BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING
FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS
ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

QR CODE DETECTION UNDER ROS IMPLEMENTED ON THE GPU
DETEKCE QR KÓDŮ NA GRAFICKÉ KARTĚ PRO PLATFORMU ROS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Milan Hurban**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Jiří Krejsa, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF MECHANICAL ENGINEERING

Master's Thesis Assignment

Institut:	Institute of Solid Mechanics, Mechatronics and Biomechanics
Student:	**Bc. Milan Hurban**
Degree programm:	Applied Sciences in Engineering
Branch:	Mechatronics
Supervisor:	**doc. Ing. Jiří Krejsa, Ph.D.**
Academic year:	2016/17

As provided for by the Act No. 111/98 Coll. on higher education institutions and the BUT Study and Examination Regulations, the director of the Institute hereby assigns the following topic of Master's Thesis:

QR code detection under ROS implemented on the GPU

Brief description:

The thesis goal is to implement a method of image processing capable of QR code detection in an image acquired by an onboard camera of a mobile robot. To reduce the work load on the onboard computer, the method is implemented on the graphical card (GPU). Additionally, the method should be integrated into the Robotic Operating System (ROS) platform.

Master's Thesis goals:

1. Research the algorithms of QR code detection in an image
2. Research the methods for the implementation of computational tasks on the GPU
3. Select an appropriate algorithm, implement it and integrate under the ROS platform
4. Verify the functionality of your implementation

Recommended bibliography:

Quigley, M.: Programming Robots with ROS: A Practical Introduction to the Robot Operating System, O'Reilly Media, 2015

Koubaa, A.: Robot Operating System (ROS): The Complete Reference, Springer, 2016

Bakowski, P.: A Practical Introduction to Parallel Programming on multi-core and many-core Embedded Systems, Amazon, 2016

Students are required to submit the thesis within the deadlines stated in the schedule of the academic year 2016/17.

In Brno, 31. 10. 2016

L. S.

prof. Ing. Jindřich Petruška, CSc.
Director of the Institute

doc. Ing. Jaroslav Katolický, Ph.D.
FME dean

Abstrakt

Tato diplomová práce se zabývá vývojem a implementací algoritmu pro detekci QR kódů s integrací do platformy ROS a výpočty běžícími na grafické kartě. Z rešerše současně dostupných nástrojů a technik je vybrán vhodný postup a algoritmus je napsán jako modul v programovacím jazyce Python, který je snadno integrovatelný do ROS. Ke zprostředkování výpočtů na vícejádrovém hardware, jako jsou grafické karty či vícejádrové procesory, je využita knihovna OpenCL.

Klíčová slova

QR kód, ROS, GPU, digitální zpracování obrazu, paralelní výpočty

Abstract

This master's thesis deals with the design and implementation of a QR code detection algorithm under the ROS platform with computations running on a graphical processing unit. Through a comparative survey of available tools and techniques, a suitable approach is chosen and the algorithm is written as a module in the Python programming language, ready to be implemented under the ROS platform. The OpenCL parallel computing platform is used to facilitate parallel computation on multi-core hardware, such as graphical processing units or multi-core CPUs.

Keywords

QR code, ROS, GPU, digital image processing, parallel computation

Bibliographic citation

HURBAN, M. QR code detection under ROS implemented on the GPU. Brno:Brno University of Technology, Faculty of Mechanical Engineering, 2017. 58 pp. Supervisor doc. Ing. Jiří Krejsa, Ph.D..

Thanks

I would like to hereby express thanks to my master's thesis supervisor, doc. Ing. Jiří Krejsa, Ph.D., for numerous comments and valuable suggestions on improving my thesis.

Honorable declaration

I declare that I have written the master's thesis *QR code detection under ROS implemented on the GPU* by myself and with the advice of my master's thesis supervisor, doc. Ing. Jiří Krejsa, Ph.D., using the sources listed in references.

In Brno, May 25th, 2017 .....

Milan Hurban

Contents

1 Introduction

In mobile robotics, resource management is essential. One of the limiting factors for the usefulness of a mobile robot is often battery life, and optimizing power usage is thus a good way of extending the robot's range and capabilities. Another issue is the scarcity of computational resources. The controlling software of a mobile robot has many tasks to perform - data acquisition and processing (e.g. localization and mapping), trajectory planning, decision making, the actual control of the robot's movement, etc. What makes the matter worse is that, in typical cases, the robot is required to perform these tasks in real-time, making any unnecessary tasks running on the CPU a burden for the entire system and any bottlenecks may become potentially hazardous to the robot's operation. It is thus advantageous to look at the nature of certain tasks and assess the feasibility of using other available hardware for their execution. This master's thesis takes on precisely this latter need for resource management - that is, freeing up precious CPU resources by utilizing for example an integrated GPU as a general-purpose parallel computing platform.

As a model example for this thesis, the detection of QR codes in images acquired by the camera of a mobile robot is considered. Quick response, or QR, codes are machine-readable binary matrices. In recent years, their use has spread from its original automotive part identification application to many other fields, including general tracking and data storage. Crucially, they have transcended the industrial sphere to become included in our daily lives, whether it be shopping, cinema tickets or public transportation. Their compactness and, thus far, also novelty, have also meant that QR codes have caught on in marketing. Often, such codes will contain an address for a webpage or a simple tagline promoting a new product, movie, etc. This has also meant that the ability to read these codes has become expected of modern smartphones and other consumer electronics. If a mobile robot's expected field of operation includes interaction with humans or simply reading encoded data for its own use, it is thus advantageous to equip it with this ability.

The other reason this was used as a model example is that the detection of QR codes, and image processing in general, is a task that is ripe for parallelization. Often, we perform computations on individual pixels, or sometimes rows and columns of a digital image. If these computations are running reasonably independent of each other - that is, they do not require other computations to be completed to provide their own result - a graphical processing unit, or GPU, with its nowadays many hundreds of

computing units, can be utilized to perform the desired operations. This often allows us to avoid using costly loops as we would have to with serial execution. Not only will this usually be many times faster, but crucially, it will allow the CPU to focus on performing other tasks, which may perhaps not be subject to advantageous parallelization.

As one of the tasks of this thesis, the integration into the Robotic Operating System (ROS) platform is considered. This platform is commonly used in mobile robotics and using this already available framework will help preserve the homogeneity of the entire system.

2 Survey

In this chapter, the structure of QR codes is briefly explained. A survey is then conducted to summarize the currently used approaches for QR code detection and decoding, as well as the possibilities in parallel computing on a PC. The Robotic Operating System (ROS) is also introduced.

2.1 QR code structure

The QR Code's name comes from "quick response", which implies the circumstances of its creation. While one-dimensional barcodes were used for decades, in the 1990s the need arose for an easily readable code that could store more information. This led to the development of various two-dimensional codes. One of these is the QR code, developed in Japan. Its main advantage over its contemporary competitors was the fact that its creators put a lot of emphasis on the speed and accuracy of detection and reading. In order to achieve this goal, they set out to develop a special marker pattern that would be easily detectable and wouldn't lend itself to false positives in detection.

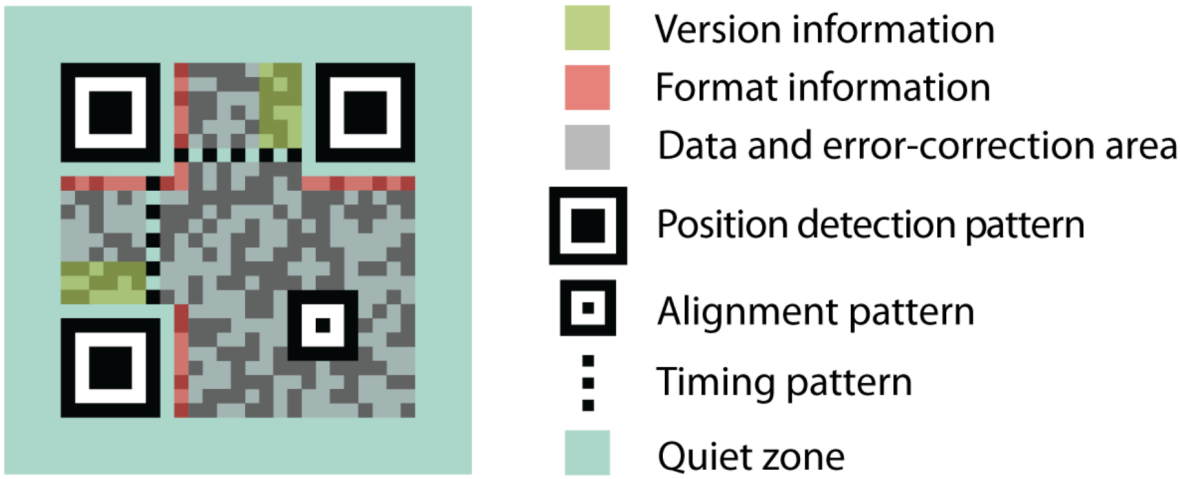

Figure 2.1: Structure of a QR code [22]

"After mulling over this problem thoroughly, they decided to do an exhaustive survey of the ratio of white to black areas in pictures and symbols printed on fliers, magazines,

cardboard boxes and so on after reducing them to patterns with black and white areas. They continued the task of surveying innumerable examples of printed matter all day long for days on end. Eventually, they came up with the least used ratio of black and white areas on printed matter. This ratio was 1:1:3:1:1. This was how the widths of the black and white areas in the position detection patterns were decided upon. In this way, a contrivance was created through which the orientation of their code could be determined regardless of the angle of scanning, which could be any angle out of 360 deg, by searching for this unique ratio." [21] The resulting markers (as seen in 2.1) are placed in three corners of the QR code, which means that detecting them will give us information about both the position and the orientation of the code.

Besides the position detection patterns, each QR code contains several other features to facilitate easier detection and decoding. In the original QR code, one additional alignment pattern is present. In larger versions of the code, multiple of these are included and their detection can help rectify distortions of nonlinear nature, such as the cylindrical distortion. Areas of the code are reserved for version information, format information and a data and error-correction area. A timing pattern of alternating black and white modules is present to help with the detection of module sizing. The area around a QR code has a so-called "quiet zone" that allows for easier detection.

