Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Efficient Image Recognition on Low-Performance CPUs

Martin Jandek

Supervisor: Ing. Michal Sojka, Ph.D.
August 2020

Acknowledgements

I would like to thank my supervisor, Mr. Sojka, for his guidance through each stage of the process and for being patient and supportive during this research.

Declaration

I hereby declare that I have worked on this thesis independently and specified all the used information sources in accordance with the Methodical guidelines about following ethical principles during the preparation of university theses.

Martin Jandek
In Prague, 14. 8. 2020

Abstract

Many possible use-cases of slower performing, but small and cheap processors exist in the world of internet of things. The power of these processors are most of the time unutilized. One possible usage to better utilize this computing power could be an object detection in images or in a video computed directly on the low-performance processors (as an example we present a “smart” kitchen oven that could detect what kind of food it currently bakes). This thesis deals with optimization of such detection running even on very low-performing processors. We used the Haar cascade classifier algorithm implemented in the OpenCV library and ran it on the i.MX6ULL applications processor. After several benchmarks of the device and tests of the algorithm, we achieved increase in performance by about 10% and using some a priori knowledge about the object size, we demonstrated a 9.5× faster algorithm run.

Keywords: Haar cascade classifier, OpenCV, Buildroot, NXP, i.MX6ULL, GCC compiler, perf, Linux

Supervisor: Ing. Michal Sojka, Ph.D.

Abstrakt

Ve světě internetu věcí existuje spousta využití nepříliš výkonných, ale za to malých a levných procesorů. Výpočetní výkon těchto procesorů je většinu času nevyužitý a jedním z možných využití volného výpočetního času může být detekce objektů v obrázcích nebo videu (jako příklad zde můžeme uvést “chytrou” kuchyňskou troubu, která sama pozná, jaké jídlo v ní aktuálně připravujeme). Tato práce se zabývá optimalizací běhu takového algoritmu i na málo výkonných procesorech. Použili jsme OpenCV implementaci algoritmu Haarova klasifikátoru, kterou jsme testovali na modulu i.MX6ULL. Po několika benchmarcích zařízení a testování zvoleného algoritmu jsme dosáhli zrychlení běhu klasifikátoru přibližně o 10%. Při použití určitých apriorních znalostí ohledně velikosti objektu, který chceme detekovat, jsme demonstrovali běh algorithmu rychlejší přibližně 9.5-krát.

Klíčová slova: Haarova kaskáda, OpenCV, Buildroot, NXP, i.MX6ULL, GCC kompilátor, perf, Linux

Překlad názvu: Efektivní algoritmy pro rozpoznávání obrazu na levných embedded procesorech

Contents

1 Introduction 1

2 Background 3

2.1 Hardware equipment 3
2.2 Buildroot 3
2.3 Application profiling 4
2.4 Haar cascade classifier 4
2.4.1 Training of the Haar cascade classifier 7
2.5 OpenCV library 8
2.5.1 OpenCV implementation of the Haar cascade classifier 9
2.5.2 Detecting objects with the OpenCV Haar cascade classifier . 11

3 Experimental setup 17

3.1 OpenCV cross compilation 17
3.2 Preparation of the Linux kernel . 17
3.3 Buildroot usage 18
3.4 Training the Haar classifier 19
3.5 The testing program 20

4 Optimization of the OpenCV Haar cascade classifier 23

4.1 Analysis of possible optimization techniques 23
4.2 Hardware functionality testing and benchmarking 24
4.3 Function call-chain analysis of the detection phase run 26
4.4 Optimization using compilation parameters 29
4.4.1 OpenCV compile-time configuration parameters 29
4.4.2 GCC compiler optimization parameters 30
4.4.3 Results for differently-sized models 32
4.5 Code optimization 33
4.5.1 Analysis of the algorithm code 33
4.5.2 Optimization of main classifier's methods arguments 34

5 Conclusion 37

Bibliography 39

Project Specification 43

Figures

2.1 Haar features 5
2.2 Features used to detect the most significant areas of a human face . . . 5
2.3 Pixel intensities detected inside the Haar feature 6
2.4 Schema of the rejection cascade . . 7
2.5 OpenCV History Timeline 9
2.6 Intel's IPP optimization speedup for some OpenCV methods (image taken from the book *Learning OpenCV* by *G. Bradski* [18] 10
2.7 Example of the .xml file representing one node (strong classifier) of the Haar cascade classifier 11
2.8 Data structures representing the cascade classifier as implemented in the OpenCV 12
2.9 Fail of the histogram equalization, when the input image is a kind of an edge case – this whole image is too dark and does not contain many bright pixels 13
2.10 Success highlighting the edges by equalizing the histogram 13
2.11 Declaration of the *detectMultiScale* method 14
2.12 An example of so-called "neighbours" – multiple subwindows, where the object is detected, which overlays each other 15
2.13 Face detection in a park scene: even tilted faces are detected; for the $1,111 \times 827$ image shown, more than a million subwindows in different scales were searched to achieve this result in about 0.25 seconds on a 3 GHz machine (taken from *Learning OpenCV* by *G. Bradski* [18] 16

3.1 The very simple testing program used for the analysis and optimization of the Haar cascade classifier run 21

4.1 Part of the program used for memory benchmarking 25
4.2 Testing the actual sizes of device caches 26
4.4 Optimization of performance using the CMake and GCC parameters – measured on 10 images of size 640×480 using the "simple model" 31
4.5 Optimization of performance using the CMake and GCC parameters – measured on 10 images of size 640×480 using the "frontalface model" 33
4.6 Algorithm performance for few different minimal object sizes (maximum object size set to 600×440 – measured on 10 images of size 640×480 using the "frontalface model" 34

Tables

4.1 GCC optimization flags in different optimization level 31

Chapter 1
Introduction

People tend to have everything “smart” and “intelligent”. What can be more *smart* and *intelligent* than for example a cooking oven, which could recognize what is inside, set itself without any human interventions and notify us, when the food is ready? This would not be a problem, but we need to take one significant consideration – the price of such a device. The is no problem to load the kitchen appliances with high-end desktop-like hardware, but the price would be extremely enormous. Fortunately, many embedded systems exist, which are cheap and just the perfect match to be used in such devices. But even here lies a massive drawback and it is the computing power.

Some companies already gets to the segment of image processing on IoT devices like the kitchen appliances. Traditional approach to implementing AI to power these appliances has been to transmit data, for example the images, to a cloud platform where the data is processed on powerful processors and the results are sent back to the appliances. This approach is pretty functional and is used in many real applications. On the other hand, this approach means transmitting data *out* and when the data are sensitive, a solution without sending anything anywhere *out* is desirable. Because the processors in IoT devices are not usually powerful at all, we need to find an acceptable algorithm and try to optimize it to achieve satisfactory performance even on low-performance devices.

This thesis aims to find a suitable image-recognition algorithm that could be run on a very low-performance embedded systems. We discuss the ability to run such algorithms, provide a description how to setup an embedded system based on the i.MX6ULL device including all needed software. We also deeply analyze one of the image-recognition algorithms, benchmark it and enhance its performance on the device.

Reader of this thesis should be able to setup an embedded system like the one used in this project and prepare the software based on recommended settings arised from this work. Even on a low-performance embedded system, he should be able to get a very well-performing object detection.

Chapter 2

Background

This Chapter provides background information about technologies used in this thesis, which is necessary for understanding the follow up Chapters. In this Chapter we discuss the i.MX6ULL manufactured by NXP as the device to be used. We also acquaint the reader with Buildroot, OpenCV open-source library, Haar cascade classifier and its OpenCV implementation.

2.1 Hardware equipment

During this project, we use NXP's "cost-optimized" ultra-efficient i.MX6ULL applications processor. It features a single-core Arm Cortex-A7 CPU operating at 528 MHz, 250780 kB of memory (exact size obtained by executing *cat /proc/meminfo* on Linux system), 32 kB of L1 instruction cache, 32 kB of L1 data cache and 128 kB of L2 cache. Also, this system on a chip (SoC) features a NEON co-processor, including 32-bit double-precision VFPv3 floating point registers [9].

Compilating all the massive libraries directly on the target hardware would be impossible due to the low amount of memory, so all supportive programs, the used libraries, and the kernel were cross-compiled on a UNIX-based laptop using the x86 instruction set.

2.2 Buildroot

Buildroot is one of the most widely used tools simplifying the whole process of building an embedded system using cross-compilation. Buildroot masks the whole building process, so it become much more convenient. It can generate a toolchain file used for cross-compilation, build a root filesystem, a Linux kernel image, and a bootloader. It is based on the build tool Make and, by default, requires only a few Linux packages (a list of mandatory and optional packages can be found in the official documentation [2]).

Over two thousand packages (from simple text libraries to advanced graphical and network libraries and programs) are available to install to the embedded system directly with a Buildroot. Even the OpenCV, which we use in this thesis, can be installed through the Buildroot directly, without any needs

to compile the library manually. Buildroot comes with its configurations and one can configure the whole wanted target system to his needs. *Make menuconfig* is a tool used to set the specific configuration. This tool uses the classic *ncurses* interface and Buildroot comes with more similar tools, for example, *make xconfig* that uses a Qt-based interface. The classic interface is the same one that is used to configure the build of a Linux kernel. During this configuration, Buildroot manages the whole process, including download and install of all wanted packages to be used in the target system. Buildroot documentation [2] is very well-written, so if some details are needed, that is the right place where to look.

2.3 Application profiling

Application profiling is a necessary step before any tweaks or optimization experiments and one of the simpliest and most widely used tools is a Linux' package *perf*. Perf is a performance analyzing tool distributed inside the codebase of the Linux kernel. Because of that, perf has much smaller overhead than any other application-based profiling tool, which does not have the direct access to the hardware as the kernel itself. To understand the location of performance bottlenecks, the perf tool needs to store backtrace information of the running application for later analysis. One way is to use frame pointers. This approach is used not only for storing the call-chain but is also used by the program itself and a debugger to establish an address of local variables in memory. A downside is that it affects runtime performance. Also, most programs use a compiler option to omit frame pointers (which prevents debuggers to perform stack unwinding in a simple way) by default. Another approach is to use the *libunwind* library, which uses DWARF debug information (produced by the compiler) to unwind the stack. [17].

