CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

Efficient Pattern Detection for Visual Guided Landing of Unmanned Aircraft

Martin Jahn
Open Informatics - Computer and Information Science

May 2019
Supervisor: Ing. Michal Zajačík, AgentFly Technologies s.r.o., Praha; Ing. Milan Rollo, Ph.D., Artificial Intelligence Center, FEE

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jahn Martin** Personal ID number: **437340**

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 Pattern Detection for Visual Guided Landing of Unmanned Aircraft

Bachelor's thesis title in Czech:

Efektivní detekce vzorů pro vizuálně naváděné přistávání bezpilotního stroje

Guidelines:

1) Study the problems of vision-based landing on a pattern of VTOL UAVs.
2) Study the possibility of programming for graphical processors.
3) Create a suitable visual pattern and implement a robust method for its detection.
4) Modify the used method to be executable on GPU.
5) Compare the speeds of both methods and quality of detection.

Bibliography / sources:

[1] Cocchioni, F., Frontoni, E., Ippoliti, G., Longhi, S., Mancini, A. and Zingaretti, P., 2016. Visual based landing for an unmanned quadrotor. Journal of Intelligent & Robotic Systems, 84(1-4), pp.511-528.
[2] Pulli, K., Baksheev, A., Kornyakov, K. and Eruhimov, V., 2012. Real-time computer vision with OpenCV. Communications of the ACM, 55(6), pp.61-69.
[3] Thiang, I.N. and LuMaw, H.M.T., 2016. Vision-based object tracking algorithm with ar. drone. Red, 160, p.179.
[4] Boyers, O.H., 2013. An Evaluation Of Detection and Recognition Algorithms To Implement Autonomous Target Tracking With A Quadrotor. Grahams town, South Africa, page (5-53).
[5] Barták, R., Hrasko, A. and Obdrzálek, D., 2014, May. On Autonomous Landing of AR. Drone: Hands-On Experience. In FLAIRS Conference.

Name and workplace of bachelor's thesis supervisor:

Ing. Michal Zajačík, AgentFly Technologies s.r.o., Praha

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

Ing. Milan Rollo, Ph.D., Artificial Intelligence Center, FEE

Date of bachelor's thesis assignment: **12.01.2018** Deadline for bachelor thesis submission: **24.05.2019**

Assignment valid until: **30.09.2019**

Ing. Michal Zajačík	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

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

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgement / Declaration

I thank to family for all the material and moral support and for getting me so far.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 24. 5. 2019

.....

Abstrakt / Abstract

Systémy, které vužívají bezpilotních letounů, vyžadují, aby tyto letouny byly schopny autonomně přistávat. Tato práce se soustředí na efektivní detekci přistávacích vzorů. Tato práce bude obsahovat návrh vzoru na základě již exitujících variant a implementaci metody, která bude detekovat navrhnutý vzor. Následně bude metode ještě modifikována pro spuštění na grafickém procesoru.

Klíčová slova: Zpracování obrazu, Paralelní Výpočty, Bezpilotní letouny, Vizuální Navádění

Překlad titulu: Efektivní detekce vzorů pro vizuálně naváděné přistávání bezpilotního stroje

Autonomous landing of Unmanned Aerial Vehicles (UAVs) is required for any system that utilizes such vehicles. This thesis focuses on efficent detection of artificial marker known as landing pattern. In this thesis a pattern will be created based upon existing variants and a method will implemented to detect such pattern using a camera. Then the method will be modified to run on a Graphical Processing Unit (GPU) to gain a performance boost.

Keywords: Image Processing, Parrallel Computing, UAV, Visual Guiding

/ Contents

Tables / Figures

Chapter 1
Introduction

Small Unmanned Aerial Vehicles (UAVs) gained interest of research communities and military/civilian companies in recent years. Due to their small size, high agility, reprogrammability they can be used in various missions both indoors and outdoors. They can sent on missions which could prove dangerous to normal presonnel such mapping out dangerous areas, damaged buildings or missions where success rate depends on time elapsed e.g. search and rescue missions and speed these missions up.

One major drawback of these UAVs is short battery life. Currently these quarotors can last anywhere between 10 and 30 minutes [1]. To overcome this drawback several automatic battery recharging solutions were developed that do not rely on extension cord plugged into a UAV, which would sacrifice its mobility. Charging pads [2] are a solution that requires a drone to land on a platform. These charging pads do not require precision landing. When the quadrotor lands on the platform charging can proceed regardless of orientation, size or position of quadrotor on platform. Charging process takes between tens of minutes to several hours which can have negative impact on mission efficiency. Other solution is to change the battery using electromechanical system [3] which takes several seconds but requires precise positioning and orientation on platform.

Figure 1.1. Charging pad with a quadrotor [2]

Several approches can be used to land on these charging platforms. Possible solutions include using Global Positioning System (GPS) or vision based approaches or combinations. Precision landing can be easily achieved using artificial landmarks detected by vision based approach [4]. Artificial landmarks can be modified to suit either of these charging solutions and their use is not limited to quadrotors guidance only. Other Vertical Take-Off and Landing (VTOL) Unmanned Aerial Vehicles(UAVs) can utilize this approach for guidance.

1.1 Organization of thesis

In this thesis I will focus on vision based detection of artificial landmarks. I chose two landmarks for detection: a symmetrical landmark that can be utilized in situations

where only precise positioning is required but there are no requirements for orientation and a asymmetrical landmark when there are requirements for both position and orientation.

Then I will implement robust detection method in c++ using OpenCV library [5]. The goal of this method is: in an image detect the position of pattern and orientation if appliable. Then I will modify the method to be executable on a Nvidia Graphical Processing unit (GPU) using existing OpenCV methods for GPU to increase its speed. At last I will compare the original method against its GPU modification performance-wise – Time needed for computing and correct classifications.

Chapter 2
Landing Patterns

It is possible to design many different landing patterns using artificial landmark. Such patterns have different properties and varying degree of robustness. I classified patterns as follows:

- Colored versus black and white.
- Geometric shapes versus QR codes and apriltags.
- With bounding shape versus without bounding shape.

The properties of landing patterns I consider important for robustness are:

- Resistance to pattern illumination changes.
- Resistance to disruptive elements - elements that are similar to targeted landing patterns or pattern's components.
- Ability to be detected during all stages of landing (over different distances).

2.1 Properties of patterns

2.1.1 Resistance to pattern illumination changes

As the illumination of pattern will change depending on various factors (time of the day, shadows, artifical light sources, ...), the color of pattern and its instensity will also change too. This may cause problems for colored patterns [6], [7]. In extreme cases mono-colored light could disrupt the detection process completely.

Black and white patterns [4] depend on intensity of light rather than its color are more resistant to this circumstance. Black and white patterns are expected to start failing when there is enough light to illuminate the pattern (night, unlit windowless room, ...).

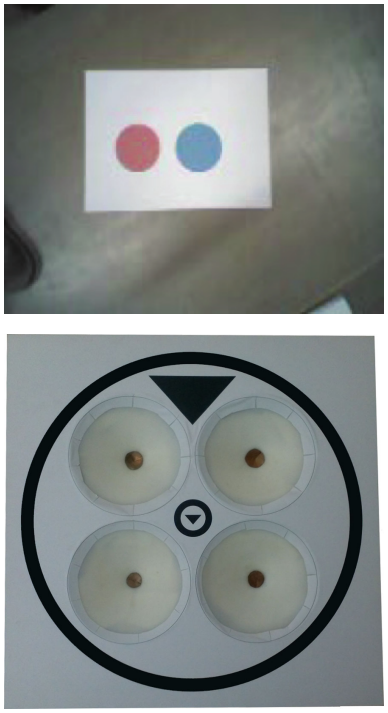

Figure 2.1. Colored pattern on top [7] and black/white pattern on bottom [4]

2.1.2 Resistance to disruptive elements

Colored patterns could also be problematic for one more reason. Introducing a similarly colored could cause problems for the detection method, if the method is insuficiently sophisticated.

Over-simplified patterns (e.g. sigle circle, single square) are also prone to disruptions, because objects similar to these pattern could disrupt the detection process completely.

