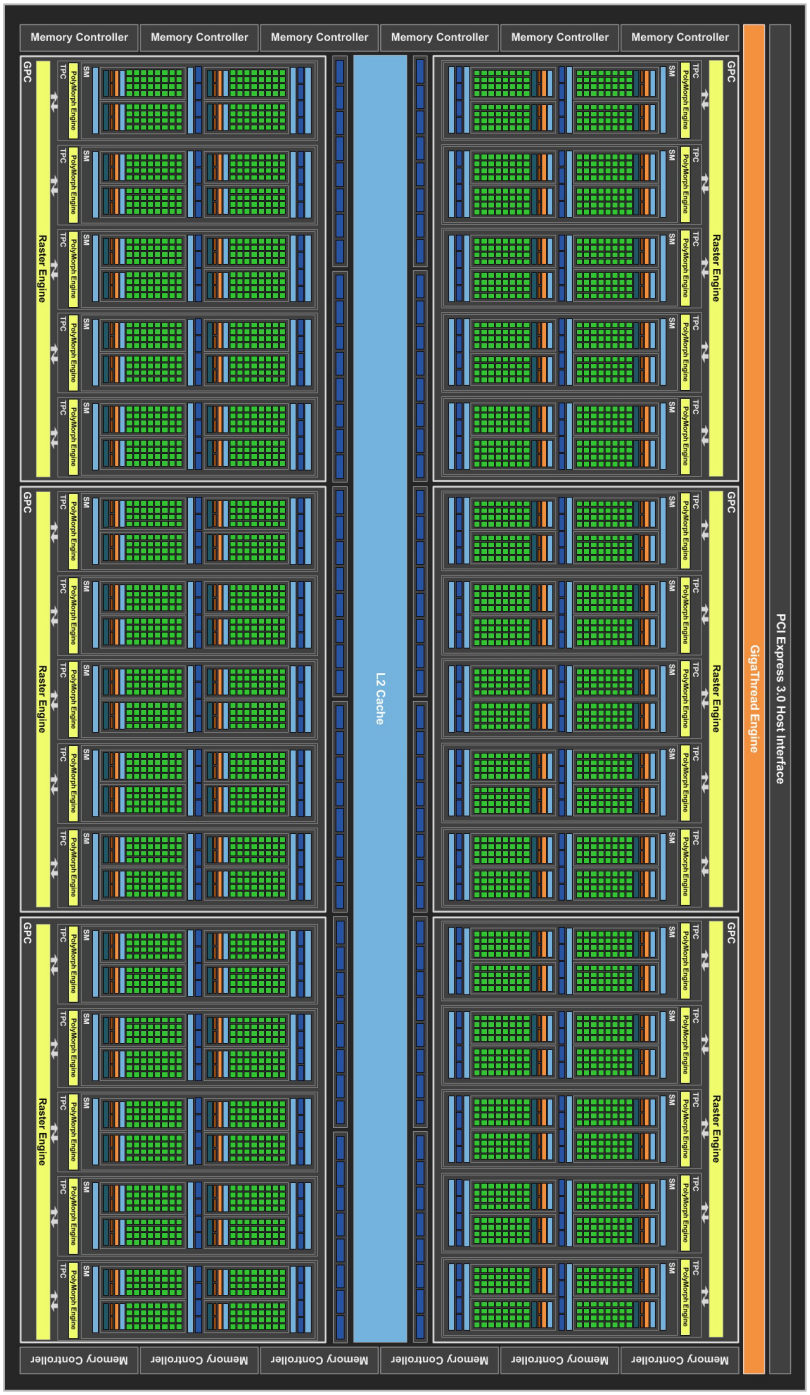


Figure A.7: NVIDIA Titan X (Pascal) chip block diagram. The GPU is based on the NVIDIA GP102 die with 2 SMs disabled. This leaves 28 SMs with 128 CUDA cores each, resulting in a total of 3584 CUDA cores.