2.4 Haar cascade classifier

Haar cascade classifier is a tree-based object detection technique first introduced by Viola and Jones [32] mainly used for a performance-critical image recognition as an alternative to the nowadays popular neural nets. We can divide the algorithm into two phases. The training part and the testing (= classification = detection) part. A model, which represents the decision-making of the classifier, arises from the training part (more on this in the Section 3.4). It was introduced mainly as a face detector but is actually capable of detecting any "mostly rigid" object (faces, cars, humans, food on a uniform background, ...) [18]. Why "mostly rigid" objects? "Mostly rigid" object contains parts that are, independently on the one particular object, the same. For example, we can say that a car is a rigid object, because all cars have wheels at the bottom, a hood in the front, a windscreen adjoining the hood, and pretty much the same contour. The same we can say about a human face. All faces contain two eyes in the same height – one eye some-

where in the top-left part and a second eye somewhere in the top-right part and also a mouth somewhere in the bottom-center part. We can define these static parts of the object by “features”. “Feature” describing one part of the object (for example a feature describing a mouth in the image of a face) is a static rectangular region containing a relevant information about a part of the object. In our case, a feature is a region containing a sharp contrast between pixel intensities (in the human face example, we can say that one feature represents the upper lip). Let us show an example of few features describing the human face in the Figure 2.2. As Viola and Jones firstly introduced the algorithm, it worked only on the base of upright position features. In the Figure 2.1, we can see even tilted features. Later on, Lienhart and Maydt introduced even features rotated by 45 degrees [24], which lead to the increase in precision of the detector.

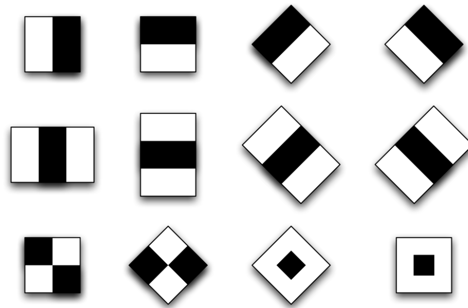

Figure 2.1: Haar features

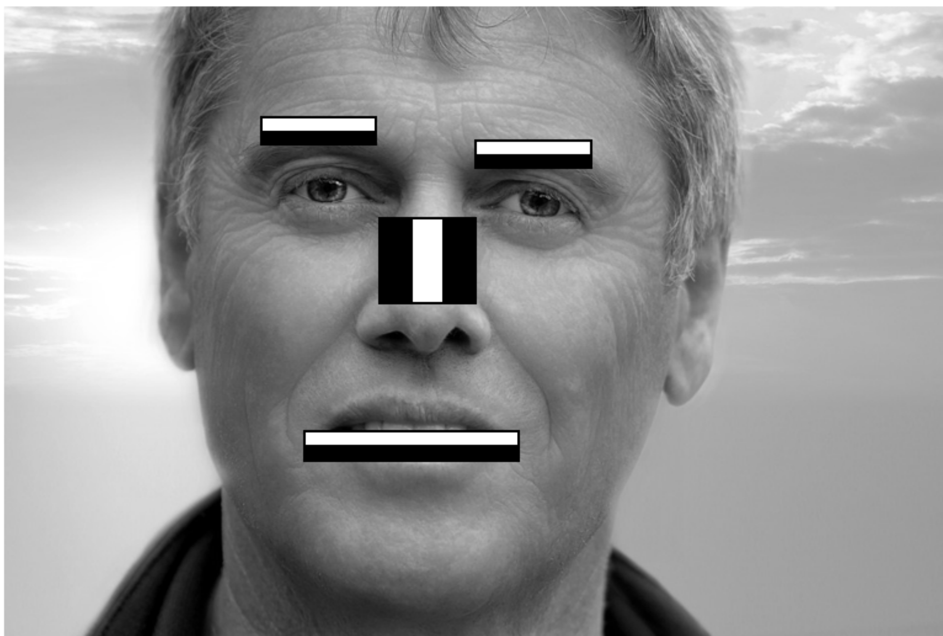

Figure 2.2: Features used to detect the most significant areas of a human face

As we can see in the Figure 2.3, each feature contains many pixels rep-

resented by their intensities. Because we want to reduce the information needed to detect the object as much as possible, we cannot have only the pixel intensities, but we also need something to represent that specific feature. This let us remember only the few features, respectively, their location in the image and the number representing the feature. A feature is, in our case (a feature is, in the image processing, a general concept, so we will define the feature only for the usage in Haar cascade classifier), then represented by a single number computed as the difference between sums of pixel intensities in the black area and in the white area (2.3). We can compute this number from any feature (a rectangle of any size) in the input image. It means that the number of all possible features in the input image is significantly higher than the total number of pixels in the picture. Let us say we have a picture of size 20×20 pixels. Now we can take all rectangles of all sizes, so we get $m*n+m*n+C(m*n,2) = 2*m*n+C(m*n,2) = 20*20+20*20+79800 = 80600$ features, where m is the height of the image, n is the width of the image and C is a function computing the number of combinations.

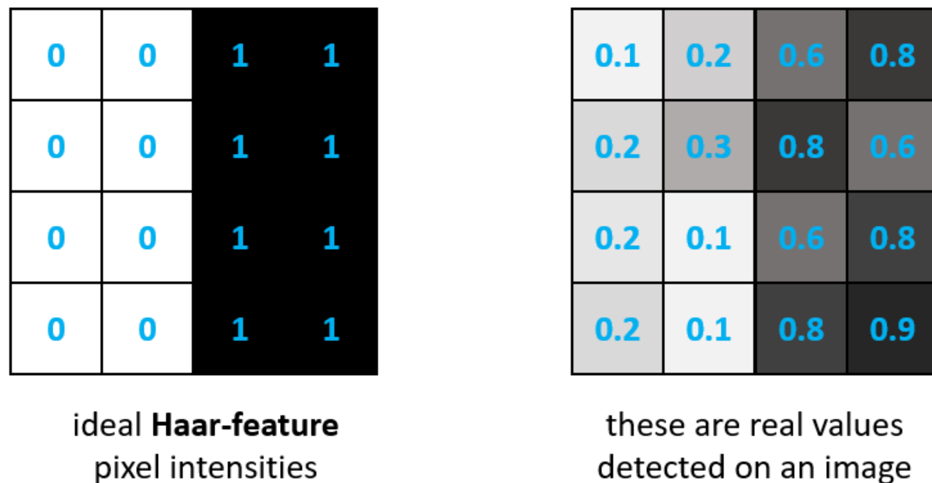

Figure 2.3: Pixel intensities detected inside the Haar feature

That does not sound like we have reduced the information needed to represent the object, does it? The Haar cascade classifier uses the Adaboost algorithm to select only the optimal (minimal) features needed to detect the wanted object (with the set probability – false negative detecion) in the training part of this algorithm. Adaboost [26] is a machine-learning algorithm, which takes a great number of weak classifiers as input (in our case, it takes all the features as input) and returns a strong classifier. This strong classifier is a weighted sum of the weak classifiers from the input (in our case, it returns the weighted sum of only the minimal number of features). This means we finally reduced the information needed to describe an object. For the next description, we refer to the figure 2.4. Each feature contains its weight (rejection rate – it means how important is the particular feature to represent the object) and the features are then grouped into so-called nodes. Each

node carries its own Adaboost classifier composited from a weighted sum of few features (we can say that the feature plays the role of a weak classifier in here and the whole node is the strong classifier) and the nodes put together the so-called rejection cascade [18]. This sums up the representation of the classifier model.

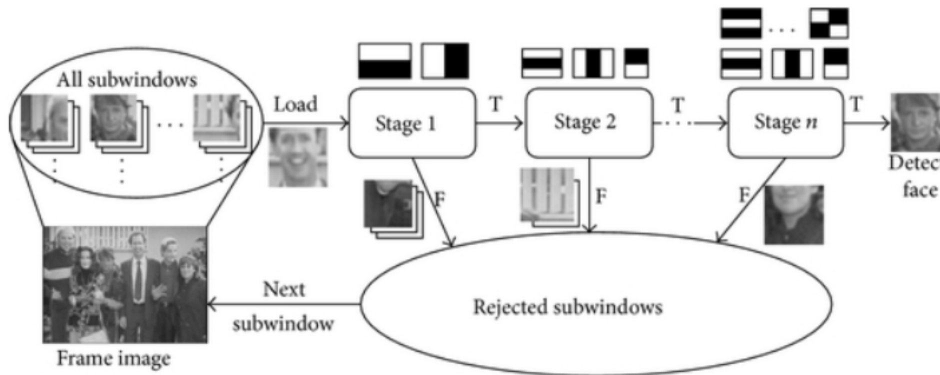

Figure 2.4: Schema of the rejection cascade

When it comes to the classification part, the Haar cascade classifier uses the *shifting window* algorithm that takes a subwindow (= part of the image, as we can demonstrate in the Figure 2.4) and the subwindow goes through the rejection cascade. *Shifting window* algorithm means that it takes a window of a particular size and iterates through the whole image (x and y coordinates), where it takes all possible windows of that size. As said, each node is a strong classifier that outputs *True* (contains the wanted object) or *False* (does not contain the wanted object). Each subwindow must pass through all the nodes and if the subwindow contains the object, all nodes in the cascade must return *True*. If *False* is returned from even one node, the subwindow is rejected and does not pass the next nodes. Each node is trained to have a high detection rate (almost 100%, usually about 99.5 - 99.9%), but also a pretty high rejection rate (almost 50%). We can afford these high rejections rates, because when we layer the nodes into the cascade, the rejection rate is greatly reduced. When one has 20 layers (20 strong classifiers = 20 nodes = 20 Adaboost classifiers) in the rejection cascade, the rejection rate of the whole cascade drops to $0.5^{20} \approx 0.000001$, and the detection rate of the whole cascade stays pretty high on $0.999^{20} \approx 0.98$ [18].

2.4.1 Training of the Haar cascade classifier

Two main approaches to the classifier training exist – we can either train the classifier on a single image and generate a dataset from this image, or get a huge dataset of different images (images of both the object and many possible object's background images). As for the first approach – single image – we take for example a logo or some other static image (usually a 2D image) and

apply many possible distortions to it. Rotate it, make it partly transparent, place other objects in front of a small part of it, blur it, just whatever comes to mind. These distortions creates tens or maybe even hundreds of positive training samples from just one image. The other approach is to collect a massive dataset of images (for example cars, faces, food) and train the classifier on this dataset. Having the dataset of positive images (which are images containing the object we want, ideally images of only the object without any background) is not enough. We have to provide the algorithm even negative images (images not containing the object) and images to be used as a background for the object. When we need to train the algorithm to detect a car, we have to provide it even images of planes, motorcycles, etc. as negative samples. To train even better performing classifier, one should generate images of the object on different negative images used as the object's background and (same as in the single image approach), apply some simple distortions to the object. We cannot just put the different positive images into the training algorithm and let it, but it is necessary to align the images. For example, when training the classifier to detect a human face, we need to align the eyes, the nose and the mouth to be in the same location on all the samples. They do not have to be at the *exact* same location in all the samples, but preferably as close as possible. We can say that without this alignment, we try to teach the classifier that eyes do not have to be in the same location in a face, but practically anywhere inside the face region [18]. In the Section 3.4, we describe the particular training process as we executed it.