One way to battle this situation is have a pattern with bounding shape (e.g. circle) as can be seen bottom part of figure 2.1. The detection can be restricted to part of image inside the bounding shape.

2.1.3 Ability to detect pattern during all stages of landing

During the landing some parts of patterns can get out of field of vision. Suitable landing patterns should have enough defining features to be detectetable at close distance while not having excessive amout of features to not be detectable over larger distances.

Apriltags [8],[9] are patterns that could have some their features undetectable over larger distances and could be properly detected only from closer distances.

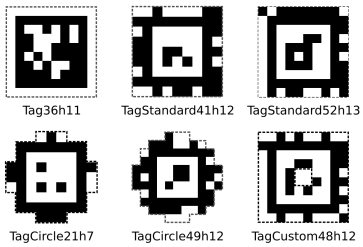

Figure 2.2. Apriltag example [9]

Several approaches can be employed for keeping the pattern detectable at wide ranges of distances. First pattern can have a smaller version at the landing point [4], recursion [10] or by detecting only part of the pattern at close ranges [11], [12].

2.2 Used patterns

I have created two patterns based on cross in a circle pattern [11], [12]. The symmetrical is modified cross in a circle and the asymmetrical is 3 triangles. The top triagle is the asymmetrical pattern has one vertex at the center of the circle so it makes the detection more straightforward. Both of these patterns can be scaled up, to increase detection range if required.

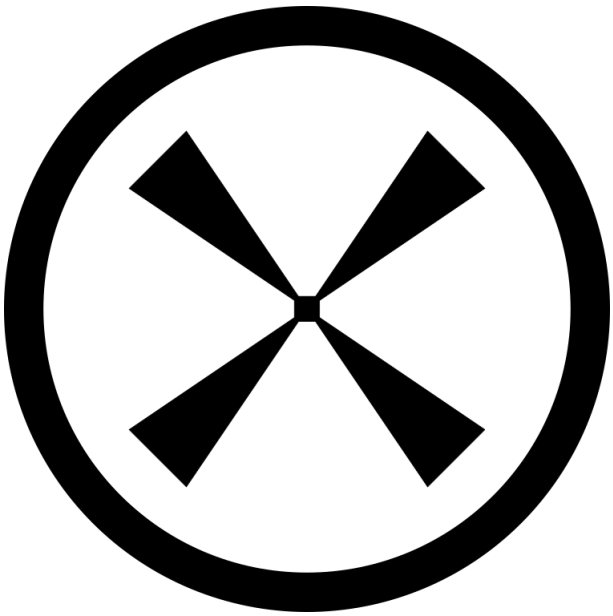

Figure 2.3. Symmetrical pattern

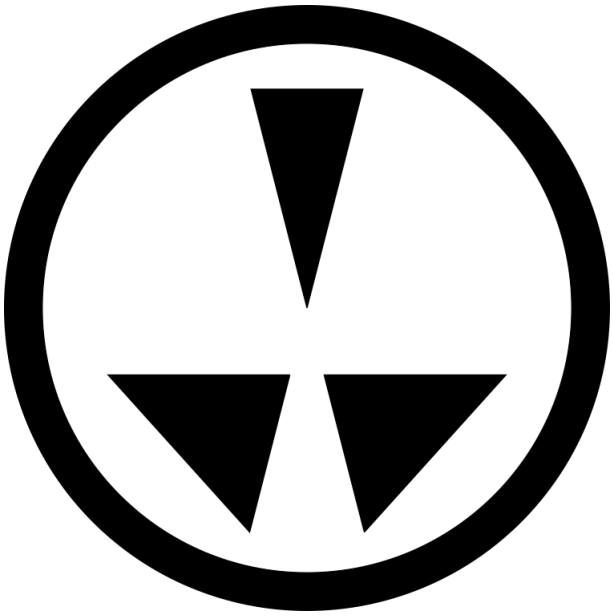

Figure 2.4. Asymmetrical pattern

Chapter 3
GPU acceleration

In my detection method I want to utilize a Nvidia GPU (Graphical Processing Unit) [13] to speed up the detection. I want to use the GPU to mainly boost image processing. Modern GPUs are specialized utilized for highly intensive computational tasks using high amount of threads at the cost of flow control and data caching, which primarly CPU's job today. GPUs are mainly used for graphical rendering and image/video processing. [14]. Also Nvidia is currently dominating in fields of Machine Learning and Artificial Intelligence thanks to its CUDA platform [15]. While there other notable companies making video cards (e.g. AMD) in market Nvidia is leader in this field.

3.1 Difference between GPU and CPU

Today main difference CPU and GPU is that CPU mainly handles flow control, and data caching while GPU handles tasks that are computationally demanding. GPU achives this by massivelly multithreading tasks that are given to it. GPU is focused mainly on computational throughput and very high memory bandwidth [14].

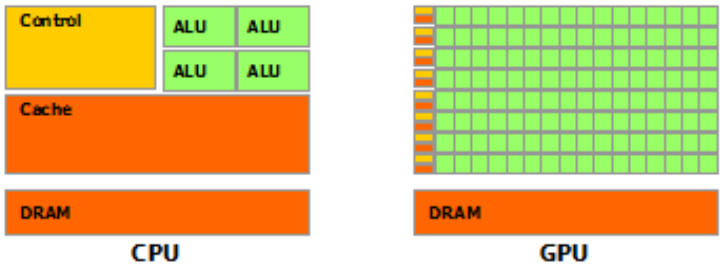

Figure 3.1. Simplified chip schematic of CPU (on left) and GPU (on right)[14]

As it can be seen in figure 3.1, GPU dedicates most of its transistors to data processing while CPU on the other hand devotes most of its transistors to other tasks. In figure 3.2 and figure 3.3 can be seen how much has Intel's CPUs has fallen behind Nvidia's GPUs in terms of computational power and memory bandwith.

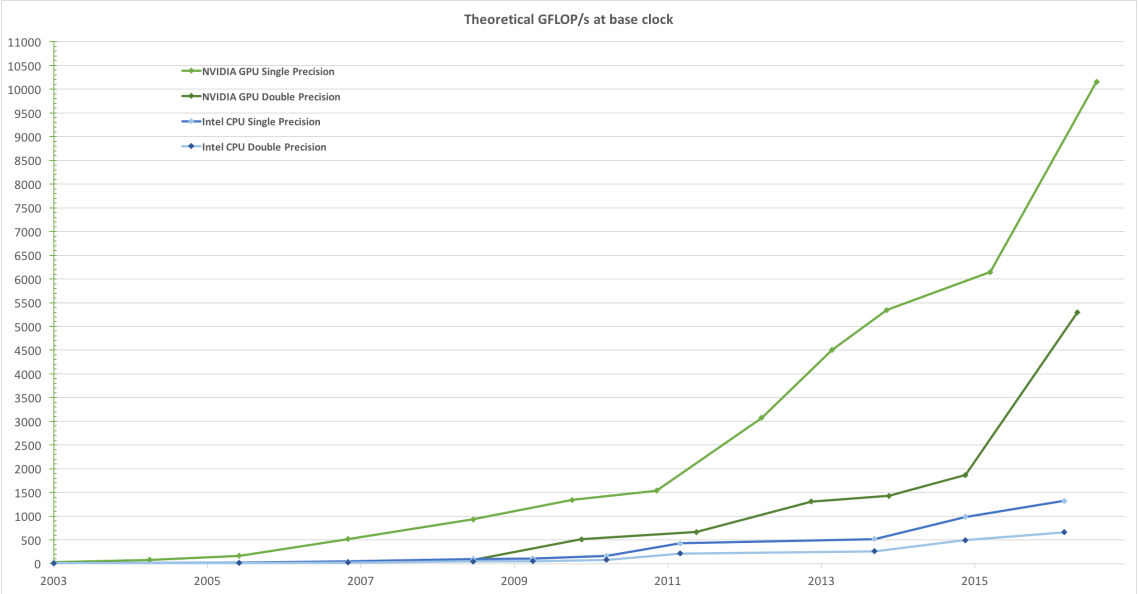

Figure 3.2. Comparison of floating point operations per second CPU between GPU[14] (GPU is green)