In addition, QR codes come in several variants that differ by their robustness in terms of being able to be read even with scanning errors. In its most robust version, 30% of the code can be corrupted and the information stored within will still be recovered. This is achieved by the use of the Reed–Solomon error correction algorithm. Of course, the redundancy introduced into the code means that less data can be stored overall. Increasingly, this ability to be read even with errors leads to QR code creators infusing artistic elements into their codes for novelty, as in 2.2.

Figure 2.2: QR code with artistic elements, but still readable [25]

2.2 QR code detection

Although the QR code standard includes its own suggested algorithm for detection, there are areas in which it is found to be lacking. Other approaches are thus also suggested in several works, and they will be shortly reviewed in this chapter.

2.2.1 QR code specification algorithm

In the QR Code barcode symbology specification [20], an algorithm for detection is proposed:

- Determine a Global Threshold by taking a reflectance value midway between the maximum reflectance and minimum reflectance in the image. Convert the image to a set of dark and light pixels using the Global Threshold.
- Locate the finder pattern. The finder pattern in QR Code 2015 consists of three identical finder patterns located at three of the four corners of the symbol. The finder pattern in Micro QR Code is a single finder pattern. As described in 5.3.2, module widths in each finder pattern form a dark-light-dark-light-dark sequence the relative widths of each element of which are in the ratios $1 : 1 : 3 : 1 : 1$. For the purposes of this algorithm the tolerance for each of these widths is 0,5 (i.e. a range of 0,5 to 1,5 for the single module box and 2,5 to 3,5 for the three module square box).

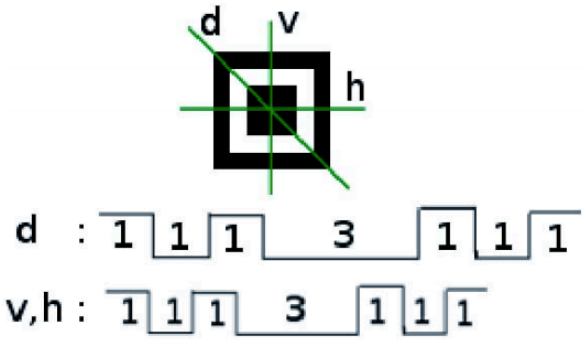

Figure 2.3: The ratio in a QR code corner marker [2]

1. When a candidate area is detected note the position of the first and last points A and B respectively at which a line of pixels in the image encounters the outer edges of the finder pattern (see Figure 31). Repeat this for adjacent pixel lines in the image until all lines crossing the central box of the finder pattern in the x axis of the image have been identified.

2. Repeat step 1) for pixel columns crossing the central box of the finder pattern in the y axis of the image.

3. Locate the center of the pattern. Construct a line through the midpoints between the points A and B on the outermost pixel lines crossing the central box of the finder pattern in the x axis. Construct a similar line through points A and B on the outermost pixel columns crossing the central box in the y axis. The center of the pattern is located at the intersection of these two lines.

4. Repeat steps 1) to 3) to locate the centers of the two other finder patterns.

5. If no candidate areas are detected, reverse the colouring of the light and dark pixels and recommence at the beginning of step b to attempt to decode the symbol as a symbol with reflectance reversal.

6. If a single pattern is identified but two further finder patterns cannot be located, attempt to decode the symbol as a Micro QR Code symbol by jumping to the Micro QR Code symbols reference decode (from step m).

While this algorithm serves as the basis for most detection algorithms in the encountered projects (e.g. [5], [1]), on its own, it is unsuitable for real world application for the detection of QR codes in arbitrarily acquired images, for example from the camera of a mobile robot. The first problem shows up at the very start, where a global thresholding method is advocated for. As we will see in later chapters, this is not sufficient for images acquired under varying light conditions. Furthermore, the basic algorithm doesn't account for any distortions at all, and as such will fail in detecting and decoding codes viewed at an angle or on uneven surfaces. This is presumably because of the assumption that QR codes will always be read by a dedicated reader under controlled circumstances, which is a luxury we cannot guarantee in mobile robotics. It is thus assumed that modifications will have to be made if this algorithm is used.

2.2.2 Viola-Jones framework

In [2], the authors took on the task of detecting QR codes in arbitrarily acquired images by making use of the Viola-Jones rapid object detection framework. In this framework, Haar-like features are used by classifiers and trained on a set of training data to select for reliable detection of desired features in a digital image. The prototypes of these are shown in 2.4. From a set of feature prototypes, the ones most useful for detecting QR code corner markers are selected by means of cascading and boosting, meaning that well-performing ones are reinforced, while features that are not useful for detection of the desired structure are suppressed.

Since the QR code corner finder patterns are a rigid structure present in every QR code, they are a good target for this approach. The fast calculations of the feature

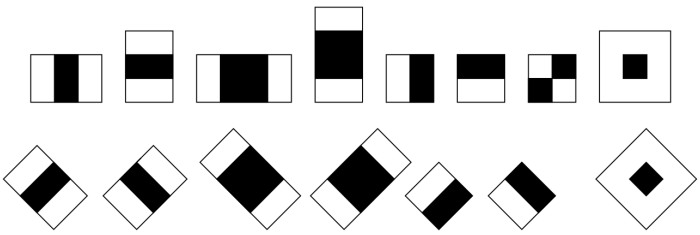

Figure 2.4: Feature prototypes [2]

values are obtained by using the integral image, that is, an image in which each (x,y) coordinate has a pixel value corresponding to the sum of all pixels values in the rectangle (0,0) to (x,y) in the original image. This allows the feature values to be calculated in constant time. The set of feature prototypes is trained on 285 images containing the QR code corner markers and achieved an accuracy of detection of about 90%. The authors advocated for and also implemented the use of post-processing to further improve QR code detection rates by using constraints based on the presumed location and size comparison of the corner markers in a single QR code. When tested on video frames of the resolution of 640x480 pixels, the best results were obtained for training samples of size 20x20. The detection then took about 50 ms, while the post-processing is largely sample size independent and takes around 120 ms.

2.2.3 Harris corner detector and Convex hull algorithm

Another approach for the detection and rectification of QR codes is proposed in [3]. Here, the authors use the Harris corner detector scheme to detect points within the QR code. They argue that since images in the real world are often distorted by perspective, the proposed standard algorithm of QR code corner marker detection cannot be used. Instead, once points within the QR code are detected, a convex hull algorithm is used to find a four-sided polygon that contains them, i.e. the boundary lines of the QR code. Once found, the four corner points are used for the correction of the image by perspective collineation. Authors report good results, but acknowledge the limitations of using this approach for heavily distorted images, such as ones with QR codes printed on cylindrical surfaces.

2.2.4 SURF

In work done at the Air Force Institute of Technology in Ohio, USA [14], the SURF feature detection algorithm is used on QR codes. The use case here is, however, different from what we are interested in. The purpose of using SURF is not to detect a QR code in the image in order to read the information stored within, but to reliably detect an already known QR code and estimate its pose. This would then be used for example

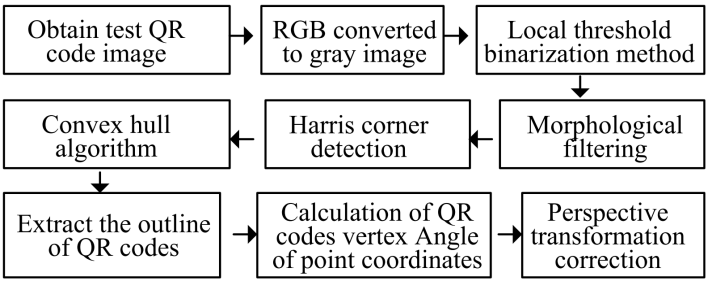

Figure 2.5: Suggested approach from [3]

for the fast and accurate estimation of the location of an aerial landing platform that an autonomous unmanned aerial vehicle (UAV) could land on, mid-air, as shown in 2.6.

Figure 2.6: Airborne landing platform, the use case in [14]

While it is not the application we are looking for, we can still gain valuable insight from it. The reason for using QR codes in this work is that, as was already shown in the case of the Viola-Jones framework, their binary black-and-white design lends itself well to feature detection. Indeed, "the descriptor stage of the SURF algorithm also uses the Haar wavelet response to describe feature orientation which consists of finding transitions from white to black or vice-versa. The result is a feature set that allows for large changes in rotation and translation between the QR code model image and a QR code in a scene. Furthermore, because of the pyramidal calculation of features by SURF, there are larger features detected in the QR code that allow the overall orientation of the code to be characterized. Overall, the traits of the QR code make it an excellent medium for determining the pose of an object it is mounted on." [14]

The work was focused on implementing the SURF algorithm on field-programmable gate arrays (FPGA), which is also hardware capable of parallel computing. This shows

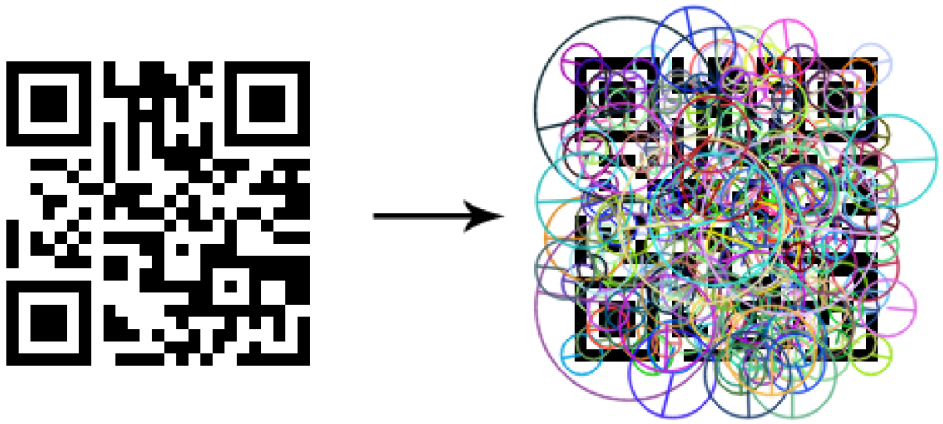

Figure 2.7: Detected SURF features in a QR code [14]

promise in the possibilities of optimization that would be beneficial in our case. If we are then able to quickly and reliably find SURF features, and many of them (as seen in 2.7) in each QR code at that, some clever postprocessing on our part would allow us to detect and decode the QR codes. For example, a suitable clustering algorithm could provide us with a good estimate of the position of QR codes in the image. By extracting these smaller areas, we wouldn't have to process the entire image and could speed up the whole process considerably. Quite worryingly for us, the FPGA-implemented SURF algorithm performed with framerate on the limits of real-time video use, even though only with a resolution of 640x480.