2.5 OpenCV library

OpenCV is an open-source library written in C++ providing an optimized implementation of many algorithms and data structures from basic image data manipulation to deep machine learning techniques. In 1999, Gary Bradski (an Intel employee at that time) started collecting well-optimized code provided mostly by students and student groups. During his visits to some prestigious universities, he realized that most student groups have at least some shared codebase, which passes from student to student so that their project does not start totally from scratch. These libraries were mostly very well-optimized for the usage of the research. That was the most significant moment, which led to the very beginning of the whole OpenCV project. We show the main releases of this library in the timeline in the Figure 2.5.

OpenCV nowadays collects libraries for a broad spectrum of computer vision procedures. The maintainers of the main codebase tries to ensure that only well-optimized (by various approaches) code of high quality gets to be merged into the library itself. One specific example could be a part of Intel's proprietary IPP low-level optimization (figure 2.6), which was provided by Intel to be used freely in the OpenCV mainline. Although the idea of OpenCV was to provide only the best free-of-charge optimized computer vision algorithms, many parts of the library are obsolete and not that well

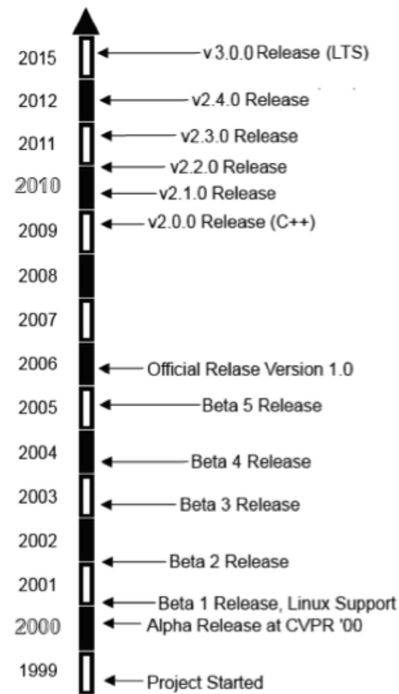

Figure 2.5: OpenCV History Timeline

optimized.

Google, Microsoft, IBM, and Samsung are some of the most notable users (and also contributors) of the OpenCV project. The most remarkable contributor is still Intel, respectively, its sub-company Itseez, who also did the most work during the starting part of the library. [18]

2.5.1 OpenCV implementation of the Haar cascade classifier

In the OpenCV library, the whole Haar cascade classifier is implemented in the *cv::CascadeClassifier* class [10], where it provides both the methods to load/store the trained model and to run the detection phase of the algorithm. OpenCV library provides its application to train the classifier and generate a *.xml* file containing its description. Although we trained our own simple model to try and test the application, for the testing purposes (practical training of the classifier is described in the Section 3.4), we used both the pre-trained model located in the official OpenCV repository (specifically, it is located in the *OPENCV_ROOT/data/haarcascades/haarcascade_frontalface_alt.xml*) and our simple model (more on the topic of different model sizes in the Section 3.4).

In the OpenCV library, the classifier is implemented as a vector of stages and a vector of internal nodes. In the Section 2.4, we defined the nodes as the objects containing the Adaboost strong classifier inside. To match the OpenCV naming, we will now call these groups of strong classifiers *stages* and the numbers representing the features itselfs *internal nodes*.

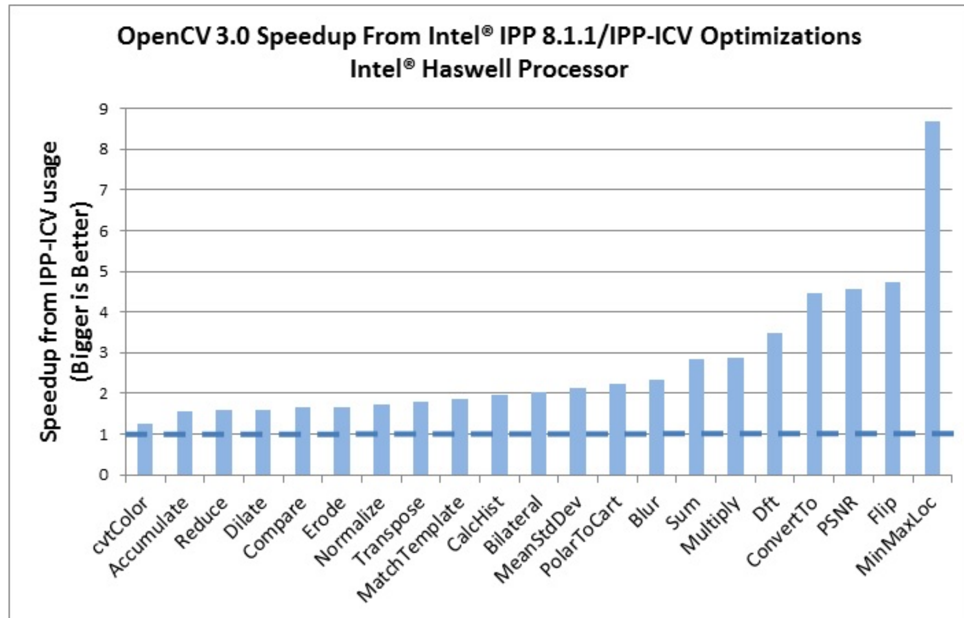

Figure 2.6: Intel's IPP optimization speedup for some OpenCV methods (image taken from the book *Learning OpenCV* by *G. Bradski* [18]

The following text is accompanied by the Figure 2.7, where we can see the .xml representation of a single stage (a strong classifier) containing three weak classifiers. Each stage has an associated threshold. By comparing the classifier output with the threshold, the algorithm determines whether the region contains the wanted object or not. Each weak classifier inside the strong one (the internal node) also has an associated threshold. The weak classifier refers to two other weak classifiers (one rejecting the subwindow and one accepting the subwindow – the subwindow contains the object or do not contain the object), where it propagates the value. We can use the Figure 2.4, but instead of the whole stages, all the weak classifiers inside can also reject the subwindow. A sum of all values (assigned during the pass through a weak classifiers is tested against the strong classifier threshold (stage threshold) and possibly rejected (depends on the threshold) as not containing the object. Subwindow decided as containing the wanted object has to be accepted by all the strong classifiers and all the weak classifiers inside. All the values whose sum we test against the strong classifier's threshold are specified during the training phase.

Let us now describe the classes and the data structures used. As we have already mentioned, the whole cascade classifier is implemented in the *objdetect* [10] module. More specifically, the *cascadedetect.cpp* and *cascadedetect.hpp* files. Training output of this classifier is a *.xml* file representing the cascade (containing values of minimal hit rate, maximal false alarm rate, number of stages, size of the wanted object, strong and weak classifiers). We can find the data structure holding these parameters in *cv::CascadeClassifierImpl::Data*. We can see the implementation of this data structure in the Figure 2.8. As

```
...
<maxWeakCount>3</maxWeakCount>
<stageThreshold>-1.5401084423065186e+00</stageThreshold>
<weakClassifiers>
    <_>
        <internalNodes>0 -1 7 -6.7867025732994080e-02</
    internalNodes>
        <leafValues>6.0000002384185791e-01 -9.5112013816833496e
    -01</leafValues>
    </_>
    <_>
        <internalNodes>0 -1 2 4.4192366302013397e-02</
    internalNodes>
        <leafValues>-9.2392551898956299e-01 2.9810953140258789e
    -01</leafValues>
    </_>
    <_>
        <internalNodes>0 -1 17 7.2595225647091866e-03</
    internalNodes>
        <leafValues>-8.7738829851150513e-013.3493724465370178e
    -01</leafValues>
    </_>
</weakClassifiers>
...
```

Figure 2.7: Example of the .xml file representing one node (strong classifier) of the Haar cascade classifier

we have stated before, *struct Stage* contains the strong classifier (respectively only the threshold value), *struct DTree* carries the number of nodes (number of weak classifiers) and *struct DTreeNode* contains the node (weak classifier) threshold, references (indexes) to two other nodes and a *featureIdx*, which represents the particular feature. Features have their own unique index, just like if they were in an array. That way we do not need to store the coordinates or any other information regarding the feature other than its index. For each node, the *featureIdx* is used to evaluate the actual feature value inside the *HaarEvaluator* class. If a stage contains only one node (it means that the depth of the binary decision tree is 1), the structure representing the stage is named *Stump*.

2.5.2 Detecting objects with the OpenCV Haar cascade classifier

We have already analyzed the data structures storing the classifier, so now, let us have a look at how the algorithm executes and its call graph. Many books and articles are dealing with the usage of the OpenCV library [19, 21], so we will mention only the necessary knowledge to analyze the code.

```
class Data
{
    public:
        struct DTreeNode
    {
        int featureIdx;
        float threshold; // for ordered features only
        int left;
        int right;
    };

    struct DTree
    {
        int nodeCount;
    };

    struct Stage
    {
        int first;
        int ntrees;
        float threshold;
    };

    struct Stump
    {
        int featureIdx;
        float threshold;
        float left;
        float right;
    };
}
```

Figure 2.8: Data structures representing the cascade classifier as implemented in the OpenCV

The only method needed for the detection is *cv::CascadeClassifier::detectMultiScale*. We show the declaration of this method in the Figure 2.11. It is called on the object of class *CascadeClassifier* that loads the cascade model from a *.xml* file. This method takes as parameters an input image, where we want to detect the object, and a vector of the object (defined by x, y coordinates and its x and y sizes) as detected by the classifier (or *detector*, because the algorithm as is only detects the object and returns the coordinates of it in the input image). What is done before or after calling this method is irrelevant and depends on the desired pre-processing and post-processing. For example, we could grab the returned vector of object's coordinates and show the input image with them color drawn on it. Before running the detector, input images are often pre-processed by *equalizeHist*. It boosts the image contrast, so it makes all edges in the image to be much more significant and so easier to detect any solid-shaped object. In the pictures 2.9 and 2.10, we can see, what the *equalizeHist* does with the input images. On the first comparison, the input image was too dark and had the space of pixel intensities too narrow. In the

output image, we can see that the histogram equalization made the picture even worse, because it tried to make the space of pixel intensities even and the whole spectrum of dark pixels was almost randomly brightened. On the second comparison, the equalization was a good step, because the edges in the input image is well highlighted. We should remind that the first comparison is really an edge case and in most cases, the histogram equalization does a good job pre-processing the detector input image.