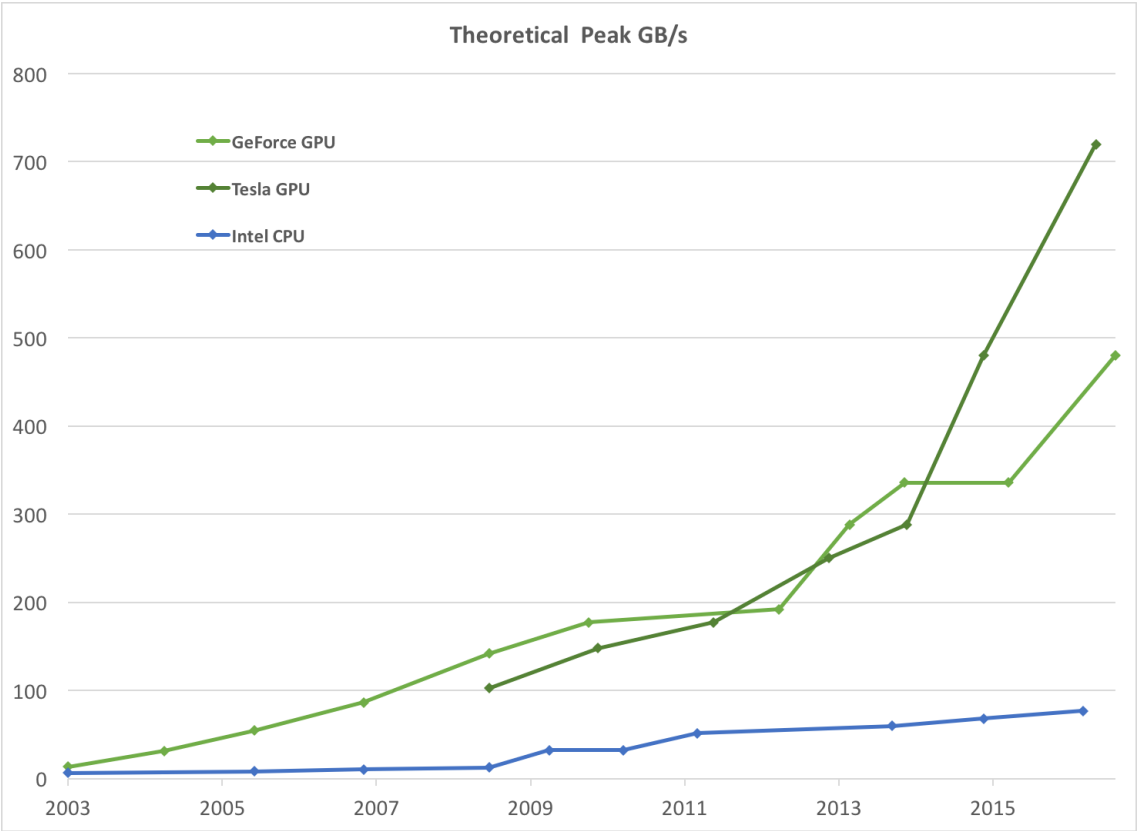

Figure 3.3. Comparison of memory bandwith CPU between GPU[14] (GPU is green)

Still the main bottleneck is the separation of Nvidia GPU's memory and CPU's memory. The rate of data transfer between CPU and GPU is getting better over time, but its not worth tasking GPU to work with small amount of data because the losses

of time by transfering the data between CPU and GPU can be bigger than time gained from processing the data on GPU instead of CPU. Good practice to gain most out of the GPU is to do as much processing on the data as possible before CPU has to retrive it from GPU's memory [14].

3.2 CUDA platform

CUDA is a parralel computing platform and programming model developed by Nvidia to leverage its GPUs [14], [15]. It allows to C programming language to used as a high-level language. CUDA also has C++ language support.

When develeping with CUDA in C++, the code splits into host code (ran on CPU) and device code (ran on GPU), so it makes programming for Nvidia cards in C/C++ convenient because writing code for GPU is very similar to writing code for CPU. Device code is implemented using kernel functions. These kernel functions are then called from host's code [14].

Chapter 4
Implementation

I implemented the application in C++ with OpenCV 3.4 and CUDA 10 Toolkit. Application is object oriented and both detection methods reside in their own class with shared parent class. Their GPU counterpars have also their own class.

The application can be simplified into following pseudocode:

```
image = loadimage()
grayImage = convertToGrayscale(image)
blurredImage = gaussianBlur(grayImage)
binaryImage = convertToBinary(blurredImage)
contours = findContours(binaryImage)
ellipses[] = detectEllipses(contours)

for each e in ellipse do:
   crop = cropImage(binaryImage)
   edges = cannyEdges(crop)
   lineSegments = houghSegments(edges)
   joinedLines = joinLines(lineSegments)
   intersections = findIntersection(joinedLines)
   patternFound, patternPosition = detectShape(intersections)
   if patternFound do:
      return patternPosition
   end
end

edges = cannyEdges(binaryImage)
lineSegments = houghSegments(edges)
joinedLines = joinLines(lineSegments)
intersections = findIntersection(joinedLines)
if patternFound do:
   return patternPosition
end

return [-1, -1]
```

I designed the application using C++ classes. Object hiearchy can be found in figure 4.1.

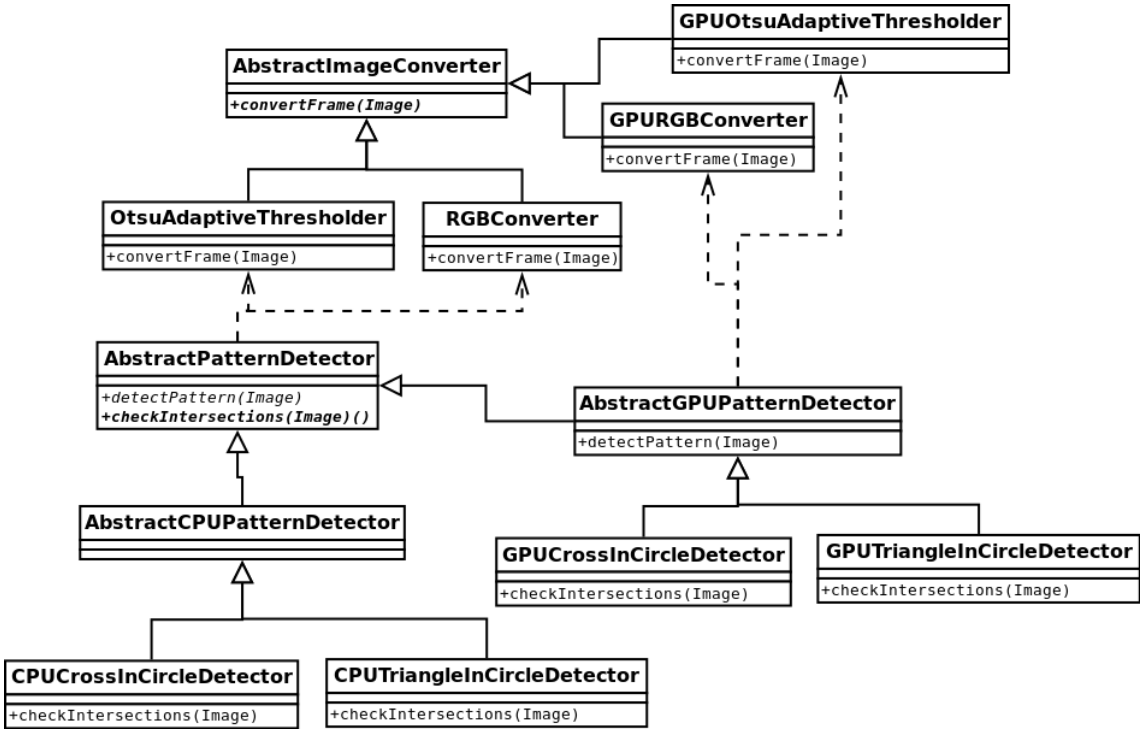

Figure 4.1. Object-orieted design of the application

4.1 Convert colored image to binary

To detect the pattern I have to first convert the initial image I have gotten from camera to binary image. This piece of code is responsible for it:

```
grayImage = convertToGrayscale(image)
blurredImage = gaussianBlur(grayImage)
binaryImage = convertToBinary(blurredImage)
```

Converting an image to grayscale image is the first step. Then a gaussian blur is applied to the image to reduce noise and detail [16]. I had to deal with a low-quality print of the patterns and I had touch them up with a black whiteboard marker, so the edges are not perfect. Then the Otsu's method [17] is applied to convert the blurred image to binary image. All three functions can found implemented in the OpenCV.