2.2.5 Contour detection

QR codes have recently been tested for use in surgical navigation systems [6], with the authors using a detection scheme based on finding the contours of QR code corner markers. A high resolution camera integrated in the surgical light acquires images of QR codes attached to a pointer instrument and the patient. The transformation between the QR codes and the camera is computed by detecting the marker position and orientation in the images and reading the QR size that is encoded in the QR codes.

When detecting a QR code corner finder pattern, three square contours can be identified, as can be seen in 2.9. To insure that three detected contours belong to a single marker, three geometrical features are considered. The first is hierarchy, meaning that contour i encloses contour j and k. The second feature is concentricity, that is, all three contours are located in the same position, or, in other words, have the same center of mass. Lastly, the proportionality of the contour perimeters is considered, so that they correspond to the expected ratios.

Once markers have been reliably found, groups of threes are taken and tested for being part of the same QR code. The authors approach this as a classification prob-

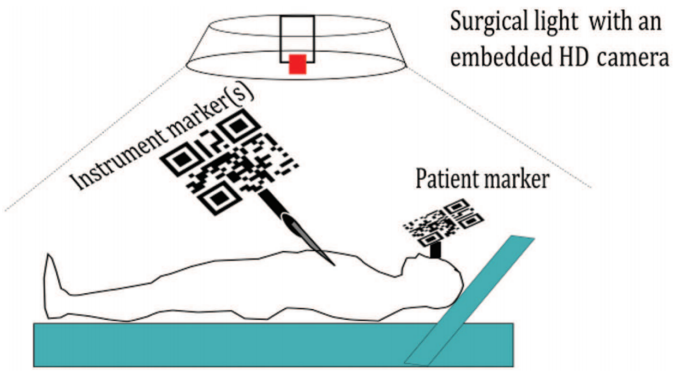

Figure 2.8: The detection setup used by [6]

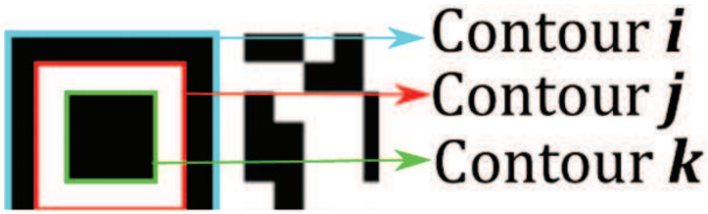

Figure 2.9: Found contours [6]

lem and again use three distinct features to classify the markers: perimeter, area and distance to the origin. The product of these features is a very robust measure.

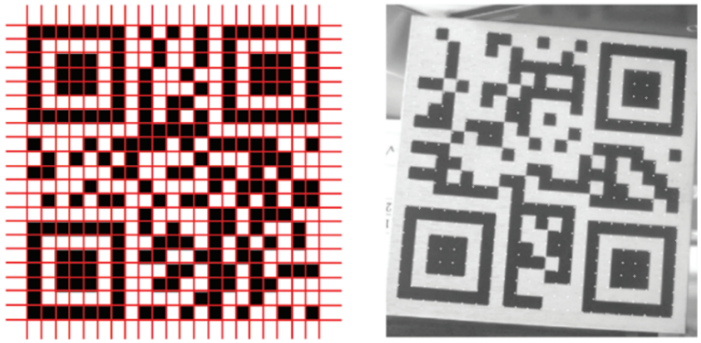

Figure 2.10: QR code segmented by fitted lines [6]

With the QR codes found, the size information stored within is read, which allows the authors to segment the code by fitted lines and read the rest of the data stored in the QR code, as seen in 2.10. The authors claim robust results even for distorted, rotated, and tilted QR codes, but admit that the detection precision is not yet up to the quality required for surgical navigation.

2.3 Parallel computing platforms

Parallelism in computing is increasingly used as a way to boost performance. Since there are technical challenges present with higher clock speeds, Central Processing Units (CPUs) now improve performance by adding multiple cores. The role of the Graphical Processing Unit has gradually evolved from being used purely to calculate graphical computations for rendering to general programmable parallel processors. With the desire to use the parallel computing capabilities of these platforms on the rise, various software is being developed to facilitate this access. [15] Of all the available frameworks, two have been chosen for closer investigation due to their popularity; Nvidia CUDA and OpenCL. [11]

2.3.1 Nvidia CUDA

CUDA® is a parallel computing platform and programming model invented by NVIDIA. It is used in many different fields like software development, scientific research or the video game industry. A testament to its popularity is the fact that several educational institutions offer whole courses on its use, such as Harvard University, University of Illinois at Urbana-Champaign or the Chinese Academy of Sciences.

The improvement in performance through parallel processing is an attractive concept, but until the last decade, general computation on graphical processing units wasn't easy to achieve, for example, assembly language was often needed to access these capabilities. The Nvidia company, with the CUDA parallel computing platform, provides a way to send C, C++ and Fortran code straight to GPU, no assembly language required.

"The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the host refers to the CPU and its memory, while the device refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches kernels which are functions executed on the device. These kernels are executed by many GPU threads in parallel.

Given the heterogeneous nature of the CUDA programming model, a typical sequence of operations for a CUDA C program is:

- Declare and allocate host and device memory.
- Initialize host data.
- Transfer data from the host to the device.
- Execute one or more kernels.
- Transfer results from the device to the host.

One advantage of the heterogeneous CUDA programming model is that porting an existing code from C to CUDA C can be done incrementally, one kernel at a time." [29]

An example kernel written for CUDA that adds together two arrays would look like this:

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

Unfortunately, the usefulness of CUDA is somewhat limited by it only being supported by hardware made by Nvidia.

2.3.2 OpenCL

"OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. OpenCL supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, close-to-the-metal programming interface, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications. OpenCL is particularly suited to play an increasingly significant role in emerging interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines.

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors; and a cross-platform programming language with a well specified computation environment." [15]

The main idea behind OpenCL is to provide a single framework that could use the capabilities of heterogeneous hardware, such a CPUs, GPUs and FPGAs. While nowadays all of these have multiple cores, they don't work exactly the same way and often differ in the way they handle memory. While the standards for CPUs usually assume a shared address space and do not inherently support vector operations, general purpose GPU programming models address complex memory hierarchies and vector operations. They are, however, traditionally platform-, vendor- or hardware-specific. These limitations make it difficult for a developer to access the compute power of heterogeneous CPUs, GPUs and other types of processors from a single, multi-platform source code base. Not only can OpenCL kernels run on different types of devices, but a single application can dispatch kernels to multiple devices at once. For example, if a

computer contains both a processor and graphics card, kernels can be synchronized to run on both devices and share data between them.

OpenCL works with five distinct data structures: device, kernel, program, command queue and context. A device is exactly that - a single piece of hardware on which kernels are run. The kernels are the computations themselves, blocks of code that are executed on target hardware. A program contains multiple kernels that are then distributed to devices. The command queue in OpenCL is the medium through which devices receive their kernels. The queue consists of commands sent to a device and one of these commands is used to make a device execute a kernel. And last, a context is a platform with a set of available devices for that platform.

Besides kernels, other important terms are the work-group and the work-item. Kernels are executed by one or more work-items. Work-items are collected into work-groups and each work-group executes on a compute unit. The kernels of OpenCL are written in C, as is shown in the following example of a simple sum of two arrays:

```
__kernel void vectorAdd(__global const float * a,
  __global const float * b,
 __global float * c)
{
 int nIndex = get_global_id(0);
 c[nIndex] = a[nIndex] + b[nIndex];
}
```

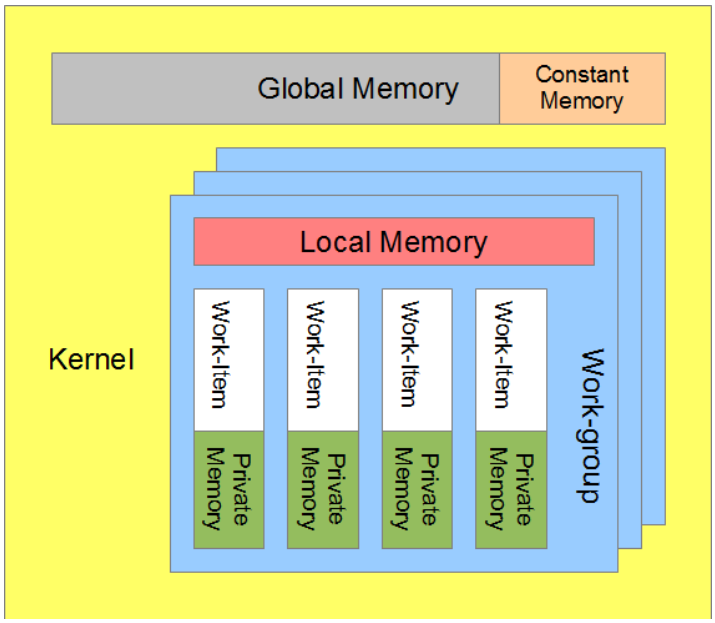

Figure 2.11: The hierarchy of memory used in OpenCL [28]

OpenCL works with multiple different kinds of memory: global, constant, local and

private. The global memory stores data for the entire device. Constant memory is very similar to that, but is read-only. Local memory stores data for the work-items in a work group and private memory stores data for an individual work-item.