Note that in the real world one should pre-process the images at least with some simple pre-processing methods before the cascade classification algorithm takes its place in the program run to get some serious results. The "golden standard" is to resize the images to a uniform size. For smaller images, the classifier would have a better performance in terms of processing speed, but we lose some of the details. It means that an object, which was of a smaller size before the resizing, would be much harder to detect, because of a low amount of pixels covered by it and, e.g., low contrast of its edges. Another approach to gain even better performance could be to remove unnecessary borders of the image, which often is just a background of a single color. This can be done, for example, by a combination of the Hough transform and the Canny edge detector [23].

Figure 2.9: Fail of the histogram equalization, when the input image is a kind of an edge case – this whole image is too dark and does not contain many bright pixels

Figure 2.10: Success highlighting the edges by equalizing the histogram

```
cv::CascadeClassifier::detectMultiScale(
    const cv::Mat& image,  // Input (grayscale) image
    vector<cv::Rect>& objects, // Vector of detected objects
    double scaleFactor = 1.1, // Factor between scales
    int minNeighbors = 3, // Required neighbors to count
    int flags = 0, // Flags (old style cascades)
    cv::Size minSize = cv::Size(), // The smallest of the object
     we will consider
    cv::Size maxSize = cv::Size() // The largest of the object
    we will consider
);
```

Figure 2.11: Declaration of the *detectMultiScale* method

First, the *detectMultiScale* calculates all possible scales of the wanted object (we want it to detect objects of any size – for example a small face and a big face). These possible scales are stored in a vector called *scales*. The whole object-finding process works with an image in grayscale, so if the input image is not already in grayscale, it converts it into grayscale. Next comes the computationally demanding part, which can be parallelized with the *pthread* library, the *TBB* library, or even the *OpenMP* library (depending on the availability and compilation parameters set). The implementation is straightforward and almost naive, only using sophisticated data structures (as described in the Section 2.5.2). Entire *shifting window* algorithm is implemented in the *operator()* method of *cv::CascadeClassifierInvoker*. For each scale of the wanted object, it runs the *runAt* detection method on all points at (x, y) coordinates. Simply filed, we cycle through all x and y coordinates of the input image, which tell us whether this particular subwindow (at coordinates x, y with a size set by the scale of the wanted object) contains the requested object. In the algorithm 1, we can see the pseudocode of the described shifting window algorithm. The 'weight' parameter of the *runAt* method is a bit wrongly named, but it is the value, which was compared to the last stage threshold during the rejection cascade run. If the library is compiled to be run on a multi-thread CPU, the *operator()* method is called in parallel. The *runAt* method performs the detection. In the Section 2.5.1, we have outlined the tree search executed in the Haar cascade classifier. The *predictOrderedStump* method (or *predictOrdered*, depending on the size of the tree), which is called in the *runAt* method, implements that tree search. If the *runAt* method returns *True* (contains the desired object), the subwindow is saved inside the output vector, waiting for the aggregation phase.

The result of the *detectMultiScale* method is that the *objects* vector is filled with subwindows, where the wanted object was detected. Sometimes, there can be a more significant number of subwindows right next to each other (e.g., a face is detected in all subwindows ±10% on an x axis and ±10% on an

Algorithm 1 Haar cascade classifier – Shifting window algorithm

```
1: for scale in scales do
2:     for x in image width do
3:         for y in image height do
4:             runAt(HaarEvaluator, Point(x, y), scale, weight)
5:         end for
6:     end for
7: end for
```

y axis). All subwindows are then grouped and filtered based on the “number of neighbours” criteria (for example, eliminate all rectangles without at least 3 *neighbours* — possibly a fault detection). Neighbours of a subwindow are other subwindows that overlaps a significant area of it – it most likely means that it is the same object, but detected in more than one subwindow (an example of multiple neighbour subwindows is presented in the Figure 2.12). In the Figure 2.13, we can see the detected objects rendered as white rectangles into the input image.

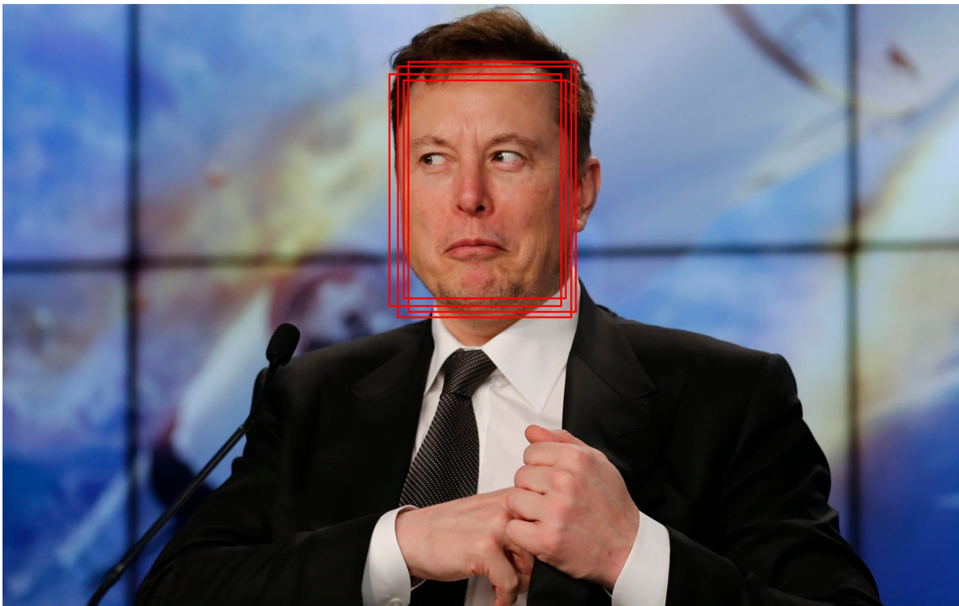

Figure 2.12: An example of so-called “neighbours” – multiple subwindows, where the object is detected, which overlays each other

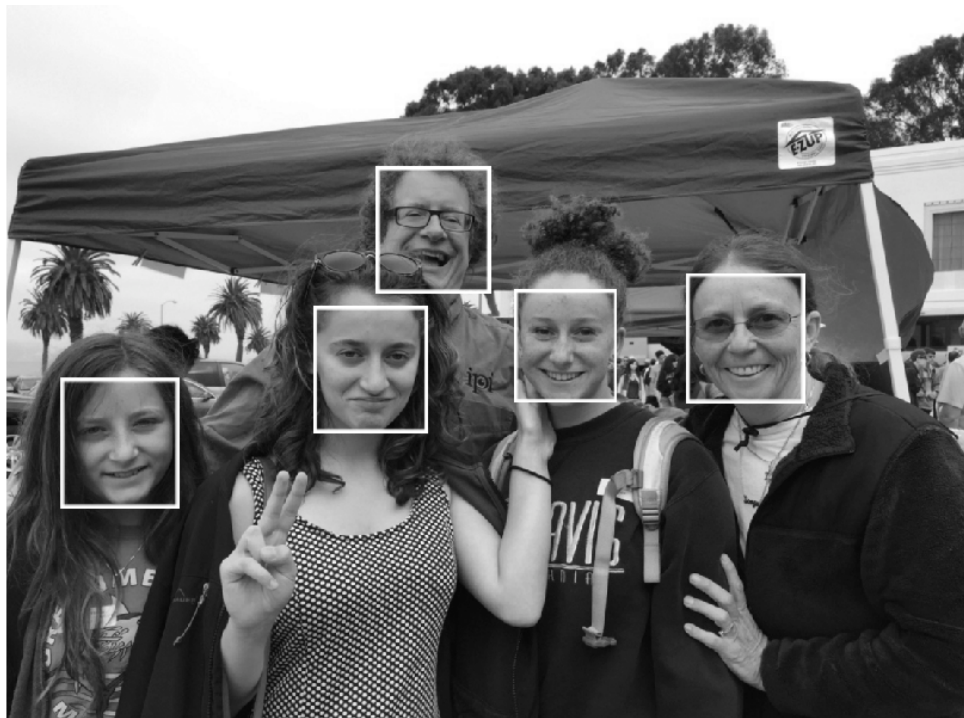

Figure 2.13: Face detection in a park scene: even tilted faces are detected; for the $1,111 \times 827$ image shown, more than a million subwindows in different scales were searched to achieve this result in about 0.25 seconds on a 3 GHz machine (taken from *Learning OpenCV* by *G. Bradski* [18]

Chapter 3

Experimental setup

In this Chapter we describe the work needed to prepare the testing environment – compilation of the used library, build of the whole system, and generating a model for the Haar cascade classifier to be used to detect objects in images.

3.1 OpenCV cross compilation

We compiled the OpenCV library, version 3.4, using the GCC ARM cross-compiler (specifically the 'arm-linux-gnueabihf-gcc', version 10.1.0), enabling the hardware support for floating-point instructions. The only prerequisites for the build are CMake, the ARM cross-compiler, the *pkgconfig* package, and Python 2. The SoC we were using does have a NEON co-processor, and it has hardware floating-point instructions support. For example, on the Ubuntu Linux distribution, two different cross-compiler packages can be installed. The 'arm-linux-gnueabi' compiler sets the *-mfloat-abi* parameter to *soft* by default, so that the generated code does not contain floating-point instructions, and should be used when compiling for target hardware that does not have a floating-point co-processor. The other ('arm-linux-gnueabihf') the *-mfloat-abi* parameter to *hard* by default, so one does not need to remember setting it to hard when compiling a specific program or library. In all following experiments, the library and all supporting programs were compiled using the 'arm-linux-gnueabihf'.

3.2 Preparation of the Linux kernel

For this project, we used an older version of Linux kernel, version '4.1.15_gtm3-1.0.1'. We were not able to make the latest mainline kernel run on the hardware used. Even with different kernel configurations, we did not determine what could be the problem that the latest kernel does not boot on the system used. Since we needed a functional kernel (there are not many changes in the kernel that could affect this research), we used a custom NXP's kernel based on version 4.1., which was already tested on this hardware and was working well.