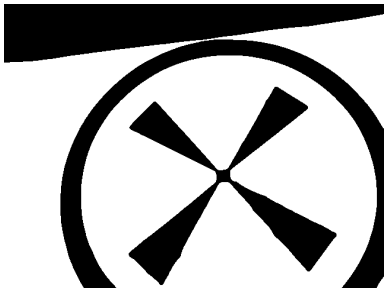

Figure 4.2. Image converted by Otsu's method

4.1.1 CUDA Implementation

OpenCV has CUDA implementation of conversion to grayscale image and gaussian filter but it does not have the implementation of Otsu's method. Only constant thresholder is available. I have repurposed an implementation [18] of Otsu's method to be run on GPU. Otsu's method consist of three steps. First a pixel intensity histogram computation of an image that is to be converted. OpenCV has CUDA implementation of histogram calculation. Then a threshold has to be computed. This step can be done using CPU due to its low computational demands. Threshold that I gained with this step was used as an threshold argument for OpenCV's CUDA implementation of constant thresholder.

4.2 Ellipse detection

The next step I have to do is to detect ellipses in the image. This is achieved using this part of the code:

```
contours = findContours(binaryImage)
ellipses[] = detectEllipses(contours)
```

First contours have to be detected using findContours function in OpenCV library [19]. Contour is a set of points which are around shape's perimeter. Then a function fitEllipse [20] is called on one contour at the time to fit an ellipse around these contours. These functions do not have a CUDA implementation and are going to be left without one. Because fit ellipse does not check whether an contour is an ellipse, I am checking every's ellipse corresponding contour whether said contours approximately same area as the ellipse.

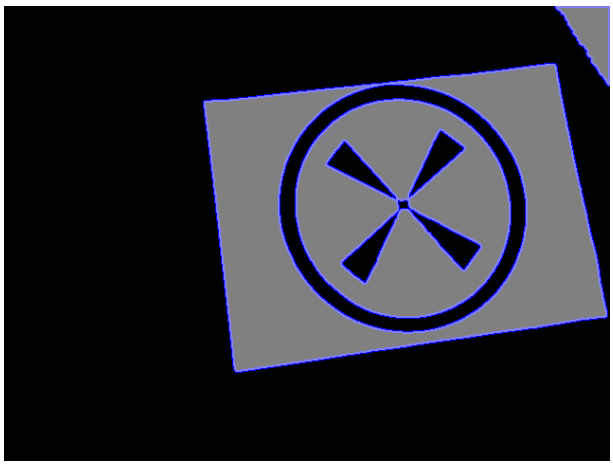

Figure 4.3. Binary image with contours highlighted in blue

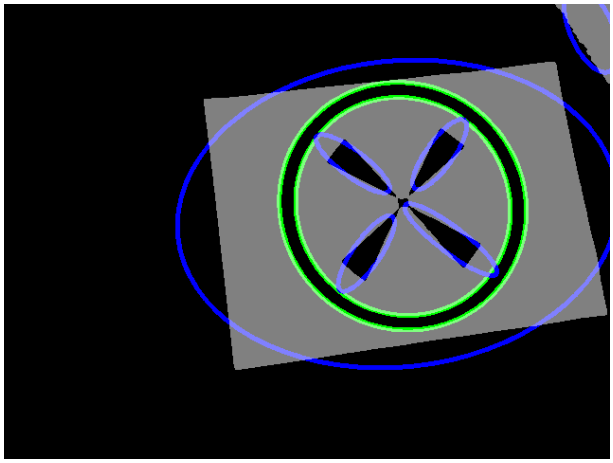

Figure 4.4. Binary image with correct ellipses highlighted in green

4.3 Line segment detection

Line segment detection is the next step to succesful pattern detection.

```
edges = cannyEdges(binaryImage)
lineSegments = houghSegments(edges)
```

First I apply to a binary image a canny edge detector [21]. It detects edges in a grayscale image. Similar to contours but instead of points this function returns an image. Canny edge detector is implemented in Open CV. Then I use probabilistic hough line transform [22] to detect line segments in the output of canny edge detector. Unlike fitEllipse this function differentiates between edges that are line segments and edges that are not. Probabilist hough transform has an implementation in OpenCV's library under name houghLinesP.

This step is first applied to cropped images figure 4.5, figure 4.6 and then the whole images figure 4.7, figure 4.8 if a pattern is not detected inside an ellipse.

4.3.1 CUDA implementation

OpenCV has a CUDA implementation for both hough line segment and canny edge detector functions. I used in an effort to speed up the detection method.

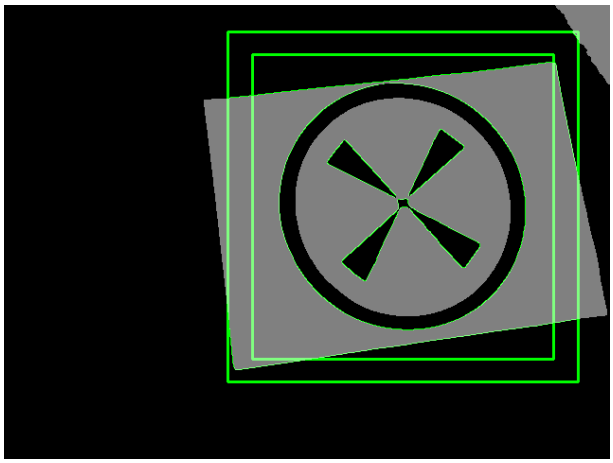

Figure 4.5. Binary image with cropped areas which are inside the rectangles and highlighted edges

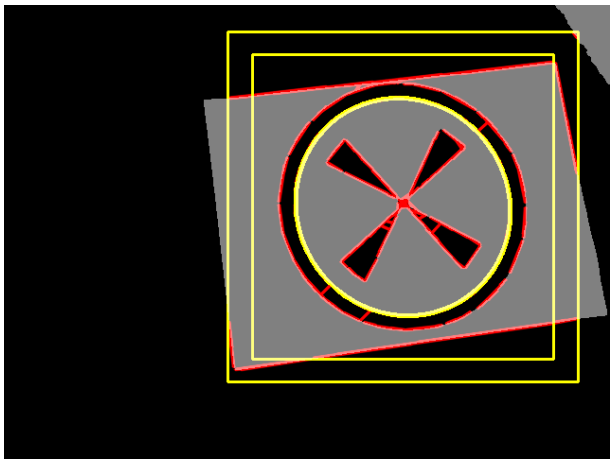

Figure 4.6. Binary image with cropped areas which are inside the rectangles and highlighted line segments in red