A.4 Multiple stream NVVP output

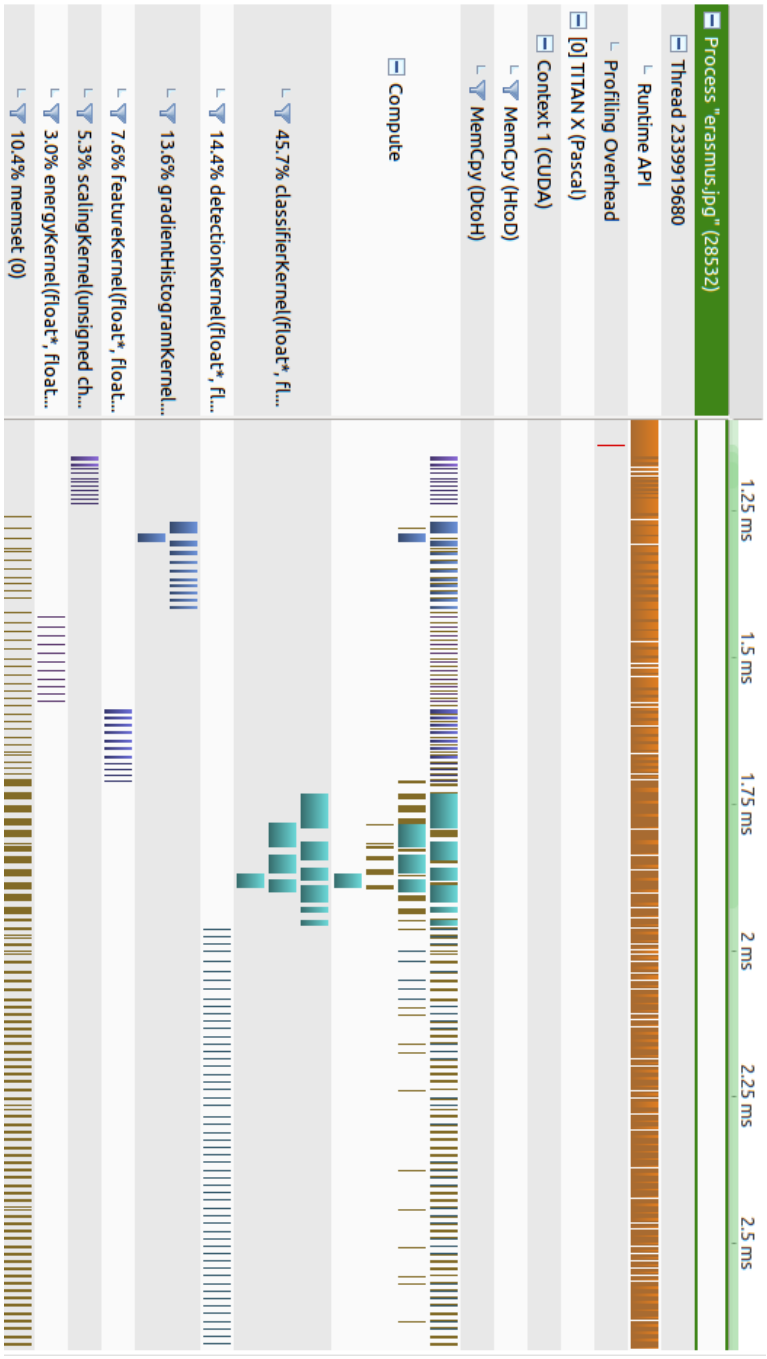


Figure A.8: Output of the NVIDIA Visual Profiler for the face detection implementation with seperate streams for each image scale. The multiple lanes in the 'compute' row indicate concurrent kernel execution, which only happens for the larger `gradientHistogram` and `classifier` kernels.