A device tree is a data structure describing the individual hardware components and how they are interconnected, so we need it working and configured to the i.MX6ULL. The device tree is passed to the Linux kernel during the booting process, so that the kernel knows, where to find the hardware components, such as the CPU, the memory or peripherals. Before the usage of device trees, the Linux kernel had a hard-coded hardware-specific code inside its codebase. With the growth of different hardware supported, there was an effort to move all the hardware-specific code outside the kernel source code. The kernel codebase still contains pieces of hardware-specific code, but the device tree specifies, whether to enable some functionality or not. The kernel official git repository contains device trees for an extensive list of hardware supported [31], but even the external provided device trees (outside the kernel source code) can be used [7]. The device tree we use for this project is a NXP's proprietary one written specifically for this SoC, but it has not been well tested yet, so we cannot be sure that everything works as it should. Device tree distributed along with a Linux kernel in its repository is not yet available for this SoC.

We realized that the Linux tool *perf* does not show any hardware-specific events it could have possibly track, but we knew that the Cortex-A7 should have a PMU inside, so there should be a possibility to track many different events. After some research, we discovered that although the device tree used contained the PMU, it was disabled. We changed the status field from *disabled* to *okay* (to allow the operating system access to the unit) and recompiled the device tree. This discovery, though, raised a question about the proper functionality of the PMU. If the device tree we use is a NXP's one written for this SoC, it should enable all the functionality. They could have found a bug with the PMU enabled, so they could have just disable it to fix it later. For the proper continuation of this project, we have to test the functionality of the PMU and if there is a bug, then find it. Using the Linux *perf* tool, we will test the values captured from this unit in the Section 4.2.

3.3 Buildroot usage

We already introduced the Buildroot as a tool, so here we can raise the specific steps we made to run the system as a whole.

We used a different Linux kernel than the ones provided by Buildroot itself and used Buildroot mainly to simplify the system build. Buildroot supplies the latest Linux kernel plus a few older ones. Buildroot can be configured to use not only the provided kernels, but also a kernel from a git repository. Buildroot will then will then fetch it, use the configured kernel parameters and compile the kernel right from the repository. Specifically, we used kernel version 4.1. with some hardware vendor (NXP) supplied additional patches. The main change made in the Buildroot regarding the kernel is enabling the Linux tool *perf*.

Other necessary preparation was to install the *libunwind* library, which is described in the Section 2.3. We can find the library in the packages provided

by Buildroot, so all we need to do is configure the Buildroot to include this library into the target system.

Other hardware-specific configuration of the Buildroot has to correspond to the actual device used. So we set the ARM target architecture with little-endian byte order. We can set the Cortex-A7 CPU as the "Target architecture variant". Also, since the Cortex-A7 supports NEON and VFPv4 instruction sets, we allow the use of these. Buildroot supports many different architectures and a vast number of packages to be installed, a situation that a particular package cannot be installed to a particular architecture often happen. Buildroot uses inside dependencies, which prevents the conflicting packages to be installed. All options are well described in the documentation [2].

3.4 Training the Haar classifier

Before we can optimize the classification part of the Haar cascade algorithm, we need to train the classifier and use the result of the training, the model. OpenCV offers some pre-trained models in the OpenCV tutorial (distributed in the same git repository as the source codes), so, for example, eyes or face recognition works well with the provided models. Also, because of the wide usage of OpenCV, one can find a pretty well pre-trained model for almost any meaningful usage (but still, most use-cases are the face, eyes or human body detection) [16, 29]. OpenCV tutorials and documentation present a pretty straightforward approach to how to train the classifier. Since the latest OpenCV version (version 4.3 at the time of writing this thesis) does not contain the applications for cascade classifier training pre-processing (the *opencv_annotation* and *opencv_createsamples*), we use the OpenCV version 3.4. These two applications make our work much easier, since they generate the files needed for the training part and we do not have to write the generating scripts manually. These applications were removed in particular because of the old C-based API. One of the OpenCV contributors stated [6] that the algorithm (Haar cascade classifier) is deprecated when compared to the deep neural networks (DNN). The classifier has been preserved due to a number of books using it as its base knowledge. Although, DNN is out of scope for this work, since its hardware requirements are extremely high and DNN are not to be used on low-performance devices. No significant algorithm changes that could affect the testing process were made since the version 3.4, so we will use this version not only for training, but for all the upcoming work.

The official OpenCV tutorial [3, 18] documents the whole process very well, so we will just quickly go through the mentioned OpenCV applications usage for the training. First, we need to obtain the negative images and create a text file describing the path (can be absolute or relative) to each image containing one path per line. Next, we need the positive samples. For that, we can use the *opencv_annotation* tool. We need to provide the algorithm specific coordinates, where the object to be detected lies in the image (on

the x and y axis). The annotation tool provides a graphical interface to annotate a bigger number of images fast and simple. It shows images (from a folder path passed as argument to the application) sequentially and we only need to draw a rectangle, where the object is. It generates a similar file to the one used for the negative samples, but each image has its own description about how many objects it contains and where the objects are (x and y coordinates of the top left corner with its width and height). Having the text file describing the positive samples, we use the *opencv_createsamples* application to generate a binary file, which is then passed to the Haar cascade classifier for the training process. The application used for training itself is called *opencv_traincascade*, which generates the final *.xml* file containing the cascade description. It takes as arguments the binary file of positive samples, text file of negative samples and various other arguments that are well documented in the tutorial [3]. Since we want to optimize the classifier run (the classification part, because the training can be done beforehand on a much more powerful hardware), we need a pre-trained model. We generated a very simple model (using about 50 positive samples and about 400 negative samples), which is almost useless for the real-world detection. Although the small model is not really useful for a reliable object detection, it provides us an opportunity to compare a big model (the OpenCV provided one) trained on several thousand positive and tens of thousands negative samples to the small one trained on tens of positive and hundreds of negative samples. For the following experiments, we use both the our generated model and the OpenCV pre-trained model. We call them *"simple model"* and *"frontalface model"* (distributed along with the OpenCV codebase in *OPENCV_ROOT/data/haarcascades/haarcascade_frontalface_alt.xml*) and we will specify the used model for each experiment.

3.5 The testing program

For the profiling and optimization part of this thesis, we used a straightforward program, based on an OpenCV tutorial application, that loads images given a path to the folder and tries to detect the wanted object/objects (based on the pre-trained model) in each of them. In the documentation of the *objdetect* module [10], we can find that the method called *detectMultiScale* does all the job regarding the search of the wanted object. The method returns a vector of rectangles defined by x, y coordinates, *width* and *height* that contains the wanted object (the detected object, as the method name suggests, can be in a different scale than the original one). We do not post-process the rectangles, since the post-processing does not come under the classifier algorithm as is. We can see the testing program code below.

Histogram equalization (as described in the Section 2.5.2) is the only pre-processing we do with the input image before calling the *detectMultiScale*.

```
CascadeClassifier cascade;
String cascade_name = parser.get<String>("cascade");
String path_name = parser.get<String>("path");

// -- 1. Load the cascade
if (!cascade.load(cascade_name)) {
    cout << "--(!)Error loading the cascade\n";
    return -1;
};
unsigned int counter = 0;

// -- 2. Read the images
vector<Mat> images;
load_images(path_name, images, false);
Mat frame_gray;
std::vector<Rect> objects;
printf("Number of images: %d\n", images.size());

for (Mat image : images) {
    equalizeHist(frame_gray, frame_gray);
    // put all found rectangles of wanted object into 'objects'
    vector
    cascade.detectMultiScale(image, objects);
}
```

Figure 3.1: The very simple testing program used for the analysis and optimization of the Haar cascade classifier run

Chapter 4
Optimization of the OpenCV Haar cascade classifier

We analyze the possible optimization opportunities, analyze some approaches and finally describe the results of some performance-oriented tests and discuss the path to increase the performance of the used algorithm.

4.1 Analysis of possible optimization techniques

Multiple layers of optimization techniques exist, which we list here, but some of them are not realized in this project. We can divide the possible optimization into three categories. The first one is a compilation-time optimization (type and version of a compiler, parameters used to build a specific library - for example CMake parameters, etc.). The second is a code optimization (used data structures, optimizing loops and conditions, branch prediction, etc.). And the last one is a hardware-specific optimization (analysis of cache sizes + optimization of data fitting into a cache line, rewrite parts of the code, which is executed the most, or is the slowest to execute, in CPU-specific assembly). The third one is the most time-demanding and usually not necessary. The rule related to all the categories is that when starting with any optimizations, we should study the SoC running the code before doing anything more complex. That includes mainly if we can perform parallel computations, if we can use for example floating-point instructions and cache sizes. In this particular project, we run the code on a single-core CPU, which means that any parallelism can only make the performance worse due to context switching overhead.

When it comes to the compile-time optimization, larger projects usually have some documented configuration files or CMake parameters that affect the implementation generally. For example, for a multi-core system, a fully-parallelized method is called instead of the single-core version. Explicitly, the OpenCV has a vast amount of CMake parameters to tweak. If compiled manually, we can adjust these parameters to match the hardware as close as possible. Also, when dealing with a larger and older (more than 20 years old as for the OpenCV) library, there may be some configurations not even functional, so at least small research of what a particular option actually do is necessary.

Another step to increase performance is very much related to the first one mentioned and still falls into the compile-time optimization category, and it is the compiler used and the compiler options used. Dealing only with the project-specific configurations is not the end of the pre-compilation optimization, because, for example, the GCC compiler, can significantly optimize the generated code. For example, the SoC we use supports NEON instructions, which means that the GCC compiler, when set correctly, can optimize most loops in the program. We will look into this optimization technique in the Section 4.4 in more detail. Analyzing the OpenCV CMake parameters along with the GCC compiler options could be the first step to any other optimization.

The last optimization technique we consider is the actual code re-writing and code optimization. Before any code adjustments, we have to understand the code, the language the code is written in, how the hardware cache memory works [20], and also how the processor work. One way is to understand the code and try to re-write the algorithm with the specific hardware in mind. Another way is to analyze the code, improve the manipulation with the used data structures, maybe use different data structures. A good example is appending to a C++ *vector*. When appending to a vector inside a loop, the operating system has to reallocate the vector, because it grows in size, and the recurring reallocation as a process costs a tremendous amount of resources. Another great example could be the usage of wrong numeric data types. Usage of smaller (in terms of size in bits) data type in a method (or a loop) that is called thousands of times might make a difference in what data fits inside the cache. It might make a vast performance difference.