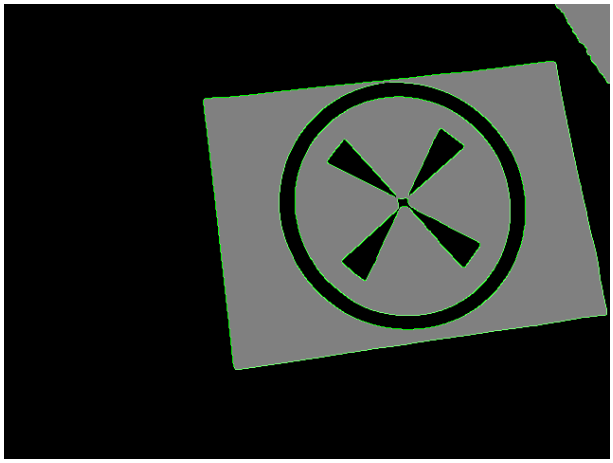

Figure 4.7. Binary image with highlighted edges

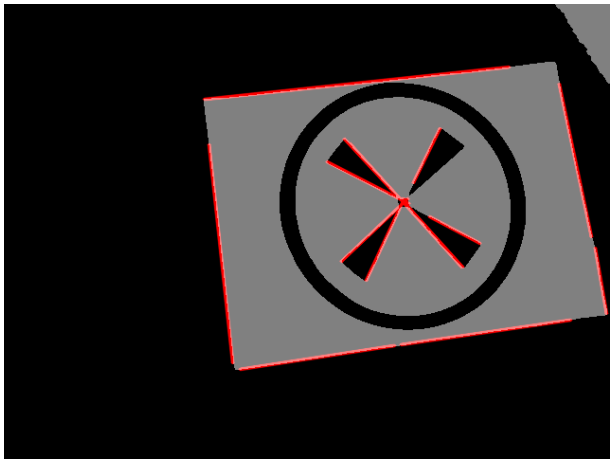

Figure 4.8. Binary image with highlighted line segments in red

4.4 Instersection detection

This is the last before final pattern detection is performed.

```
joinedLines = joinLines(lineSegments)
intersections = findIntersection(joinedLines)
```

I implemented these two functions myself. The first takes lines segments detected by hough line transform and joins ones that lie on the same line and returns these newly created line segments. The point is that segments detected by hough transform never form an intersection. But these newly created lines can. The patterns are designed to exploit this functionality and allow for a relatively robust detection. After creating these lines a search for their intersections is performed and these are after used for pattern detection. As with the previous function these functions first get into contact with cropped images figure 4.9, figure 4.10 and then with the full images figure 4.11, figure 4.12 when a pattern is not detected within an ellipse. These functions are only ran on CPU, because hough lines segment detector returns low amout of segments most of the time.

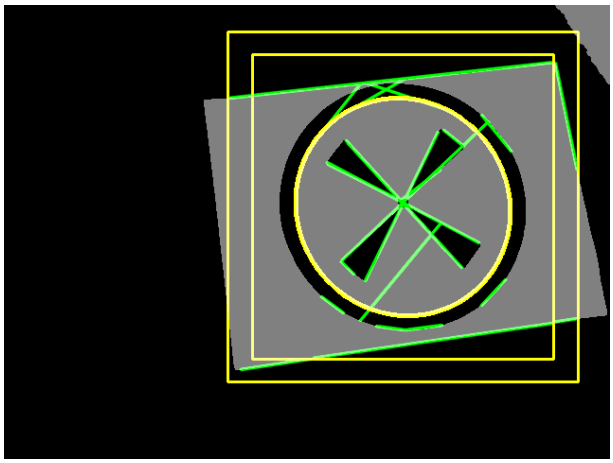

Figure 4.9. Binary image with cropped areas which are inside the rectangles and highlighted joined line segments

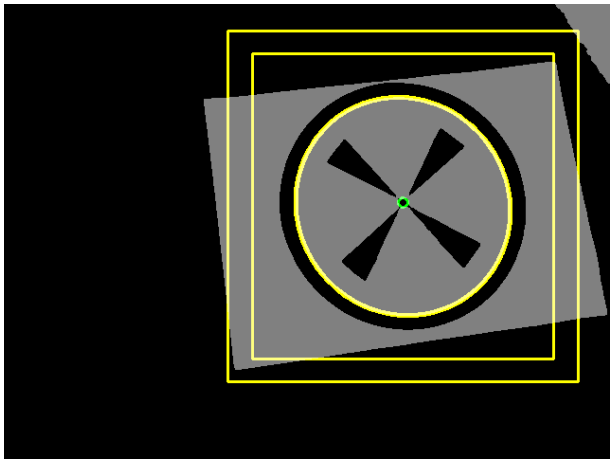

Figure 4.10. Binary image with cropped areas which are inside the rectangles and highlighted intersection of joined lines

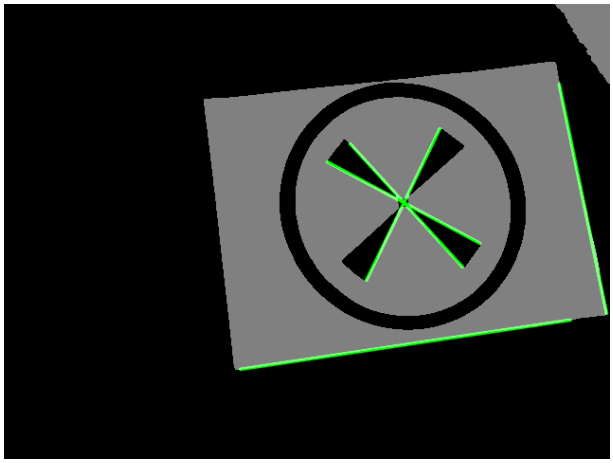

Figure 4.11. Binary image with cropped areas which are inside the rectangles and highlighted joined line segments

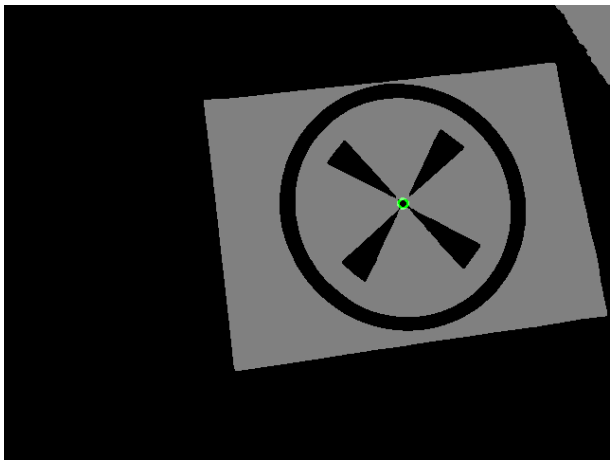

Figure 4.12. Binary image with highlighted intersection of joined lines

4.5 Pattern Detection

This is the code that handles the final step of detections.

```
patternFound, patternPosition = detectShape(intersections)

if patternFound do:
return patternPosition
end
```

4.5.1 Symmetrical Pattern

Cross in a circle is detected when at least 4 intersections ale located very close to each other. When the detection is performed inside an ellipse there is also an extra requirement that all the intersections must lie in the centre of the ellipse.

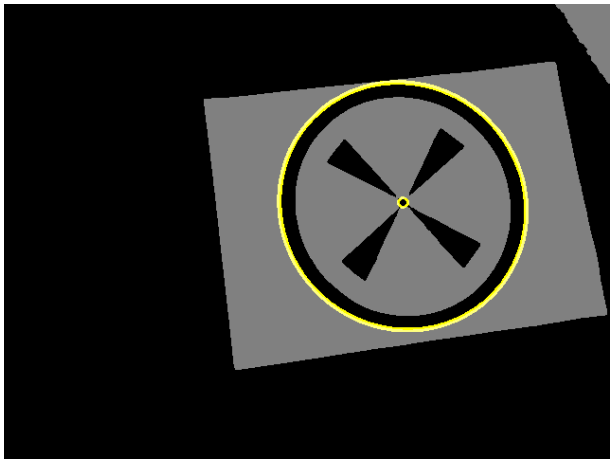

Figure 4.13. Pattern detected inside an ellipse, yellow circle shows its center

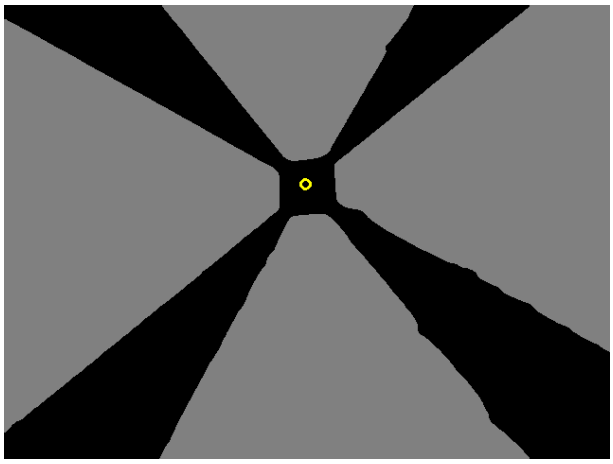

Figure 4.14. Pattern detected, yellow circle shows its center

4.5.2 Asymmetrical Pattern

Triangles in a circle are detected when three intersections are forming a rough isosceles triangle. One edge of this triangle has to close to half the length of other edges. Vertex between two sides of the same length lies in the centre of the ellipse. Intersection form at vertex of triangles.