Global memory can be read from and written to by both the host and the device. This memory is usually the largest on an OpenCL device, but it's also the slowest for work-items to access. A block of local memory is only accessible by work-items in one work-group, but they can do so much faster than with global memory. It is useful for storing intermediate results of the computations from individual work-items. Private memory is accessible and used purely by a single work-item for its own computations. This address space is the fastest to access, but it is also much smaller and cannot thus be used for storing vast amounts of data.

2.4 ROS

"The Robot Operating System (ROS) is a framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms." [9] Envisioned as a common platform for code- and idea-sharing, it allows for significantly decreased development time in new robotics projects. Among its chief aims is modularity, that is, the resulting software is structured as a large number of small programs that rapidly pass messages to one another. As such, it encourages reuse of already written programs, eliminating the need to reinvent the wheel with every new robot. [10]

"ROS consists of a number of parts:

1. A set of drivers that let users read data from sensors and send commands to motors and other actuators, in an abstracted, well-defined format. A wide variety of popular hardware is supported, including a growing number of commercially available robot systems.

2. A large and growing collection of fundamental robotics algorithms that allow users to build maps of the world, navigate around it, represent and interpret sensor data, plan motions, manipulate objects, and do a lot of other stuff. ROS has become very popular in the robotics research community, and a lot of cutting edge algorithms are now available in ROS.

3. All of the computational infrastructure that allows users to move data around, to connect the various components of a complex robot system, and to incorporate other algorithms. ROS is inherently distributed and allows users to split the workload across multiple computers seamlessly.

4. A large set of tools that make it easy to visualize the state of the robot and the algorithms, debug faulty behaviors, and record sensor data. Debugging robot

software is notoriously difficult, and this rich set of tools is one of the things that make ROS as powerful as it is.

5. Finally, the larger ROS ecosystem includes an extensive set of resources, such as a wiki that documents many of the aspects of the framework, a question-and answer site where users can ask for help and share what they've learned, and a thriving community of users and developers.

ROS provides all the parts of a robot software system that would otherwise have to be written. It allows the user to focus on the parts of the system that he cares about, without worrying about the other parts." [9]

From the beginning, ROS has been developed at multiple institutions and for multiple robotics projects. Nowadays, it is increasingly popular in education, scientific research, and the commercial sphere. It shares most of its philosophy with the Unix project, emphasizing peer-to-peer communication between the individual parts of the system, of which there are many, usually small in size and scope. It supports multiple programming languages, the most used of these being C++ and Python. The creation of standalone libraries is encouraged and their connection to the other ROS modules is relatively straightforward. The core of ROS is released under the BSD license, which allows commercial and noncommercial use.

3 Method

In this chapter, the chosen methods for the task of QR code detection implemented on the GPU will be introduced, along with an explanation for why they were chosen.

3.1 Suitable detection algorithm

When looking at the detection algorithms, the choice was made to adapt the basic algorithm outlined in the QR code bar cide symbology specification [20]. This is because it was specifically designed to take advantage of the structure of a QR code. By scanning each line and each row of an image, we can detect all QR codes contained in it in one go. These can then be extracted and dealt with one-by-one. The fact that the detection algorithm scans each line and column independently of the others means that parallelization of this task could be highly advantageous in terms of performance.

As noted in the survey part of this thesis, some modifications will have to be made. In the binarization step, a global threshold will give unsatisfactory results under uneven lighting conditions. Based on the works processed in the survey, a local thresholding method seems like a suitable choice in terms of both accuracy and speed. To this end, a uniform filter will be used. In order to improve performance, the optimized OpenCV library will be used to this end, since it is also able to take advantage of parallel computing on the GPU.

Another issue which is not covered by the basic algorithm is that codes are assumed to be undistorted. This means that QR codes viewed under a perspective transformation will have to be further processed to find at least four points with known initial positions in a square, undistorted QR code. These four points are needed to compute the homography transformation matrix, and the further apart they are, the more accurate the rectification will be. As such, the best candidates for these four points are the four corners of a QR code. Three of them are relatively easy to find, thanks to the presence of position detection patterns. The fourth one will have to be found by different means, further elaborated upon in later chapters.

3.2 Parallel computing platform

Of the available parallel computing frameworks, two of them (Nvidia CUDA and OpenCL) were shortly introduced in the previous chapter as the two main technologies used nowadays. Of these two, the OpenCL framework was chosen as a good fit for our requirements. While the CUDA technology is certainly powerful and has substantial support from the developers, its limitation to the hardware to that from one vendor is a disadvantage. On the other hand, OpenCL can perform computations on various heterogeneous computational platforms, from CPUs to GPUs and FPGAs. Its open-source license is certainly an advantage as well.

Another reason for this choice is the availability of direct personal experience with this platform in the vicinity of the author of this thesis. Valuable input was gained from work done by Petr Schreiber in his diploma thesis [13] on parallel computation, in which the OpenCL framework was also used. This helped avoid tricky bug-fixing stemming from unfamiliarity with the platform. The thesis outlines all the intricacies of OpenCL use, explains the structure of the various kinds of memory used by the framework and shows on basic examples how to write an OpenCL kernel and prepare it for its execution.

The use of OpenCL is also advantageous for the fact that several libraries are available in various programming languages that allow the user access to the OpenCL API directly from these languages, without the need for complicated interfacing.

3.3 Programming language

Because of the author's familiarity with the language and the ease of development it provides, the Matlab programming language was used to write a working-as-intended first version of the program. The readily available tools for plotting and visualization of intermediate results, as well as profiling features to assess the performance of individual blocks of code, were used during development to great advantage. Since Matlab supports vector and matrix operations extensively, it is even here apparent how unnecessary loops in the code can lead to slow execution. However, it is clear that the Matlab version won't be used in the final iteration of the algorithm, because integration into ROS would be far too cumbersome.

ROS easily integrates code written in two languages; C++ and Python. Because of the similarity of syntax to Matlab, Python was decided upon as the language of choice. The fact that modules such as NumPy provide support for vector and matrix operations much in the same way as Matlab, and the MatPlotLib module has similar plotting capabilities, meant that the rewriting of the algorithm into Python was mostly a matter of changing function names. Of course, Python has its own optimized structures and procedures, many to do with memory allocation, list comprehension and so on.

These have also been taken advantage of.

The open-source nature and popularity of Python also mean that many user-made modules are available to provide the capabilities of certain libraries directly from Python. One of these is *OpenCV*[17], and optimized image processing library. Another module used in this thesis is *PyOpenCL*[12], which provides access to the OpenCL API and the possibility to insert OpenCL kernels written in native C code directly into the Python script. For the decoding of the QR codes, as well as performance comparison to commonly used methods, the *PyZbar*[19] library is used. This open-source library can detect and decode many different versions of QR codes and barcodes. Seeing as the *Zbar* library is serially run, but written in optimized C code, it will provide an interesting comparison with the results of the work done in this thesis.

4 QR code detection and decoding

4.1 Detection algorithm

As discussed in the previous chapter, the detection algorithm outlined in the QR code specifications will be used. An example image has been constructed from a real camera-acquired image. The reason for this is to fully illustrate the features of the selected detection algorithm, mainly the ability to detect multiple QR codes in an image, but also the ability to accurately extract even those codes distorted by perspective. It is assumed that the acquired image is an 8-bit grayscale image, that is, each pixel value is an 8-bit unsigned integer, meaning a maximum value of 255. The conversion of an image acquired by an RGB camera of a mobile robot to this format is a trivial matter and will not be further expanded upon in this thesis.

Figure 4.1: An example input image

4.1.1 Binarization

The first step is binarization, that is, the conversion of an acquired image into an array of boolean values. Each pixel thus contains a value of either 1 or 0, which is sufficient for the rest of the algorithm, but also allows us to save memory and computational time. The chosen method for binarization is local thresholding. In this method, we compare the value of a pixel to the average value of pixels in its surroundings. In practice, this is usually done by blurring the entire image by using a convolution with a uniform averaging kernel, because by blurring, individual pixels in the blurred image will also contain information about the surrounding pixels in the original image. The size of the averaged area corresponds to the size of the uniform kernel. Thus, all that is necessary to obtain a binarized version of our image is to compare the pixel value in the original to the pixel value in the blurred one, and if it is greater, it will be set to 1, otherwise to 0. A numerical constant is often subtracted from the blurred image to influence the distribution and amount of white/black pixels in the resulting image, often drastically reducing resulting noise. In our case, a kernel size of 33x33 and a constant value of 10 were used.

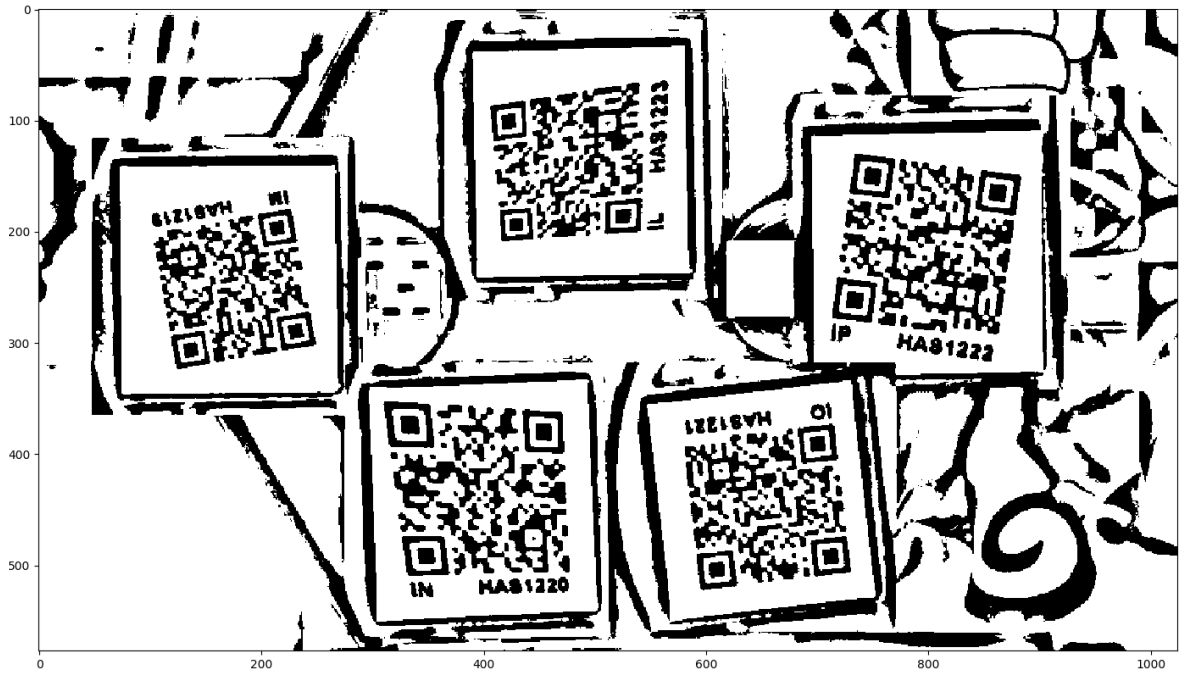

Figure 4.2: Binarized image

It should be noted that the QR code symbology specification [20] argues for the use of a global threshold. This approach is wholly unsuitable in non-even lighting conditions, for example when there is a shadow gradient over the QR code. In works referenced in this thesis, one of the global thresholding approaches tested was Otsu's method,