To sum it up, we list the possible optimizations in the following list:

- Optimization of compile-time parameters of the OpenCV
- Setting up the compiler and its options
- Code optimization
- Deeper hardware-specific optimizations (mostly re-writing parts of the code to assembly)

In this project, we will fully cover the pre-compilation optimization, analyze its impacts, and examine the code for other possible optimizations.

4.2 Hardware functionality testing and benchmarking

The previous analysis and configuration raised two major questions - the PMU support was disabled in the device tree, and we cannot be sure about the actual size of the hardware cache memory. Here we cover the testing and benchmarking part of the device. We did the cache size testing using a memory benchmarking program written in C language. Also, along with the

```
struct s array[MAX_CPUS][64 * 0x100000 / sizeof(struct s)]
    __attribute__((aligned(2 * 1024 * 1024)));

// preparation of the linked list
if (sequential) {
    for (i = 0; i < count - 1; i++)
        array[i].ptr = &array[i + 1];
    array[count - 1].ptr = &array[0];
} else {
    memset(array, 0, size);
    struct s *p = &array[0];
    for (i = 0; i < count - 1; i++) {
        p->ptr = (struct s *)1; /* Mark as occupied to avoid
    self-loop */
        for (j = rand() % count; array[j].ptr != NULL; j = (j >=
     count) ? 0 : j + 1)
            p = p->ptr = &array[j];
    }
    p->ptr = &array[0];
}
```

Figure 4.1: Part of the program used for memory benchmarking

benchmark program, we used the Linux perf tool to collect the data, compare it to the output of a standard laptop, and determine if the PMU in the device works correctly.

We used a benchmark program provided by the supervisor [30]. As we can see in the Figure 4.1, the program allocates a simple array of pre-defined size and connects the elements into the linked list (random or sequentially, depends on the input parameter). It goes through the linked list (read-only or read and write) a given number of times, once again set by the input parameter. The graph in the Figure 4.2 from several runs (read-only) using the random order. More specifically, we ran the benchmark ten times for each of the different array lengths, then calculated the mean value (as plotted) and the standard deviation (error bars) of the runs. On the left y-axis axis, we illustrate the number of CPU cycles taken by the benchmark program using the array size labeled on the x-axis. CPU cycles are log-scaled to demonstrate the increase better. Also, the CPU cycles number is the output of the perf command, so it does represent even the preparation of the benchmark, not only the values reading. On the right y-axis (not scaled to the left one), we can see a percentual ratio of cache misses to the total cache references separated by cache levels. It is undeniable that the size of the L1 cache really is $32\,kB$. The cache misses of level one cache rise slightly in the interval from $24\,kB$ to $32\,kB$, but the primary growth is in the interval from $32\,kB$ to $48\,kB$. Analogically, we can say that the level two cache is $128\,kB$, since the ratio of cache misses of L2 cache expands since that array size.

Another thing we need to test and demonstrate is the proper functionality of the PMU. We cannot be 100% sure that the values are very exact. However,

we can determine from the graph in Figure 4.2 that the values from multiple benchmark runs are valid and do not differ much from the benchmark runs on the laptop processor with a genuinely functional PMU.

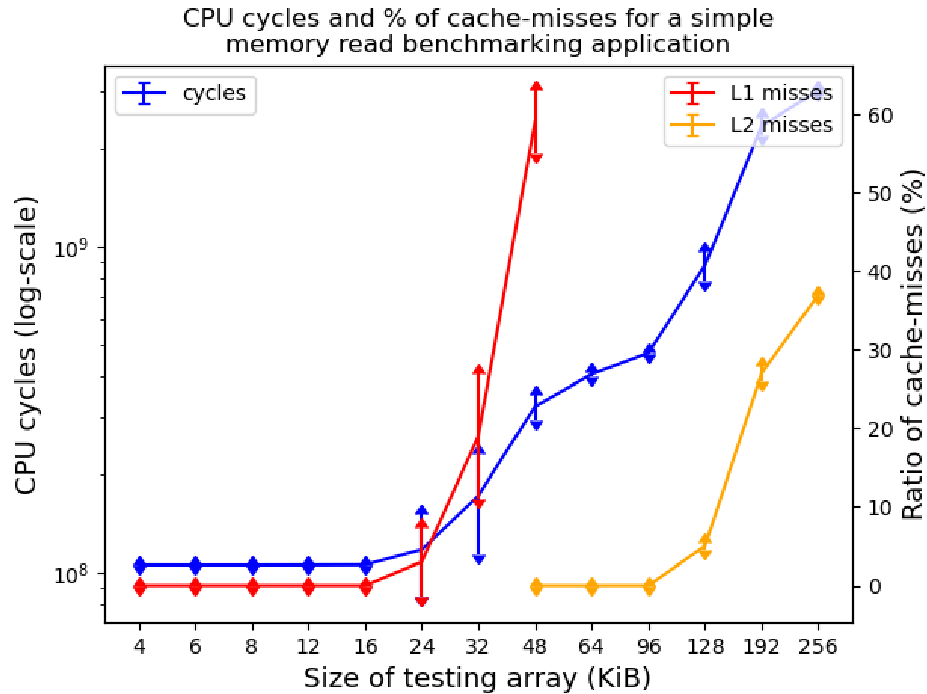

Figure 4.2: Testing the actual sizes of device caches

4.3 Function call-chain analysis of the detection phase run

Here we analyze the call-chain of the algorithm's classification phase and figure out where most of the cache-misses happen. We show the flame graphs of the two call-chains – on the laptop and on the i.MX6ULL itself and compare them.

In Figures 4.3a and 4.3b, we can see the comparison of *all* cache-misses (both L1 and L2) in the call-chain first recorded on the i.MX6ULL itself and on the laptop. We can say that the cache-misses happen in the same methods. The difference in the naming of the *integral* function is caused by the Intel's IPP optimization mentioned before. The most significant number of misses happen in the method called *integral*. That is the method, where the feature value is computed from the integral image. To achieve a near-to-zero cache-misses here, we would have to store the whole integral image in the hardware memory cache. Furthermore, we would have to store all the images, in which we want to detect the object, in cache, and that is, considering the size of the L1 and L2 cache, unattainable. These particular results as seen on the

graphs are measured using the cascade trained by ourselves against. About 80 images were used (with about 50% positive and 50% negative samples) of size 640×480 to test the performance of the algorithm. We do not show these graphs for the pre-trained cascade provided by OpenCV, which is much bigger than the ours, but the results were very similar and we get almost identical graphs for both the cascades. This simple flame-graph of cache-misses gives us information about where we should focus our optimization work. If we optimize the number of cache-misses in the *integral* method, we will make the whole algorithm run significantly faster.

(a) : Cache-misses happened in the classification algorithm run on the i.MX6ULL

(b) : Cache-misses happened in the classification algorithm run on a laptop

4.4 Optimization using compilation parameters

In this part of the work, we will study the possibilities of optimization at compile time, we will tweak the OpenCV compile-time configuration parameters and GCC command-line arguments to gain the best results possible on our specific hardware.

4.4.1 OpenCV compile-time configuration parameters

In this Section we describe the OpenCV compile-time parameters and we try to tweak them to gain the best performance possible. Since we compile the OpenCV manually, the first and easiest one is tweaking the build parameters [4]. OpenCV comes with a lengthy list of CMake options.

To build a library for a different platform using CMake, we need a so-called toolchain file, which is passed to the CMake itself and contains information regarding the compiler and other tools used. Since the OpenCV uses CMake as its build system and we need to cross-compile the whole library, we need that toolchain file. A possible way is to write one from scratch, but as said before, we since we use Buildroot [2, 28] to build the system, which automatically generates a toolchain file during the system build. We use this toolchain file and tweak it afterward if any changes are necessary. If one has a Buildroot system already built, the toolchain file is located in *BUILDROOT_ROOT/host/share/buildroot/toolchain.cmake*. It sets some necessary compilation parameters, such as C compiler or CXX compiler.

After the analysis of the *CMakeLists.txt* OpenCV file, we found that there could possibly be considerable differences in performance when the compilation parameters are correctly set.

First, turn off everything, which is not available on the embedded system. That is, turn off all parallelization (we use a single-core processor), graphical interfaces (GTK is enabled by default), and OpenCL support. OpenCV provides Python and Java support, which we also do not require. To sum that up, use *WITH_PTHREADS_PF=OFF*, *WITH_GTK=OFF*, *WITH_OPENCL=OFF*, *BUILD_JAVA=OFF*, *BUILD_opencv_python2=OFF* and *BUILD_opencv_python3=OFF*. None of these options is essential to increase the performance of the test code, but it speeds up the compilation of the OpenCV and excludes unneeded files from compiling.

Now, for the optimization itself. The most significant difference could make usage of NEON vector instructions. OpenCV supports enabling NEON instructions in its CMake configuration. Adding *ENABLE_NEON=ON* should do the trick (we will check the process later on). The processor in our embedded system supports even VFPv3 instruction set, which should be better optimized. The opposite is exact, and after a series of tests, enabling VFPv3 in OpenCV did not speed up the tested algorithm at all. Enabling the VFPv3 instruction set, we found that the tested program was even slower by about 6% than without enabling anything. We can see the comparison

of different options enabled in the Figure 4.4. For each label, a + character means that the particular option is enabled/set and a − character means the opposite. The *ENABLE_NEON* parameter does not do anything at all. At least for our purpose.

We should be aware that CMake tweaking is always a bit of a problem, because one cannot be sure it does what it says. Even though *ENABLE_NEON* was set correctly, there were no changes in the testing program run performance (as we can see in the Figure 4.4) and if it worker, there would have to be at least a small performance increase. The reason is that OpenCV does not use NEON data types at all. NEON support for OpenCV is still under development, and although there is a CMake parameter enabling it, it does not affect the program or libraries at all. There is almost a test program to check, whether the NEON is supported located in the OpenCV source codes (*OPENCV_ROOT/cmake/checks/cpu_neon.cpp*) that uses the NEON support, but support for the whole library is not yet implemented. We can verify that by searching for the *CV_NEON* phrase in the whole project. *CV_NEON* is the macro used for NEON support in the OpenCV project, but only a supporting file for an optimization library exists (*OPENCV_ROOT/modules/core/include/opencv2/core/neon_utils.hpp*), but the library is not used in any of the classes we use. We did one more confirmation on this finding – even in the dump of the generated assembly code, there is no usage of NEON instructions.