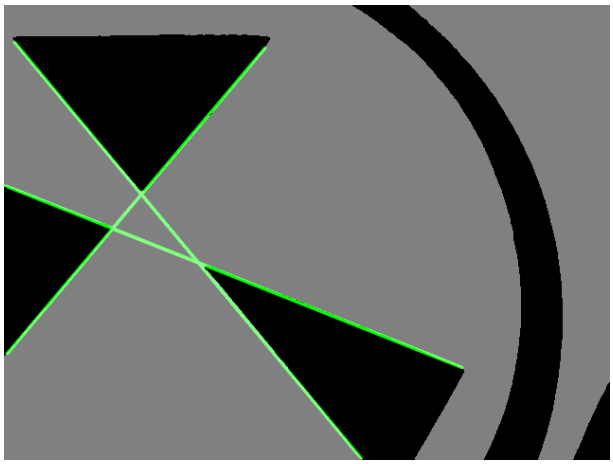

Figure 4.15. Intersection at the verteces of the triangles

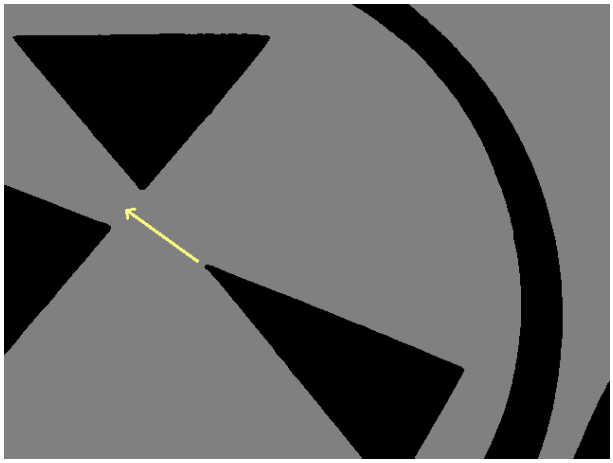

Figure 4.16. Asymmetrical pattern detected, yellow arrow shows its oriantation

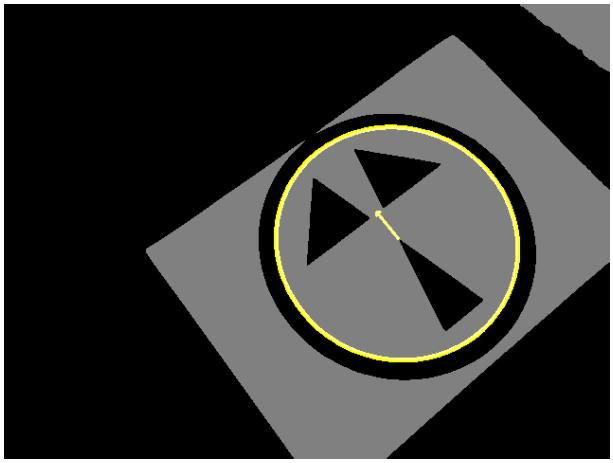

Figure 4.17. Asymmetrical pattern detected in ellipse, yellow arrow shows its oriantation, arrow begins at the center of the ellipse

Chapter 5
System overview

The code was tested on my laptop here are my specifications:

```
Msi GE 70 2PE ApachePro laptop
Intel® Core i5-4200H CPU @ 2.80GHz ■ 4
8 GB RAM
Ubuntu 18.04.2 LTS

Linux x64 (AMD64/EM64T) Display Driver
Version:  430.14
Driver Release Date:  2019.5.14
Operating System:  Linux 64-bit


Number of CUDA devices 1.
There is 1 device supporting CUDA
For device #0
Device name:                GeForce GTX 860M
Major revision number:      5
Minor revision Number:      0
Total Global Memory:        2101870592
Total shared mem per block: 49152
Total const mem size:       65536
Warp size:                  32
Maximum block dimensions:   1024 x 1024 x 64
Maximum grid dimensions:    2147483647 x 65535 x 65535
Clock Rate:                 1019500
Number of muliprocessors:   5
```

Images were acquired with:

```
Logitech HD Pro Webcam C910
Xiaomi Redmi Note 4 64GB
```

Chapter 6
Testing

In this part I will examine the performace of implemented method, its weakness and unexpected behavior. I will compare between performan between CPU only method and GPU accelerated method and their components. All tests were done in terminal mode in Ubuntu with lightdm service stopped.

6.1 Symmetrical patterm

6.1.1 Landing pattern far

The input data for both tables are the same.

Figure 6.1. Input data - Img 1

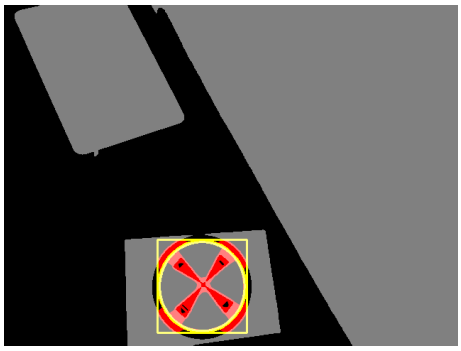

Figure 6.2. Input data - Img 1 outlier

In table 6.2 are times for Total on CPU + GPU very high. Its due some bug in opencv that in terminal caused massive delays when calling gpuMat's release function which was measured by this item. There is also an outlier in the same in the first row.

Hough segment detector running on CUDA failed and classified too many lines. I had problem with this implementation during debugging.

It seems like gaussian blur on GPU is slower than its CPU counterpart. Hough transform in the Img 2-5 and RGB to Grayscale conversion were a lot faster than its CPU counterparts. Also no detections on CPU's side. Contour detection was struggling on GPU's side.

CPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on CPU	1.85	1.86	2.13	1.94	1.82
Gaussian blur on CPU	0.36	0.31	0.31	0.33	0.31
Otsu's binarization on CPU	0.4	0.37	0.37	0.36	0.37
Contour detection on CPU	0.47	0.46	0.46	0.47	0.45
Ellipse detection on CPU	0.44	0.4	0.39	0.37	0.4
Canny on CPU	0.73	0.76	0.85	0.75	0.82
Hough transform on CPU	3.48	3.62	3.7	3.58	4.65
Line detection on CPU	0.22	0.15	0.26	0.13	0.37
Cross validation on CPU	0	0	0	0	0
Total on CPU	8.49	8.5	9.09	8.52	9.86
Detected	0	0	0	0	0

Table 6.1. Symmetrical pattern far on CPU

GPU in ms	Img 1	Img 2	Img 3	Img 4	Img 5
RGB to Grayscale conversion on GPU	0.26	0.27	0.25	0.25	0.25
Gaussian blur on GPU	0.35	0.37	0.37	0.36	0.36
Otsu's binarization on GPU	0.23	0.22	0.21	0.21	0.21
Contour detection on CPU	1.98	1.88	2.01	1.97	1.97
Ellipse detection on CPU	0.42	0.41	0.4	0.38	0.38
Canny on GPU	2.02	0.32	0.33	0.42	0.42
Hough transform on GPU	1.43	0.45	0.41	0.44	0.44
Line detection on CPU	17.53	18.95	6.2	9.02	9.02
Cross validation on CPU	553.52	29.41	0	1.81	1.81
Total on CPU + GPU	775.91	603.64	561.68	567.64	567.64
Detected	0	1	1	1	1

Table 6.2. Symmetrical pattern far on GPU

6.1.2 Landing pattern midpoint

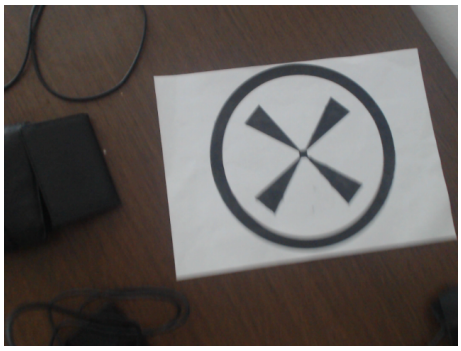

Figure 6.3. Input data - Img 1

Again hough segment detector on GPU failed. There is a minor speed increase in Otsu's GPU Implementation. Again same bug with the release function, so the total times on GPU side are hardly representative.

CPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on CPU	1.9	1.88	1.9	1.83	1.88
Gaussian blur on CPU	0.31	0.32	0.33	0.32	0.33
Otsu's binarization on CPU	0.37	0.36	0.35	0.36	0.36
Contour detection on CPU	0.46	0.44	0.46	0.46	0.46
Ellipse detection on CPU	0.39	0.33	0.39	0.36	0.37
Canny on CPU	0.39	0.3	1.59	0.26	0.26
Hough transform on CPU	2.4	2.11	3.17	1.61	1.7
Line detection on CPU	0.04	0.04	0.36	0.09	0.06
Cross validation on CPU	0	0	0	0	0
Total on CPU	6.87	6.39	7.96	5.9	6.02
Detected	1	1	1	1	1

Table 6.3. Symmetrical pattern midpoint on CPU

GPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on GPU	0.26	0.27	0.26	0.25	0.25
Gaussian blur on GPU	0.36	0.35	0.36	0.35	0.37
Otsu's binarization on GPU	0.21	0.21	0.22	0.21	0.21
Contour detection on CPU	1.92	2.04	1.99	2	2.02
Ellipse detection on CPU	0.4	0.33	0.41	0.36	0.39
Canny on GPU	2.07	0.88	0.4	0.61	0.64
Hough transform on GPU	1.21	0.56	0.46	0.64	0.63
Line detection on CPU	0.39	0.12	2.46	3.07	2.59
Cross validation on CPU	0	0	0	0	0
Total on CPU + GPU	555.75	557.88	559.433	561.01	599.84
Detected	1	1	1	1	1

Table 6.4. Symmetrical pattern midpoint on GPU

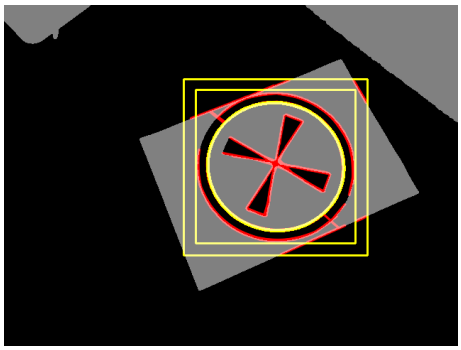

Figure 6.4. Input data - Img 4 outlier

6.1.3 Landing pattern close

Figure 6.5. Input data - Img 1

CPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on CPU	1.83	1.83	1.86	1.87	1.84
Gaussian blur on CPU	0.3	0.3	0.3	0.33	0.32
Otsu's binarization on CPU	0.38	0.4	0.39	0.42	0.44
Contour detection on CPU	0.5	0.51	0.43	0.44	0.45
Ellipse detection on CPU	0.48	0.56	0.39	0.37	0.36
Canny on CPU	0.67	0.45	0.63	0.68	0.6
Hough transform on CPU	3.64	2.65	2.81	2.48	2.41
Line detection on CPU	0.04	0.07	0.03	0.02	0.03
Cross validation on CPU	0	0	0	0	0
Total on CPU	8.41	7.39	7.38	7.19	7.05
Detected	0	1	0	1	1

Table 6.5. Symmetrical pattern close on CPU

GPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on GPU	0.26	0.27	0.26	0.25	0.25
Gaussian blur on GPU	0.36	0.35	0.36	0.35	0.37
Otsu's binarization on GPU	0.21	0.21	0.22	0.21	0.21
Contour detection on CPU	1.92	2.04	1.99	2	2.02
Ellipse detection on CPU	0.4	0.33	0.41	0.36	0.39
Canny on GPU	2.07	0.88	0.4	0.61	0.64
Hough transform on GPU	1.21	0.56	0.46	0.64	0.63
Line detection on CPU	0.39	0.12	2.46	3.07	2.59
Cross validation on CPU	0	0	0	0	0
Total on CPU + GPU	555.75	557.88	559.433	561.01	599.84
Detected	0	1	0	1	1

Table 6.6. Symmetrical pattern close on GPU

6.2 Asymmetrical patterm

6.2.1 Landing pattern far

Figure 6.6. Input data - Img 1

CPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on CPU	1.84	1.86	1.91	1.97	1.83
Gaussian blur on CPU	0.32	0.3	0.32	0.31	0.31
Otsu's binarization on CPU	0.36	0.37	0.37	0.36	0.36
Contour detection on CPU	0.44	0.45	0.45	0.43	0.45
Ellipse detection on CPU	0.34	0.37	0.41	0.33	0.35
Canny on CPU	0.25	0.26	0.25	0.25	0.24
Hough transform on CPU	1.57	1.62	1.51	1.37	1.34
Line detection on CPU	0.17	0.23	0.12	0.08	0.11
Triangle validation on CPU	0	0	0	0	0
Total on CPU	5.91	6.08	5.94	5.67	5.58
Detected	1	1	1	1	1

Table 6.7. Asymmetrical pattern far CPU

GPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on GPU	0.25	0.29	0.25	0.25	0.26
Gaussian blur on GPU	0.35	0.38	0.35	0.35	0.35
Otsu's binarization on GPU	0.21	0.23	0.21	0.21	0.21
Contour detection on CPU	1.98	2.09	1.98	1.98	2
Ellipse detection on CPU	0.34	0.37	0.38	0.34	0.34
Canny on GPU	0.38	0.44	0.43	0.38	2.16
Hough transform on GPU	0.45	0.62	0.64	0.46	1.48
Line detection on CPU	4.89	4.17	6.01	5.37	5.98
Triangle validation on CPU	0	0	0.02	0.03	0
Total on CPU + GPU	558.19	561.67	559.8	559.14	564.9
Detected	1	1	1	1	0

Table 6.8. Asymmetrical pattern far GPU

6.2.2 Landing pattern midpoint

Figure 6.7. Input data - Img 1

CPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on CPU	1.87	1.84	1.85	1.84	1.85
Gaussian blur on CPU	0.34	0.31	0.32	0.3	0.31
Otsu's binarization on CPU	0.41	0.35	0.36	0.39	0.37
Contour detection on CPU	0.46	0.47	0.47	0.47	0.42
Ellipse detection on CPU	0.4	0.35	0.4	0.37	0.34
Canny on CPU	0.69	0.32	0.36	0.46	0.27
Hough transform on CPU	2.9	2.19	1.76	2.4	1.68
Line detection on CPU	0.03	0.04	0.16	0.04	0.06
Triangle validation on CPU	0	0	0	0	0
Total on CPU	7.65	6.52	6.27	6.89	5.91
Detected	0	1	1	1	1

Table 6.9. Asymmetrical pattern midpoint CPU

GPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on GPU	0.26	0.26	0.26	0.26	0.26
Gaussian blur on GPU	0.36	0.35	0.35	0.35	0.36
Otsu's binarization on GPU	0.21	0.22	0.22	0.21	0.21
Contour detection on CPU	1.96	2.04	2.05	2	1.96
Ellipse detection on CPU	0.35	0.36	0.37	0.37	0.35
Canny on GPU	1.89	0.89	2.34	1.23	0.54
Hough transform on GPU	1.04	0.6	1.5	0.59	0.49
Line detection on CPU	0.07	0.19	0.44	0.07	1.97
Triangle validation on CPU	0	0	0	0	0
Total on CPU + GPU	556.15	555.72	561.25	555.37	560.925
Detected	1	1	0	1	1