which assumes a bi-modal histogram of pixel values in a grayscale image, and then calculates an optimum threshold that minimized the variance within the two classes. It is, however, quite clear from the results of work on QR code detection, that local thresholding will always yield more satisfactory results. Even here, multiple approaches are possible. Besides the basic use of a uniform averaging filter, like in this thesis, other, more sophisticated methods can be used. Sometimes, a Gaussian filter is used instead of the uniform one. An interesting way to deal with image binarization is adaptive multi-level thresholding [26], which can also automatically choose an appropriate number of levels based on the histogram of pixel values, and can thus sometimes perform as global thresholding, while in more complicated cases it can give results similar to local thresholding [1].

4.1.2 Scanning lines

With a binarized image, we can start scanning the lines and columns of the image for the detection markers of QR codes. As explained in previous chapters, what we are looking for is a 1:1:3:1:1 ratio of black:white:black:white:black pixels. This operation is by far the most costly in terms of computational time. Thankfully, it is also completely independent within each row/column of the others. This makes it a prime candidate for parallelization. The general idea here is that we feed the image flattened into a 1-D array into a function (an OpenCL kernel) running on the GPU. This function will look up its global ID, and according to this number will start scanning a certain line or column of the image. Once the function has found a suitable marker center that satisfies the ratio condition, it saves its position and width and continues scanning. Multiple centers can thus be found in a single line/column. The kernel itself accepts three arguments as input:

```
__kernel void scan(__global const bool *im, __global const int *s,
    __global ushort *c)
```

The first argument is the binarized image flattened into a 1-D array, the second is an integer array containing two values needed to properly navigate the flattened arrays, and the last corresponds to a 1-D array of the 16-bit unsigned integer data type into which the found marker width and position are written.

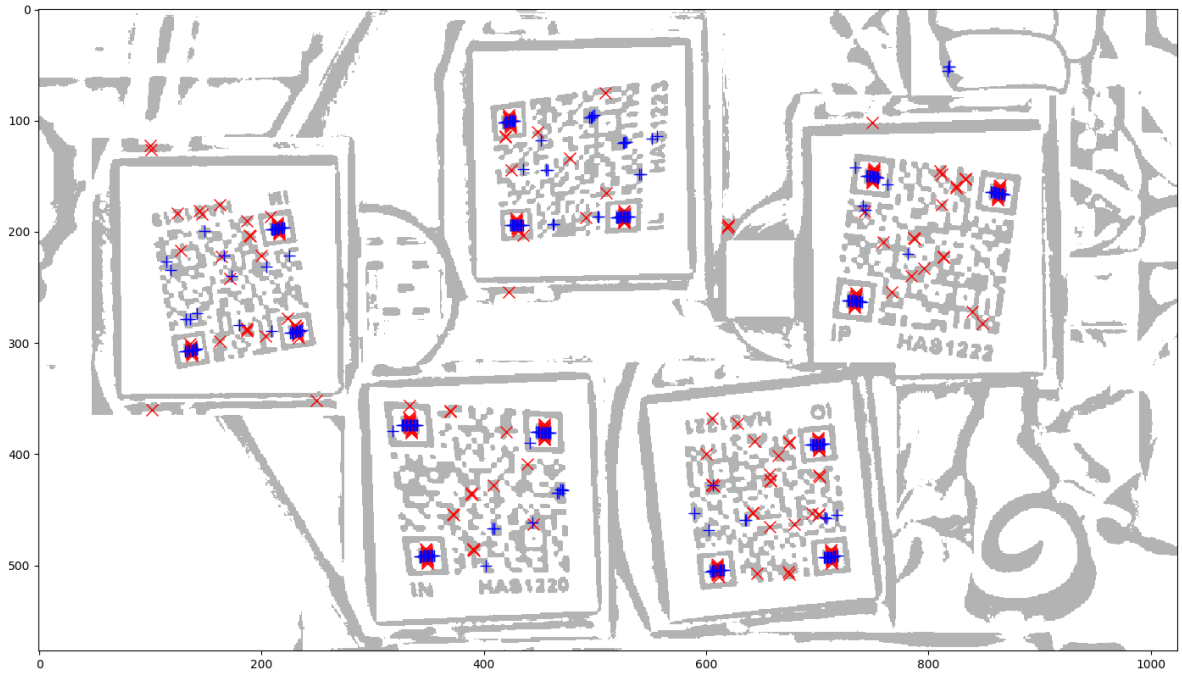

Figure 4.3: Detected markers in scanned lines and columns, differentiated by color

4.1.3 Finding clusters

After finding the marker center candidates in the rows and columns of the binarized image, we wish to cluster these together, separately at first. As we can see from the resulting image in the previous step, at the points where there truly are QR code markers, larger amounts of candidates are clumped together. By going through all the found row and later column marker candidates and computing their distances to each other, we can eliminate those that are far apart from any other. To this end, we use the information about the suspected marker width from the previous step. On the other hand, once there are several candidates found close to each other and of similar predicted size, we combine these into one a preserve them as an average of their individual sizes and positions. As we can see in the resulting image, this eliminates the vast majority of false positives and leaves us with reasonable assumptions about the QR code marker locations.

Figure 4.4: Clustered potential marker center points

4.1.4 Finding markers

Once we have clustered the candidates in both rows and columns, we perform the final step in finding the QR code markers. The assumption here is, that if a marker is truly present at a location in the image, there will be a candidate in both the rows and columns at that location. Any false positives still remaining after the previous steps should thus be eliminated. It should be noted that total elimination of false positives isn't a necessary condition for the rest of the algorithm to work, it does however reduce the computational complexity considerably. In the resulting image, we see that all 15 markers in the image were successfully found, in the case of this example (as in most tested cases) with no false positives.

Figure 4.5: Found markers

4.1.5 Finding QR codes

Having found the QR code markers, it is time to identify the locations of the actual codes. To this end, we go through the list of all found markers and try to match as closely as possible their size. We then try to find a triplet of markers that together form a right-angled triangle, with some leeway in the geometry to allow for distorted codes. Once three markers have been found to reasonably form the expected QR code shape, we decide which one corresponds to which corner, trying to correct for rotated codes. In the resulting image, the markers are labeled by color, with the bottom-left, top-left and top-right markers being indicated by red, green and blue, respectively.

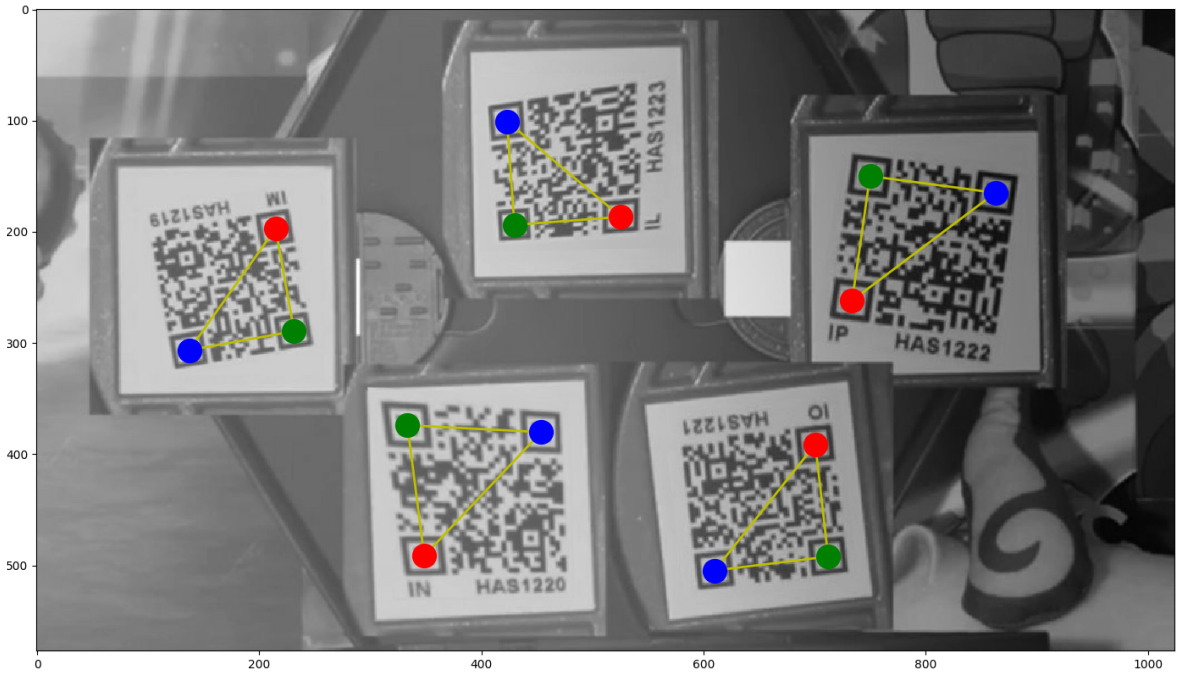

Figure 4.6: Found QR codes

4.1.6 Finding QR code corners

Now that we have found the QR codes and identified their orientation, we must find the locations of all four corners. This is because at least four points are needed if we want to estimate the homography transformation that would allow us to extract and rectify the found QR codes into their original square form.