4.4.2 GCC compiler optimization parameters

In this Section, we analyze and test the performance of different compiler options. The default GCC optimization for CMake build type *RelWithDebInfo* is -*Os*, which optimizes the code only if the optimization does not increase the size of the generated code. Only by using the -*O2* optimization, we achieved performance better by about 16% compared with the -*Os* option. The differences in GCC optimization flags are not extensive. If we call *gcc -Q -Os (-O2) –help=optimize* [12], we can see all the flags and their state (enabled / disabled). After checking the diff of flags of these two optimizations, only six flags differ. That are -*falign-functions*, -*falign-jumps*, -*falign-labels*, -*falign-loops*, -*foptimize-strlen* and -*finline-functions*. In the table 4.1, we can see that the first five flags are disabled in -*Os* optimization and enabled in -*O2*. The last one is the opposite.

What we thought could make the most notable difference is -*falign-loops*, because of the *shifting window* algorithm used for many possible scales of the image. We recompiled the whole OpenCV with -*Os* optimization used, but with -*falign-loops* parameter enabled. Unfortunately, without any difference in performance against the plain -*Os* option. In the graph 4.4, the -*Os* optimization was enabled in the first four experiments (first four bars). Since the generated code of -*O2* optimalization is not marginally larger than using the -*Os* option, we will use the -*O2* optimization.

Last but not least, we try the profile guided optimization (PGO) [12] implemented directly by the GCC compiler. This approach is the most

GCC optimization flag	-Os optimization	-O2 optimization
-falign-functions	disabled	enabled
-falign-jumps	disabled	enabled
-falign-labels	disabled	enabled
-falign-loops	disabled	enabled
-foptimize-strlen	disabled	enabled
-finline-functions	enabled	disabled

Table 4.1: GCC optimization flags in different optimization level

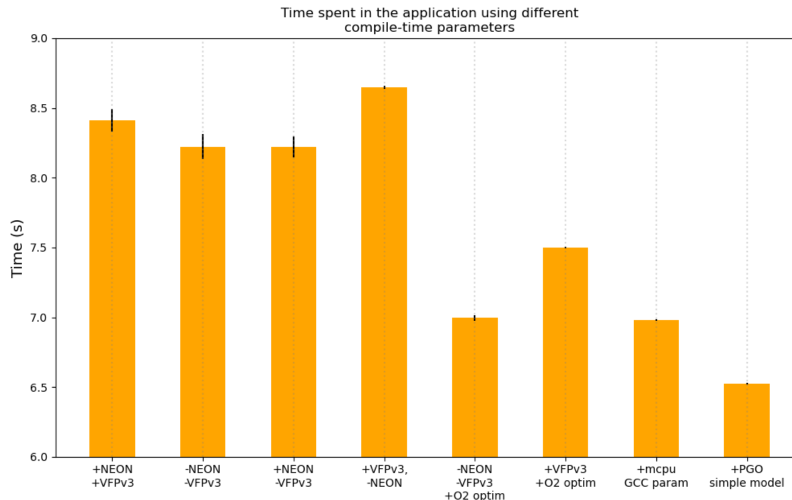

Figure 4.4: Optimization of performance using the CMake and GCC parameters – measured on 10 images of size 640 × 480 using the "simple model"

demanding of possible optimizations to be done by tweaking the GCC compiler options. Nevertheless, it makes a significant difference in improving the performance of an application that executes very similar code path in each run (e.g., detecting an object in an image – many subwindows not containing the wanted object means many runs in the similar code path that rejects the subwindow). To execute the profile guided optimization, we need to recompile the OpenCV library twice, each time with different compiler parameters. The first time, we have to use the *-fprofile-generate* parameter along with the *-fprofile-dir*. The *-fprofile-dir* only sets the path, where the optimization files (*Gcov data files* [1]) should be generated. One compiles the project using these parameters, run it once (or multiple times with different input data), and GCC generates those *Gcov data files*. We pass a path to these files to the *-fprofile-use* GCC parameter (in the following compilation), and the final compilated program should be a bit better in terms of performance.

What it actually does under the lid that during the training run, the compiler collects runtime data such as a number describing how many times a branch is taken and data about values of expressions used during the program

run. These data can then be used mainly for branch prediction optimization in the following project compilation. That is why two compilations of the whole project are needed.

We tried it once without any specification of the path (by default, it should be the same folder as the binary files). The relative path (the same folder as the binary files) do not work when cross-compiling. When we set the *-fprofile-dir* parameter to the absolute path (in the target system), the files were generated as they should. In the graph above, we can see the performance enhance after the PGO second compilation. The total performance enhancement is about 21% in the testing program performance (counted against default OpenCV compilation without manual changes and tweaks). We present the results from the most important compile-time options in the Figure 4.4. We ran the algorithm using each option 10 times – each bar in the graph shows the average run-time and the black lines on the top show the standard deviation for each option. The results were acquired using the "simple model" and 10 images of size 640×480.

4.4.3 Results for differently-sized models

We have mentioned that the model we trained was really small in comparison for example to the pre-trained one provided by OpenCV, so we measured the three most important compile-time parameters (taken from the graph 4.4) – default OpenCV options without both NEON and VFPv3 support, the *-O2* optimized without NEON and VFPv3 support and the generted code using profile-guided-optimization – again, now using the "frontalface model". When we change the used cascade, whole PGO generated code is not effective at all, because it causes the algorithm to have about 40% branch-misses. The conclusion from this finding is that we need to optimize the code using PGO for each different cascade separately. That is because each cascade has a different number of strong and weak classifiers, so the branch prediction trained on one cascade simply cannot be the same for much bigger cascade.

Using the "frontalface model" result in much lower performance. For the same number of images of the same size, only using the larger model causes about 4 times slower performance (we can see the difference in the Figure 4.5 vs Figure 4.4 – 8.25 vs 34.5 seconds for the default option without any optimization and without NEON and VFPv3 instructions). Using the *-O2* GCC optimization level increases the performance by about 9% (in comparison to about 6% using the "simple model"). What is remarkable is that even the PGO "trained" on the "frontalface model" actually made the algorithm slower when comparin to the library compiled without using PGO. We executed the experiment twice – we "trained" the PGO using the same images, which we then used for measuring the algorithm run-time and generating the graph 4.5 and for the second time, we "trained" the PGO on a different dataset. The results were both times the same – PGO makes the algorithm slower by about 3% when using a larger model. We can see the results in the Figure 4.5. The results are generated based on data from the average of 10 algorithm runs on images of size 640×480. Standard deviation

of the results are displayed as the black line in the top of each bar.

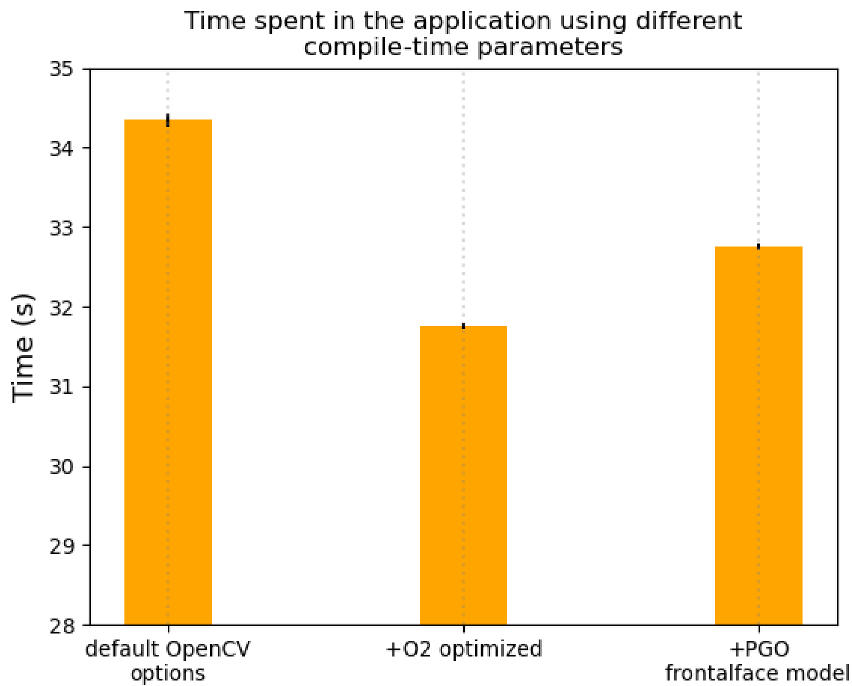

Figure 4.5: Optimization of performance using the CMake and GCC parameters – measured on 10 images of size 640×480 using the "frontalface model"

4.5 Code optimization

In this Section we analyze the algorithm code itself, find some places to optimize the algorithm performance and describe results of few more tests to gain even better performance.

4.5.1 Analysis of the algorithm code

We analyze the algorithm code and possible optimizations to implement. The algorithm itself is written so efficiently that we did not find almost anything we could re-write to gain a better performance. Because we ran the classifier on a single-core CPU, we removed all remains from parallelization – mutex locking and unlocking in the CascadeClassifierInvoker's *operator()* method. Unfortunately, this change is too small to have an impact on the algorithm performance and the average performance from 10 runs stays the same (the standard deviation was bigger than the performance difference, so we cannot determine any performance increase).

4.5.2 Optimization of main classifier's methods arguments

In the Section 2.5.2, we presented the declaration of the *detectMultiScale* method and in this Section, we analyze the changes that the method's arguments changes cause and try to optimize them to gain better performance.