Table 6.10. Asymmetrical pattern midpoint GPU

6.2.3 Landing pattern close

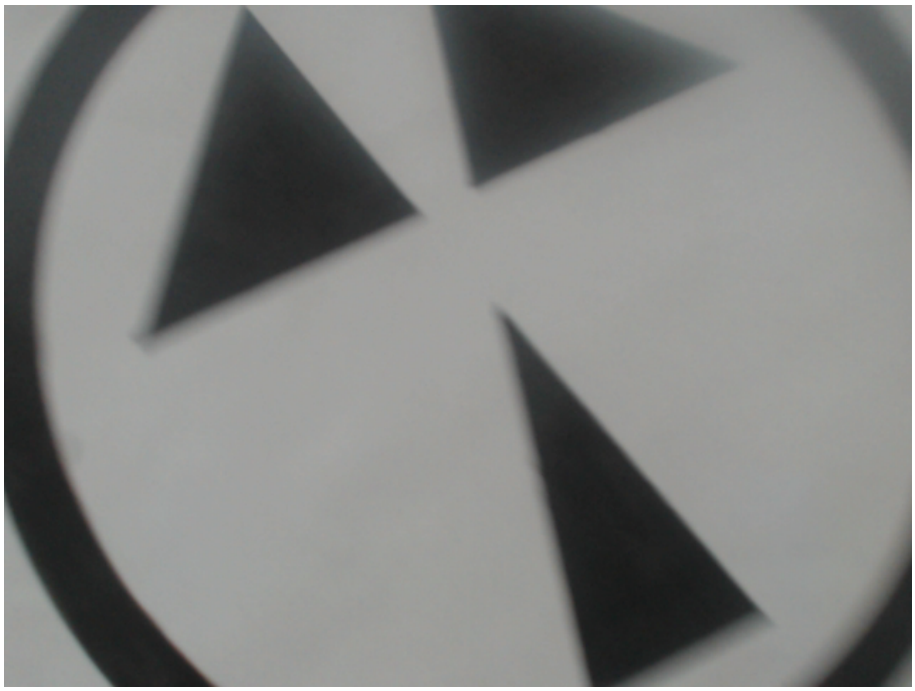

Figure 6.8. Input data - Img 1

CPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on CPU	1.82	1.83	1.86	1.86	1.85
Gaussian blur on CPU	0.31	0.31	0.33	0.32	0.32
Otsu's binarization on CPU	0.4	0.42	0.4	0.4	0.4
Contour detection on CPU	0.47	0.44	0.46	0.47	0.48
Ellipse detection on CPU	0.39	0.31	0.41	0.42	0.43
Canny on CPU	0.65	0.76	0.65	0.65	0.64
Hough transform on CPU	2.71	2.44	3.01	2.97	2.84
Line detection on CPU	0.03	0.04	0.04	0.06	0.05
Triangle validation on CPU	0	0	0	0	0
Total on CPU	7.36	7.08	7.7	7.71	7.54
Detected	1	1	1	1	1

Table 6.11. Asymmetrical pattern close CPU

GPU in ms	Img 1	Img 2	Img 3	Img 4	img5
RGB to Grayscale conversion on GPU	0.26	0.25	0.25	0.25	0.26
Gaussian blur on GPU	0.37	0.35	0.35	0.35	0.36
Otsu's binarization on GPU	0.22	0.22	0.21	0.21	0.22
Contour detection on CPU	2.01	1.93	1.9	2.06	1.95
Ellipse detection on CPU	0.4	0.43	0.44	0.41	0.313
Canny on GPU	1.91	1.87	1.87	1.88	1.86
Hough transform on GPU	1.03	1.06	1.06	1.07	1.04
Line detection on CPU	0.06	0.08	0.1	0.09	0.05
Triangle validation on CPU	0	0	0	0	0
Total on CPU + GPU	563.877	554.73	557.68	554.25	556.38
Detected	1	0	1	1	0

Table 6.12. Asymmetrical pattern close GPU

Chapter 7
Conclusion

I have designed two patterns and implemented an application for a vision based detection. Patters are designed in such a way, that they can resized without need of larger modification to the app. I have modified the to use a Nvidia CPU on my laptop to speed up detection process. While some of the method were good idea, some of the method performed worse than their CPU counterparts. I would reccomend retesting this on newer graphics card.

With the system I had CPU only approch takes about 5-10 miliseconds to process the frames sized 640x480. GPU proved inconclusive since there was a problem that was beyond my reach.

For future I would recommend putting into a on-board o a UAV and I would try to expand this even further and optimizing further the CUDA support.

References

[1] 3d Insider. *Long battery life drones*. 2019.
https://3dinsider.com/long-flight-time-drones/.

[2] Skysense. *Autonomous Charging In Outdoor Environment*. 2019.
https://www.skysense.co/charging-pad-outdoor.

[3] FUJII Katsuya; Keita HIGUCHI; Jun REKIMOTO. *Endless Flyer: A Continuous Flying Drone with Automatic Battery Replacement.* 2013.
http://ieeexplore.ieee.org/document/6726212/.

[4] Francesco COCCHIONI; Emanuele FRONTONI; Gianluca IPPOLITI; Sauro LONGHI; Adriano MANCINI; Primo ZINGARETTI. *Visual Based Landing for an Unmanned Quadrotor.* 2016.
http://link.springer.com/10.1007/s10846-015-0271-6.

[5] OpenCV. 2019.
https://opencv.org/.

[6] Thiang I.N.; LuMaw H.M.T. *Vision-based object tracking algorithm with ar. drone.* 2016.

[7] D. Barták R.; Hrasko A.; Obdrzálek. *On Autonomous Landing of AR. Drone: Hands-On Experience*. 2014.

[8] Feng Y.; Zhang C.; Baek S.; Rawashdeh S.; Mohammadi A. *Autonomous Landing of a UAV on a Moving Platform Using Model Predictive Control.* 2018.

[9] *April tags*. 2019.
https://april.eecs.umich.edu/software/apriltag.

[10] ARAAR Oualid; Nabil AOUF; Ivan VITANOV. *Vision Based Autonomous Landing of Multirotor UAV on Moving Platform.* 2017.
http://link.springer.com/10.1007/s10846-016-0399-z.

[11] D. Falanga; A. Zanchettin; A. Simovic; J. Delmerico; D. Scaramuzza. *Vision-based Autonomous Quadrotor Landing on a Moving Platform.* 2017.

[12] T Baca; P Stepan; M Saska. *Autonomous Landing On A Moving Car With Unmanned Aerial Vehicle.* 2017.

[13] NVidia. 2019.
https://www.nvidia.com/en-us/.

[14] NVidia. *CUDA C Programming Guide*.
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[15] Martin Heller. *What is CUDA? Parallel programming for GPUs*. 2018.
https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html.

[16] *Gaussian blur*.
https://en.wikipedia.org/wiki/Gaussian_blur.

[17] Nobuyuki OTSU. *A Threshold Selection Method from Gray-Level Histograms. IEEE Transactions on Systems, Man, and Cybernetics*. 1979. http://ieeexplore.ieee.org/document/4310076/.

[18] *Otsu Thresholding*. http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html.

[19] Satoshi Suzuki, and others. *Topological structural analysis of digitized binary images by border following. Computer Vision, Graphics, and Image Processing, 30(1):32–46*. 1985.

[20] Andrew W Fitzgibbon, and Robert B Fisher. *A buyer's guide to conic fitting.* 1995.

[21] J. Canny. *A Computational Approach To Edge Detection*. 1986.

[22] J. Matas, C. Galambos, and J.V. Kittler. *Robust Detection of Lines Using the Progressive Probabilistic Hough Transform.* 2000.

Appendix A
DVD

/app - Source codes for app
  /app/images - Testing data
  /patterns - Patterns used in this thesis
  /thesis/ctustyle2 - sources for this document
  /thesis/ctustyle2/images - images used in this document