Finding the three corners next to the markers is easy enough - we can simply scan the image in the expected direction of the corner, starting from a marker center, and once we hit the last black-to-white transition in both directions, we declare it to be the location of a corner. This has been found to work fast and reliably across all tested examples.

The location of the fourth corner is much trickier to find, because in an image distorted by perspective, angles and distances are not preserved, only lines. We cannot thus use any of the so far obtained information to directly compute the location of the fourth corner. What we can do, is take the three corners found so far and estimate the location of the fourth based on the assumption that the QR code corners form a parallelogram. In the case of an undistorted QR code, this will of course be a square and we will, in fact, find the fourth corner perfectly. In all other cases, we use this estimate as our starting position.

Figure 4.7: Found QR code corners

Multiple ways of finding the last corner were devised during the writing of this thesis, but most of them worked only under specific conditions and were too unreliable for general application. In the end, however, a reasonably robust implementation was found. The basic idea is to take the estimated location, try to move it in the expected direction of the last corner and all the while scan the border lines for changes in their variance. As this is done independently for both of these border lines, this will be explained for one of them and the reader will surely be able extend this to the other by analogy.

Let's say we have a distorted QR code, and the estimate of its fourth corner lies somewhere inside the code, not too far from the actual location of the real fourth corner. Following the convention of the previous step of the algorithm, the already found corners are labeled red, green and blue. If we now take the line from the blue corner to the estimate and read the pixel values underneath, it will be a mixture of black and white pixels, because the line crosses the data encoded in the code. If our estimate for the fourth corner was, however, located *outside* the code, the line would only contain white values. The assumption now is, if we compute the sum of the values of the line, normalized by the number of pixels underneath, and move the estimated corner along a direction towards the QR code edge, the greatest change in the sum will occur precisely at the boundary.

What we thus do, is move the estimated location of the fourth corner, compute the sum of the pixel values under the line from an already found corner to this estimate, save these values for different positions and then calculate a differential (which corresponds to a discrete derivative) of these values. The location with the greatest differential is the location at which our estimate hit the QR code edge, and if we thus do this for both of the edges, we should find the fourth corner.

It should be noted that after the part of our algorithm which does the scanning of lines and columns for QR code markers, the finding of the fourth corner is the most computationally demanding task. This is because during the search for the fourth corner, many lines between two points have to be rasterized and the pixels counted to obtain an array which we then differentiate. The rasterization is done by Bresenham's algorithm, one of the most well-known basic algorithms in image processing.

4.1.7 Extracted code

Once the four corners have been found, we isolate the QR code into its own smaller image and compute a homography transformation matrix that we then use to rectify the QR code into its original square shape. As this is a common operation in image processing, the optimized OpenCV library module in Python was used to perform these tasks. In our example, five codes were successfully extracted and on of them is in 4.8. It should be noted that the chosen algorithm for finding the last corner is robust enough that it found the true location, even though the corner pixel of the QR code is white, which has been a problem with all previously tested algorithms.

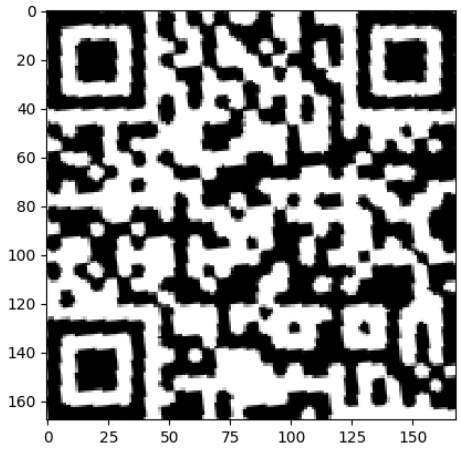

Figure 4.8: One of the extracted codes

4.2 Decoding

Once the QR codes are found and properly rectified, their decoding follows a set procedure outlined in the QR Code bar symbology specification [20]. The general sequence of actions taken during decoding is illustrated in 4.9. Since this procedure has been programmed an innumerable amount of times already, it was decided to not reinvent the wheel and instead use an open-source QR code-reading library, namely the Zbar library [18]. The library is highly optimized and able to decode many different variants of the code. It can also detect codes in an image, but this feature will not be used, since it is the topic of this thesis.

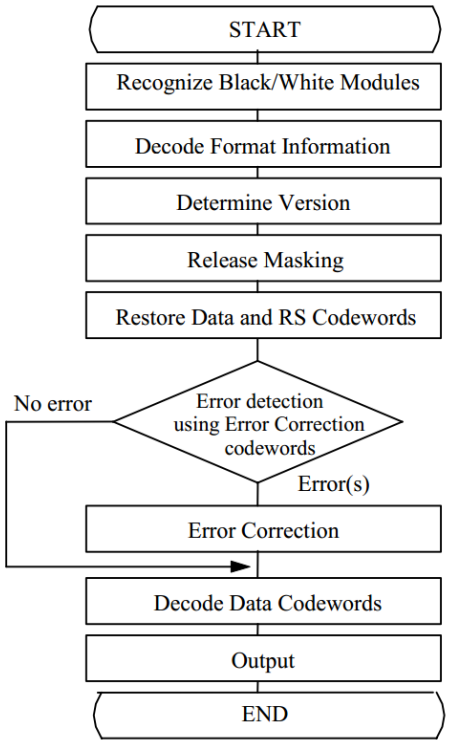

Figure 4.9: QR code decoding [20]

4.3 Implementation details

As should already be clear from previous chapters, the Python general purpose programming language was used for the bulk of the work. We use several non-core modules:

```
import numpy as np
import cv2
import pyopencl as cl
from pyzbar.pyzbar import decode, Decoded, ZBarSymbol
from timeit import default_timer as time
import matplotlib.pyplot as plt
```

Numpy is a module that contains many functions useful in scientific computing, like extensive support for matrices and matrix operations and other elements of linear algebra. *OpenCV* is an open-source image processing library, widely used in academic and scientific work due to its optimized performance. The library is also capable of using OpenCL for parallel computing on multi-core hardware. In this thesis, it was only used for the loading and saving of images, uniform filtering for local thresholding and the computation of the homography matrix used to rectify an extracted code into its original square shape. *PyOpenCL* is a module that allows its users access to the OpenCL parallel computation API directly from Python. Similarly, *PyZbar* is a Python module that allows the use of the optimized QR code reading library Zbar. The last two included modules were only used during testing - *timeit* contains functions for tracking the execution time of arbitrary blocks of code, the *matplotlib* module was used to show the results after every step.

After loading an image and converting it first to grayscale and then into a binary 2D-array by means of local thresholding, we perform the necessary tasks to initiate parallel processing. These include preparing the input data into the format needed by the kernel and preparing an array for storing the results. We set the global size equal to the number of rows (or columns in the next step), meaning there will be exactly that many tasks to be done. Next, we set the number of workers to be assigned to this task, which corresponds to the number of computing units we can use at a given time. After executing the code, we read back the buffer containing the results into our array and reshape it into a form that fits better with our postprocessing.

```
im_row = im_big.flatten()
result_num = np.int32(10)
c_result = np.zeros((row_num*result_num)).astype(np.uint16)
s = np.array([column_num, result_num]).astype(np.int32)

mf = cl.mem_flags
im_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=
    im_row)
s_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=s)
c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, c_result.nbytes)

global_size=(row_num,)
local_size=(workers,)

exec_evt = prg.scan(queue, global_size, local_size, im_buf, s_buf,
    c_buf)
exec_evt.wait()

c = np.zeros((row_num*result_num)).astype(np.uint16)

cl.enqueue_read_buffer(queue, c_buf, c).wait()

c = np.reshape(c, (row_num, result_num))
```

If we take a closer look at the parallel-running part of our algorithm itself, we see that it is a relatively simple series of conditions, attempting to detect a roughly 1:1:3:1:1 ratio of black:white:black:white:black in a single line or column of the image. In the beginning, we must retrieve the global ID of the worker. In the case of scanning the lines of an image, this will be the number corresponding to the line, and there will be as many IDs as there are horizontal lines of pixels in the image.

```
int gid = get_global_id(0);
```

With this ID retrieved, we can construct a for loop running through the pixels corresponding to the identified line.

```
for(int i=0; i < columnnum; i++)
{
        if (im[gid*columnnum + i] == current)
        {
                cnt += 1;
        }
                ...
}
```

What follows is a tree of conditions that checks whether a sequence of the last few detected pixels adheres to the color ratio we seek. As we will always have pictures of finite resolution and containing various distortions, we don't look for a precise ratio with units of 1 or 3. Instead, we look for values within a reasonable tolerance of these

idealized ones. As further and further conditions on the ratio are satisfied, the variable named *level* increases until it hits the final value of 6, at which point the position and the width of the currently found marker are returned from the function and the search begins anew in the remaining segment of the scanned line. The following code extract has been redacted for size to only show the parts deemed important. The full source code can be found in the file attachments of this thesis.

```
else if (level == 1)
{
        if (cnt > 0)
        {
                ...
        }
        cnt = 0;
}
else if (level == 3)
{
        if ((2.2*(float)base <= (float)cnt) && ((float)cnt) <=
           3.8*(float)base)
        {
                level = 4;
        }
        else
        {
                ...
        }
        ...
}
else
{
        if ((0.5*(float)base <= (float)cnt) && ((float)cnt) <=
           1.5*(float)base)
        {
                current = !current;
                level += 1;

                if (level == 6)
                {
                        c[gid*resultnum + valpos] = 2*startpos + i
                           - startpos;
                        c[gid*resultnum + (resultnum/2) + valpos] =
                            i - startpos;
                        ...
                        level = 1;
                }
        }

        ...
}
```