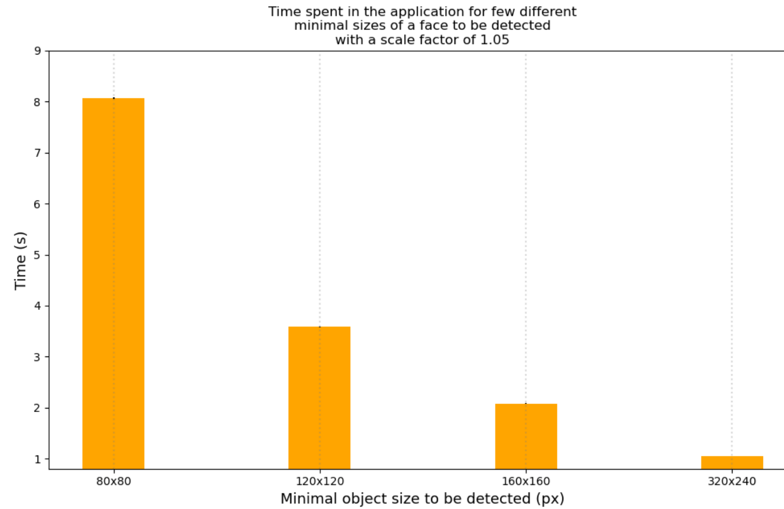

Figure 4.6: Algorithm performance for few different minimal object sizes (maximum object size set to 600×440 – measured on 10 images of size 640×480 using the "frontalface model"

Since the classifier is able to detect objects of any size (default minimal size is the minimum of sizes of the object as described in the text file of positive samples, which is used during the training phase), we could control the minimal and maximal size of the detected object (and ignore the rest) and thus increase the performance, because we significantly reduce the number of tested subwindows. We executed this particular optimization only on the "frontalface model", since that one uses a size of 20×20 as the minimal size of a detected human face. Size of images, where the object is to be detected, has to be taken into consideration. Because we run the classifier on images of size 640×480, we set the minimal size of detected human face to 120×120 (for the experimental purposes, we do not need to detect very small faces, but we want to demonstrate the performance difference). Also, we limited the maximum size of the detected face to 600×440. Since we increased the minimal object size, we should decrease the scale factor (to be able to detect faces of more different sizes in the allowed size interval). We changed the default scale factor (which is 1.1) to 1.05. All these values can be set when calling the *detectMultiScale* method, as shown in the Figure 2.11.

The performance difference is enormous and the time needed to run the algorithm on 10 images of size 640×480 drops from about 32 seconds to about 3 seconds, which means that the performance is about 10.5 times better. In the Figure 4.6, we present few different minimal sizes of the detected objects and their impact on the algorithm performance. All the results are the average

from 10 runs using the "frontalface model" on 10 images of size 640×480 with the standard deviation in time differences shown as the black error bars on top of each orange bar (on the Figure 4.6 are almost zero deviations, so the black dot really is the error bar).

Considering this experiment, we can say that if we have an a priori knowledge about the object size in the tested images, we can limit the minimal and maximal object size to significantly increase the classifier's performance.

Chapter 5 Conclusion

The goal of this thesis was to find, select, analyze and optimize an image recognition algorithm running on a low-performance embedded device i.MX6ULL.

The selected “Haar cascade classifier” performs pretty well and even on the low-performance hardware, it is able to provide precision object detection in a matter of milliseconds per image. We were able to increase the performance of the algorithm by about 10%. Using some a priori knowledge about the object size, we demonstrated a 9.5× faster algorithm run. Running the classifier on a single core CPU, we removed all parallelization from the implementation. Unfortunately, it did not led to any performance increase. We tested the algorithm with two different model sizes, which led to a conclusion that we need to limit the detected object size in order to have a sufficient performance when running on the slow processor. We also optimized the compile-time parameters of the OpenCV library and GCC options, which led to the mentioned 10% increase in performance.

We did not succeed in running the latest mainline Linux kernel, which could be a big drawback in future works using the i.MX6ULL applications processor. Using the manufacturer’s custom older Linux kernel, we ran several memory benchmarks to verify the actual hardware parameters and its capabilities.

We also described the Haar cascade classifier and its OpenCV implementation to the detail and generated our own classifier’s model, which could be a steppingstone for similar future works needing an efficient image recognition algorithm for a lower-performance processor.

Bibliography

[1] *Brief Description of gcov Data Files.* – URL https://gcc.gnu.org/onlinedocs/gcc/Gcov-Data-Files.html

[2] *The Buildroot user manual.* – URL https://buildroot.org/downloads/manual/manual.html

[3] *Cascade classifier training.* – URL https://docs.opencv.org/3.4.0/dc/d88/tutorial_traincascade.html

[4] *CPU optimizations build options.* – URL https://github.com/opencv/opencv/wiki/CPU-optimizations-build-options

[5] *Cross compilation of OpenCV for ARM based Linux system.* – URL https://docs.opencv.org/2.4/doc/tutorials/introduction/crosscompilation/arm_crosscompile_with_cmake.html

[6] : *Deprecation of the Haar Cascade applications in the OpenCV codebase.* – URL https://github.com/opencv/opencv/issues/13231

[7] *Device tree reference.* – URL https://elinux.org/Device_Tree_Reference

[8] *Histogram equalization.* – URL https://docs.opencv.org/3.4/d4/d1b/tutorial_histogram_equalization.html

[9] *i.MX6ULL Data Sheet.* – URL https://www.nxp.com/docs/en/data-sheet/IMX6ULLCEC.pdf

[10] *OpenCV Docs: Object Detection.* – URL https://docs.opencv.org/3.4.0/d5/d54/group__objdetect.html

[11] *OpenCV Documentation.* – URL https://docs.opencv.org/3.4.8/index.html

[12] *Options That Control Optimization.* – URL https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html

[13] *Perf Wiki.* – URL https://perf.wiki.kernel.org/index.php/Main_Page

[14] *Profiling OpenCV Application.* – URL https://github.com/opencv/opencv/wiki/Profiling-OpenCV-Applications

[15] *QEMU Documentation.* – URL https://www.qemu.org/documentation/

[16] AUTHORS, Multiple: *Pre-trained models for Haar cascade classifier – face / eyes / human body detection.* – URL http://alereimondo.no-ip.org/OpenCV/34

[17] BASTIAN, Théophile ; KELL, Stephen ; ZAPPA NARDELLI, Francesco: Reliable and fast DWARF-based stack unwinding. In: *Proceedings of the ACM on Programming Languages* (2019), S. 1–24

[18] BRADSKI, G. ; KAEHLER, A.: *Learning OpenCV: Computer Vision with the OpenCV Library.* O'Reilly Media, 2008. – ISBN 9780596554040

[19] CULJAK, I. ; ABRAM, D. ; PRIBANIC, T. ; DZAPO, H. ; CIFREK, M.: A brief introduction to OpenCV. In: *2012 Proceedings of the 35th International Convention MIPRO*, 2012, S. 1725–1730

[20] DREPPER, Ulrich: What every programmer should know about memory. In: *Red Hat, Inc* (2007)

[21] GUENNOUNI, S. ; AHAITOUF, A. ; MANSOURI, A.: Multiple object detection using OpenCV on an embedded platform. In: *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, 2014, S. 374–377

[22] HILLIARD, George: *Mastering embedded Linux.* – URL https://www.thirtythreeforty.net/posts/2020/01/mastering-embedded-linux-part-3-buildroot/

[23] KURUKULASOORIYA, Ayesha ; DHARMARATNE, Anuja: Image Searching with Eigenfaces and Facial Characteristics, 01 2011, S. 215–224

[24] LIENHART, Rainer ; MAYDT, Jochen: An extended set of haar-like features for rapid object detection. In: *Proceedings. international conference on image processing* IEEE (Veranst.), 2002

[25] MAGID, Salma A. ; PETRINI, Francesco ; DEZFOULI, Behnam: Image Classification on IoT Edge Devices: Profiling and Modeling. In: *arXiv preprint arXiv:1902.11119* (2019)

[26] MARTÍNEZ ZARZUELA, Mario ; DÍAZ-PERNAS, Francisco ; ANTÓN-RODRÍGUEZ, Miriam ; PEROZO, Freddy ; GONZÁLEZ-ORTEGA, David: AdaBoost Face Detection on the GPU Using Haar-Like Features, 05 2011, S. 333–342

[27] NUHI, Ali: *ARM NEON development.* – URL http://www.add.ece.ufl.edu/4924/docs/arm/ARM%20NEON%20Development.pdf

[28] Petazzoni, Thomas ; Electrons, Free: Buildroot: a nice, simple and efficient embedded Linux build system. In: *Embedded Linux System Conference*, 2012

[29] Sobral, Andrews: *Pre-trained model for Haar cascade classifier – car detection.* – URL https://github.com/andrewssobral/vehicle_detection_haarcascades/blob/master/cars.xml

[30] Sojka, Michal: *Memory latency benchmark.* – URL https://github.com/CTU-IIG/thermobench/blob/8c2607924b025ee34b52fbd2a567cbeff252933b/benchmarks/mem/membench.c

[31] Torvalds, Linus: *Linux kernel git repository.* – URL https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/

[32] Viola, Paul ; Jones, Michael u. a.: Rapid object detection using a boosted cascade of simple features. In: *CVPR* (2001)

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jandek Martin** Personal ID number: **475389**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Efficient Image Recognition on Low-Performance CPUs

Bachelor’s thesis title in Czech:

Efektivní algoritmy pro rozpoznávání obrazu na levných embedded procesorech

Guidelines:

Consumer electronics must be cheap, but users expect more and more functionality (or intelligence) even from cheap devices. The goal of this project is to try to combine those contradictory requirements and optimize several image recognition/classification algorithms for the given low-cost hardware. The focus is not on developing brand new algorithms, but rather on optimization and efficient use of computing resources provided by the hardware platform – a MIPS/ARM-based CPU running Linux.

1. Make yourself familiar with popular image detection/classification algorithms, OpenCV library and software performance evaluation techniques.
2. Implement a simple program using the OpenCV library for object (e.g. food) detection. Consider using algorithms such as Haar cascade or SVM classifier. Use freely available datasets for training the classifier.
3. Evaluate performance of the implemented algorithms (detection phase only) and find performance bottlenecks. Carry out the evaluation on both common laptop CPU as well as low-performance embedded CPU (ARM- or MIPS-based).
4. Propose changes to the detection algorithms to remove or reduce the performance bottlenecks and evaluate the performance gain.
5. Document the results and discuss trade-offs between quality of object detection and the resulting performance.

Bibliography / sources:

[1] https://perf.wiki.kernel.org/index.php/Main_Page
[2] B. Greeg, Linux performance, http://www.brendangregg.com/linuxperf.html
[3] Viola, Paul & Jones, Michael. (2001). Rapid Object Detection using a Boosted Cascade of Simple Features. IEEE Conf Comput Vis Pattern Recognit. 1. I-511. 10.1109/CVPR.2001.990517

Name and workplace of bachelor’s thesis supervisor:

Ing. Michal Sojka, Ph.D., Embedded Systems, CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.01.2020** Deadline for bachelor thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Ing. Michal Sojka, Ph.D. Supervisor’s signature	doc. Ing. Tomáš Svoboda, Ph.D. Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

 © VUT v Praze, Design: VUT v Praze, VIC

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student’s signature

© VUT v Praze, Design: VUT v Praze, VIC