A.5 Multiple stream NVVP output for multi image implementation

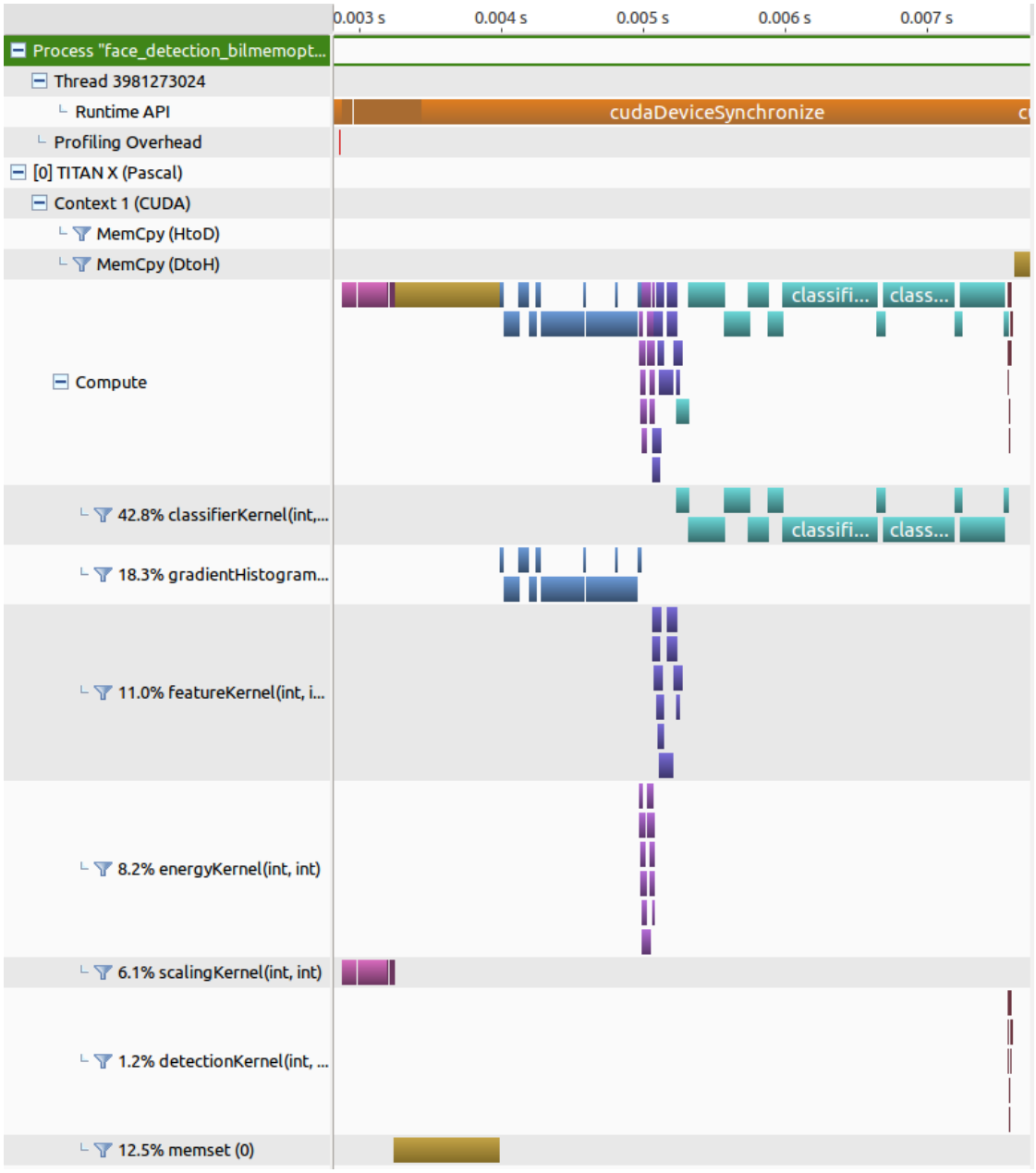


Figure A.9: Output of the NVIDIA Visual Profiler for the face detection implementation where the GPU is running face detection on batches of 16 images. Each image scale has its own GPU stream. The multiple lanes in the 'compute' row indicate concurrent kernel execution, which happens for the kernels that are too small to occupy all GPU SMs on their own.