The preceding algorithm was the only part written as a C kernel for OpenCL, to be performed in parallel by available multi-core hardware. The advantages of this will be assessed in the following chapter, but it should be noted that other parts of our algorithm were considered for parallelization as well. One of these was the known Bresenham's algorithm for the rasterization of a line segment between two points. It takes four integers as inputs (the x and y coordinates of the start and end points) and returns an array containing the x and y values of the pixels between these.

```
def bresenham(x0, y0, x1, y1) :

    dx = x1 - x0
    dy = y1 - y0

    xsign = 1 if dx > 0 else -1
    ysign = 1 if dy > 0 else -1

    dx = abs(dx)
    dy = abs(dy)

    if dx > dy :
        xx, xy, yx, yy = xsign, 0, 0, ysign
    else :
        dx, dy = dy, dx
        xx, xy, yx, yy = 0, ysign, xsign, 0

    D = 2*dy - dx
    y = 0

    for x in range(dx + 1):
        yield x0 + x*xx + y*yx, y0 + x*xy + y*yy
        if D > 0:
            y += 1
            D -= dx
        D += dy
```

There are many ways to cluster the points of detected markers. The one that was used in our implementation is simply takes points in sequence, computes their distances to the ones not yet processed, and if they are found to be close enough with respect to the presumed width of the marker, they are clustered together. The relevant part goes as follows:

```
dist = np.abs(linex[(i+1):] - linex[i]) + np.abs(liney[(i+1):] -
    liney[i])

if np.min(dist) < (width[i]/4.) :

        dset = [np.where(dist<(width[i]/2.)), np.where(width[(i+1)
            :] > 0.8*width[i]), np.where(width[(i+1):] < 1.2*width[i
            ]), np.where(clust_num[(i+1):]==range((i+1),maxcnt))]
```

Similarly, the clustered points in x and y are then taken and their distances compared. If a marker was detected in both a line and a row close to each other, we combine them into a marker.

```
for i in range(maxcnt) :
        dist = np.abs(clustxx[i]-clustyx) + np.abs(clustxy[i]-
            clustyy)

        dset = [np.where(dist<(clustxwidth[i]/4.)), np.where(
            clustxwidth[i] > 0.5*clustywidth), np.where(clustxwidth[
            i] < 1.5*clustywidth)]
```

The remaining parts are either too long or too trivial to be included here, so we'll only note that after finding all four corners of the QR code, two OpenCV functions are used to find the homography transformation and for the rectification of the code into a square shape by the use of this homography. The first function takes the four corner points as inputs, and we set the output points as corners of a square with a side length equal to the shorter of the two dimensions of the QR code isolated from the original image.

```
h, status = cv2.findHomography(pts_src, pts_dst)

im_new = cv2.warpPerspective(im_separated, h, (squaresize,
    squaresize))
```

4.4 Integration into ROS

The integration of the written algorithm into ROS is fairly straightforward, as it is already written in Python and the only remaining thing to take care of is to write functions to facilitate the communication between ROS and our module. To receive data from our module, we create a so-called *publisher*[30], which takes care of periodically sending an image source from the camera. Then we create a *listener*, which will detect if an image was sent by the publisher and will execute a callback function, in which the image is sent to our module and decoded. We use the message type *Image*, which stores the image data in an array of the data type uint8. If we assume that our QR code decoding module is named *qrreader* and it contains a function *decode()*, we simply log the output string in the following code:

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import Image
import qrreader

def callback(data):
    rospy.loginfo("The decoded string is %s", qrreader.decode(data.
        data))

def listener():
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("img", Image, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

5 Results

With the implementation specified in the preceding chapter, we now take a look at the results. Of course, the main benchmark has to be the successful decoding of a code, regardless of the time. Given that we merely adopted the algorithm outlined in the official QR code specifications, it is not surprising that our implementation performs very similarly to other implementations based on this algorithm, i.e. all cases that are undetected by our implementation usually remain undetected also when using other libraries for reading QR codes. It is then time to look at the performance in terms of decoding speed, and to look whether the parallelization of line and column scanning truly gives us the performance boost we expected.

5.1 Performance comparison

The following bar chart 5.1 shows a speed comparison of three different approaches - a purely Pythonic implementation, an implementation with the use of OpenCL for line/column scanning, and an implementation which only uses the PyZbar wrapper which executes the Zbar library under Python, whereas in the first two cases it is only used to decode already extracted QR codes. These have all been tested on the same images with various numbers of QR codes in them. The parallel computing is done on a GeForce GTX 960M GPU.

The first two comparisons were made on images with no QR codes in them. This is perhaps the most important test, since the program for QR code detection is supposed to be running all the time on the intended platform (a mobile robot which interacts with humans), meaning that its execution will always need computational resources and will try to take those from other, usually more crucial real-time tasks. In these first two test cases, we can clearly see that an implementation of the algorithm running purely in Python in a serial way is entirely unsuitable for the intended use. On an image in HD resolution (1920x1080), this implementation spent over three seconds reading the image, only to find no codes (since there were none to be found). For the purely Pythonic algorithm, we see that the execution time is mostly dependent on image resolution, much more so than on the number of QR codes in the image. From this we can deduce that it is indeed the line/column scanning which will take up most of the resources in QR code detection and decoding.

Once we look at the Python implementation with the use of OpenCL for QR code

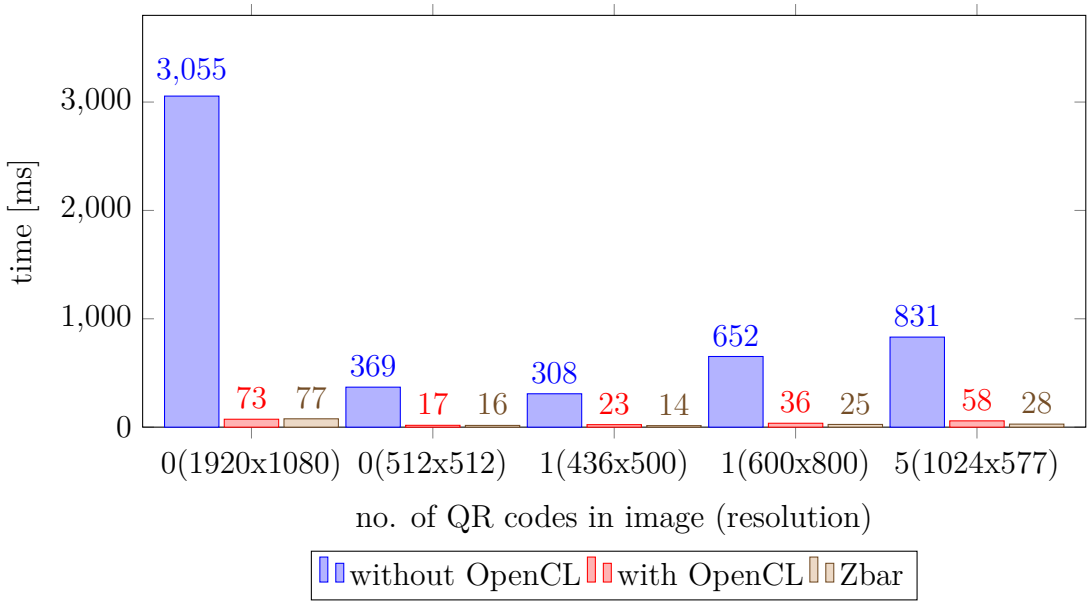

Figure 5.1: Performance comparison

marker detection, we immediately see the advantage of parallelization. The execution time is drastically reduced, so much so that this implementation is basically on par with the Zbar optimized C library, even though the majority of tasks are still handled by the comparatively slow and high-level Python general-purpose programming language. Times in the tens of milliseconds mean that this implementation is usable in real-time, with ordinarily used sampling rates in the cameras of mobile robots. We might at first wonder why the execution time of this variant in the case of five QR codes is roughly double that of Zbar, but we must realize that in its current form and use, Zbar quits execution once it scans one QR code, while the algorithm devised in this thesis will detect and decode any and all of them.

Furthermore, while the runtime of the latter two compared variants is roughly equal, it should be noted that the main intended advantage is hidden in this comparison - that is, that the most demanding code is now running on the GPU, leaving the CPU free to do other vital tasks. The results of the comparison are highly encouraging for future work on this problem, as it can be expected that with the conversion of more partial tasks into parallel, GPU-executable code, further gains in performance will be noticed. As the program so far was mostly written to merely satisfactorily perform its function, but without a large focus on optimization, we can conclude that the rest of the tasks that will remain performed serially in Python can also be improved upon.

There might, however, be a limit to the proposed improvement by parallelization. During work on this thesis, an attempt was made to rewrite the known Bresenham's algorithm for the rasterization of a line into OpenCL-executable code. However, no

significant improvement in performance was measured. This is due to the fact that the operations performed on the data in order to prepare them for parallel processing, as well as the initialization of the OpenCL API for computations, simply take too much time, so much as to almost offset the gains in performance from multi-core code execution.

5.2 Limits of detection

While the algorithm performs to expectations in most cases, we can still find some where it fails. In the vast majority of cases, it is caused by the low resolution of the input image, or simply the small size of the code within the obtained image. This leads to problems in the first parts of the algorithm, where QR code markers are being scanned for. When the cell size of a QR code corresponds to only a couple pixels in the image, it can be very hard to successfully find the desired 1:1:3:1:1 ratio needed for marker detection. Mostly, this is caused by binarization, and this problem is inherent to all implementations of the detection algorithm outlined in the QR code specification, including the one used in the Zbar library. It can be assumed that a different algorithm that would intelligently use the pixel values in the grayscale image could be more suitable in these lower-resolution images. Another way to circumvent this limitation would be to use an entirely different approach of marker detection, such as an approach based on phase correlation or feature detection.

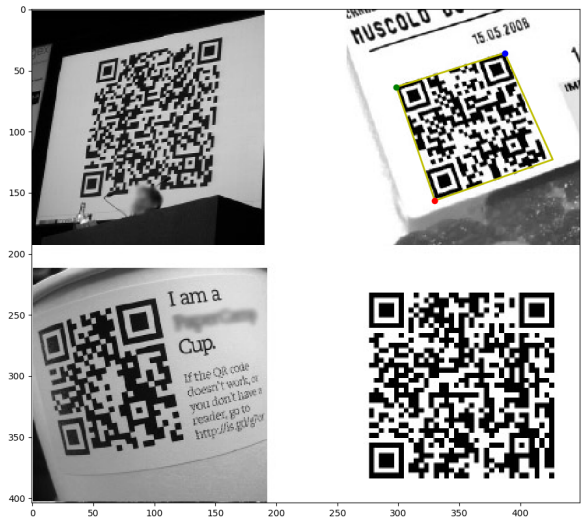

Figure 5.2: A test case with unsatisfactory results

As an illustration of this limitation, 5.2 shows an image containing four QR codes, of which only one (in the upper-right corner) is detected and decoded. This is because

the binarization of the three undetected codes has caused the ratios of black and white in their markers to be outside the tolerances programmed into the algorithm. It is presumed that, given a somewhat larger resolution of this image, all should be successfully decoded even by the algorithm from the QR code specification.

There are also distortions that are somewhat more complex and require a more developed mathematical framework to rectify. One of the more common ones occurs when the QR code lies on a curved surface, such as the bottle in 5.3. The algorithm from the QR code specifications will reasonably well find the localization markers themselves, however, the problem occurs when looking for the corners of the QR code. This is because we assume that the only distortion is caused by the perspective, meaning that lines remain straight, which is no longer true in this case. Sure enough, work has been done on these kinds of distortions, proposing various solutions [4] [23] [24], it is however seen as beyond the scope of this thesis to deal with them.

Figure 5.3: A QR code distorted by being printed on a curved surface

6 Conclusion

In this master's thesis, the usage of parallel computing applied on the task of detecting QR codes was considered. In the first part, a survey was conducted to gain familiarity with the current state-of-the-art solutions. The structure and specifications of QR codes were briefly discussed, as well as the basics of parallel computing. Also, a short introduction to the ROS platform was given.

The following section had to do with the choice of proper tools for our task. Out of the available parallel computing libraries, mainly two were considered - Nvidia CUDA and OpenCL. Of these two, OpenCL was chosen as a suitable library, mainly because of its support of many different parallel computing platforms, as well as the author's familiarity with this platform.

For the detection algorithm itself, it was decided to base it on the algorithm outlined in the QR code specifications. This algorithm scans the lines and columns of an image to detect the three square corner marks, for which they were specifically devised. The image is converted to grayscale and binarized by the use of local thresholding beforehand. The line and column scanning was chosen as a prime candidate for parallelization, as the scanning algorithm runs independently for each line and column of the image. Afterwards, the points are clustered and QR code location and pose are estimated based on the predicted geometry that the corner markers form. The found code is cut out of the image and rectified into its original square shape, allowing it to be read and decoded by standard algorithms. The decoding is done by the Zbar open-source library, which is able to decode various types and sizes of QR codes and is highly optimized for this purpose.

Besides the OpenCL kernel used for the parallel task of scanning lines and rows of an acquired image, the program is written in Python. This was mainly chosen due to its open-source license, ease and speed of development, and availability of wrappers for many useful libraries. In our case, the PyOpenCL wrapper was used to access the OpenCL API functions, and the PyZbar wrapper for the QR code decoding library Zbar was used as well. Another advantage is the fact that Python code can be integrated into the ROS platform with ease.

The implementation of the QR code reading a decoding algorithm outlined in this thesis was tested to assess the usefulness of using parallel computing for this task. It was found that, with OpenCL code running on a GPU, performance was as much as 50 times better in terms of speed than a purely Pythonic serialized approach, with the

same accuracy of decoding information from QR codes. In fact, the performance of the Python implementation with one OpenCL kernel was comparable to the highly optimized C library Zbar. This hints at the possibility of future improvements, with which our algorithm could perform better than other, currently used methods. Such improvements include the rewriting of other functions into parallel tasks with OpenCL. The place to start would probably be the clustering algorithm or the algorithm for searching for the fourth corner of a QR code, which could take advantage of parallelization and are currently the most computationally intensive parts of the algorithm.

Other room for improvement is in the overcoming of the current limitations on image distortions. Better thresholding methods or an improved version of the line scanning algorithm based on multi-level grayscale images could improve the accuracy of detection even in lower-resolution images. A more sophisticated mathematical processing of found QR codes would also allow us to rectify codes that have been printed on curved surfaces, such as cylinders.

All tasks specified in the assignment were completed successfully.

Bibliography

[1] Y. Liu and M. Liu, "Automatic recognition algorithm of quick response code based on embedded system," in *Intelligent Systems Design and Applications, 2006. ISDA'06. Sixth International Conference on*, vol. 2. IEEE, 2006, pp. 783–788.

[2] L. Belussi and N. Hirata, "Fast qr code detection in arbitrarily acquired images," in *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*, Aug 2011, pp. 281–288.

[3] S. Kong, "Qr code image correction based on corner detection and convex hull algorithm." *Journal of Multimedia*, vol. 8, no. 6, pp. 662–668, 2013. [Online]. Available: <http://dblp.uni-trier.de/db/journals/jmm2/jmm8.html#Kong13>

[4] K. T. Lay, L. J. Wang, P. L. Han, and Y. S. Lin, "Rectification of images of qr codes posted on cylinders by conic segmentation," in *2015 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, Oct 2015, pp. 389–393.

[5] W. Chen, G. Yang, and G. Zhang, "A simple and efficient image pre-processing for qr decoder," in *Proceedings of the 2nd International Conference on Electronic and Mechanical Engineering and Information Technology (2012)*. Paris, France: Atlantis Press, 2012, pp. –. [Online]. Available: <http://www.atlantis-press.com/php/paper-details.php?id=3264>

[6] M. Katanacho, W. D. la Cadena, and S. Engel, "Surgical navigation with qr codes," *Current Directions in Biomedical Engineering*, vol. vol. 2, no. issue 1, pp. –, 2016-01-1. [Online]. Available: <http://www.degruyter.com/view/j/cdbme.2016.2.issue-1/cdbme-2016-0079/cdbme-2016-0079.xml>

[7] E. Marchand, H. Uchiyama, and F. Spindler, "Pose estimation for augmented reality: A hands-on survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 12, pp. 2633–2651, Dec 2016.

[8] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[9] M. Quigley, B. Gerkey, and W. D. Smart, *Programming robots with ROS*, 1st ed. O'Reilly Media, 2015.

[10] A. Koubaa, *Robot Operating System (ROS): The Complete Reference (Volume 1)*, 1st ed. Springer Publishing Company, Incorporated, 2016.

[11] P. Bakowski, *A Practical Introduction to Parallel Programming on multi-core and many-core Embedded Systems*. Amazon, 2016.

[12] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[13] P. Schreiber, "Realizace vybraných výpočtů pomocí grafických karet," Master's thesis, Vysoké učení technické v Brně, Fakulta strojního inženýrství, Brno, 2010.

[14] A. C. Leighner, "Fpga accelerated discrete-surf for real-time homography estimation," Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2015.

[15] Khronos, "The opencl specification," Khronos OpenCL Working Group, p. 299, 2015. [Online]. Available: <https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf>

[16] J. Krejsa, S. Věchet, J. Hrbáček, and P. Schreiber, "High level software architecture for autonomous mobile robot," *Recent Advances in Mechatronics*, p. 185, 2010. [Online]. Available: <http://link.springer.com/10.1007/978-3-642-05022-0_32>

[17] R. Laganiére, *OpenCV 2 computer vision application programming cookbook*, 1st ed. Brimmingham: Packt Publishing, 2011.

[18] J. Brown, *ZBar bar code reader*, 2011, accessed 09.05.2017. [Online]. Available: <http://zbar.sourceforge.net/index.html>

[19] L. Hudson, *Pyzbar - a ctypes-based Python wrapper around the zbar barcode reader*, 2017, accessed 09.05.2017. [Online]. Available: <https://github.com/NaturalHistoryMuseum/pyzbar>

[20] *QR Code bar code symbology specification ISO/IEC 18004:2015*, DENSO WAVE INCORPORATED, 2015. [Online]. Available: <https://www.iso.org/standard/62021.html>

[21] *QR Code History*, DENSO WAVE INCORPORATED, 2017, accessed 09.05.2017. [Online]. Available: <http://www.qrcode.com/en/history/>

[22] *QR Code Essentials*, DENSO WAVE INCORPORATED, 2011, accessed 09.05.2017. [Online]. Available: <http://www.nacs.org/LinkClick.aspx?fileticket=D1FpVAvvJuo%3D&tabid=1426&mid=4802>

[23] X. Li, Z. Shi, D. Guo, and S. He, "Reconstruct argorithm of 2d barcode for reading the qr code on cylindrical surface," *2013 International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, pp. 1–5, 2013. [Online]. Available: <http://ieeexplore.ieee.org/document/6825309/>

[24] Z. Shi, X. Li, C. Zhao, and S. Lin, "Extraction of qr codes on flat and cylindrical surface," in *Proceedings of the 2012 Second International Conference on Electric Information and Control Engineering - Volume 01*, ser. ICEICE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 694–697. [Online]. Available: <http://dx.doi.org/10.1109/ICEICE.2012.179>

[25] Wikipedia, *various images*, accessed 13.05.2017. [Online]. Available: <https://en.wikipedia.org/wiki/QR_code>

[26] S. Arora, J. Acharya, A. Verma, and P. K. Panigrahi, "Multilevel thresholding for image segmentation through a fast statistical recursive algorithm," *Pattern Recognition Letters*, vol. vol. 29, no. issue 2, pp. 119–125, 2008. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0167865507002905>

[27] *Nvidia CUDA*, 2017, accessed 11.05.2017. [Online]. Available: <http://www.nvidia.com/object/cuda_home_new.html>

[28] Accessed 11.05.2017. [Online]. Available: <https://scs.senecac.on.ca/~gpu610/pages/images/openclMemory.png>

[29] *An Easy Introduction to CUDA C and C++*, 2017, accessed 11.05.2017. [Online]. Available: <https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/>

[30] *Writing a Simple Publisher and Subscriber*, 2017, accessed 11.05.2017. [Online]. Available: <http://wiki.ros.org/rospy_tutorials/Tutorials/WritingPublisherSubscriber>

List of Figures