KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

Real world applications of Artificial Intelligence on constrained hardware

Wiebe Van Ranst

Supervisors:
Prof. dr. J. Vennekens
Prof. dr. ir. T. Goedemé

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

January 2019

Real world applications of Artificial Intelligence on constrained hardware

Wiebe VAN RANST

Examination committee:
Prof. dr. ir. M. Versteyhe, chair
Prof. dr. J. Vennekens, supervisor
Prof. dr. ir. T. Goedemé, supervisor
Prof. dr. ir. G. De Samblanx
Prof. dr. ir. M. Verhelst
Prof. dr. ir. J. Lemeire
(Vrije Universiteit Brussel (VUB))
Dr. ing. F. De Smedt
(RoboVision)
Prof. dr. ir. S. Gong
(Queen Mary University of London)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

January 2019

© 2019 KU Leuven – Faculty of Engineering Technology
Uitgegeven in eigen beheer, Wiebe Van Ranst, Jan De Nayerlaan 5, B-2860 Sint-Katelijne-Waver, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

So I guess this is the part where you look back and reflect on your past years working on a PhD. Not to sound too cliche, but I don't think I'm really your typical PhD student. You see, doing a PhD was never really a goal of mine. Getting to this academic level would have been laughable to me just 10 years ago. Back then, in high school, I was never really a good student. Progressing academically was not really of that much importance to me back then. What, I think, did keep me going, and what I think is a big part of the reason I am here right now, is that I did always have some passions. Passions I'm able to get quite obsessed about up to a, sometimes unhealthy, point. Computers and programming has always been one of those. Later, long distance running became another. I don't really think I've ever pursued one of those things with a clear goal in mind. I always just liked doing things that I liked and were stimulating to me, without having to be the best, or achieving a clear goal. Loving the act of doing what I did was what I was good at. Although I wasn't always convinced of it, which I can imagine must have been frustrating at times for my promoter Joost Vennekens, looking back at the past years, I'm happy and thankful that I went the way of doing a PhD. Working on my own projects, presenting my research on conferences (which was never a problem to attend!) has allowed me to grow as a person, it forced me to put myself out there to be independent. At the same time working closely together with colleagues and doing research in collaboration has also taught me a lot about working with other people. Overall, I learned a lot, and while I might not have always seen the value in doing a PhD, seeing where I am now, I certainly do.

Thank you, Joost, for asking me if I wanted to do a PhD! Without you I certainly wouldn't be here right now. Thank you for being willing to see it through, even after at some point, I wasn't willing to do the same. Thank you for acting as my supervisor throughout my PhD, especially at the start. Thanks also to my co-promoter Toon Goedemé, for always having some interesting project ready, for willing to provide funding and supervising me in the last years of my PhD. Thank you both for guiding me in writing my thesis, for providing funding and

allowing me to present my work on many international conferences.

Thanks to all members of the examination committee for giving me advice and for taking the time to read and evaluate this thesis.

Also thanks to my colleagues for all the interesting discussions and fun times we've had together.

I would also like to thank my teachers and professors throughout my carrier as a student for fostering such a stimulating environment to work and learn in.

Thanks to all my friends for being there when I needed it, for all the fun times we spend together and various trips we took during my PhD. You offered me a much-needed relief and change of mind.

Bedankt ook aan mijn familie en in het bijzonder Mama en Papa om mij de kans te geven om verder te studeren, en om mij te ondersteunen tijdens mijn studies. Zonder jullie was ik nooit zo ver geraakt.

And lastly I would also like to thank my girlfriend Aline. Thanks for supporting me in the last years of my PhD and for proofreading my thesis. I'll do my best to do the same for you!

— March 2019, Sint-Katelijne-Waver, Belgium.

Abstract

These days the field of Artificial Intelligence (and its many subfields) is moving really fast, many new techniques are becoming available from various different subfields. However, many of these algorithms are only made to run on very powerful research workstations without considering how they can be used on real-world hardware, be it embedded hardware, powerful GPUs or even distributed systems.

In this PhD we try to bridge this gap. We study how different algorithms, from fields like artificial intelligence, computer vision or image processing, can be adapted to different hardware that is accompanied by some restrictions or/and programming challenges.

We look at hardware that is constrained in terms of resources (eg. embedded devices). The big challenge here is to make sure that the requirements of the algorithm, and the requirements of the contexts in which it is used (eg. being able to run in real time) are fulfilled using only these limited resources.

Other than that, we also look at hardware that uses different programming paradigms which often makes them challenging to program (eg. GPUs). While these kinds of hardware are often not limited in terms of resources (often the opposite is true), exotic architectures mean that using this hardware to its full potential remains a challenge.

In this thesis we examine these challenges based on many applications, and lay them out on top of a framework with constraints and trade-offs that outlines what they have in common and sets them apart.

Beknopte samenvatting

De wereld van de Artificiële Intelligentie (en vele van zijn subdomeinen) maakt dezer dagen zeer snel vooruitgang, veel nieuwe technieken worden beschikbaar gemaakt uit verschillende subdomeinen. Veel van deze algoritmes worden echter enkel ontwikkeld op krachtige onderzoek computersystemen, zonder er bij stil te staan hoe deze kunnen gebruikt worden op hardware die in de praktijk gebruikt wordt, bijvoorbeeld ingebedde systemen, krachtige GPUs of zelfs gedistribueerde systemen.

In dit doctoraat proberen we deze kloof te overbruggen. We gaan na hoe verschillende algoritmes, komende uit verschillende deeltakken van Artificiële Intelligentie, zoals probabilistisch logisch programmeren, computer visie, en beeldverwerking, kunnen worden aangepast aan verschillende types hardware die verschillende restricties en/of programmeer uitdagingen met zich teweegbrengen.

Aan de ene zijde bestuderen we hardware die beperkt is in rekenkracht (b.v.: ingebedde systemen). Waar de grote uitdaging ligt in het vervullen van de vereisten van het algoritme, en de context waarin het gebruikt wordt (b.v.: het moet uitgevoerd worden in “real-time”), op hardware die beperkingen heeft.

Aan de andere kant kijken we ook naar hardware die een andere programmeer paradigma’s gebruiken (b.v.: GPU’s) wat hen vaak ook uitdagend maakt om te programmeren. Hoewel deze hardware vaak niet gelimiteerd is in performantie (het tegendeel is vaak waar), betekenen exotische architecturen dat het gebruik van deze hardware een grote uitdaging blijft.

In dit doctoraat bestuderen we deze uitdagingen op verschillende toepassingen, en passen we ze toe op een raamwerk met restricties en afwegingen dat verduidelijkt wat de verschillende toepassingen gemeen hebben en wat ze verschillend maakt.

List of Abbreviations

ACF Aggregate Channel Features. 2, 14, 16, 17, 25, 75–79, 88, 90, 91, 99, 102–104, 107, 108, 110, 112–115, 119, 124–126, 129, 133, 138

AI Artificial Intelligence. 1–3, 33, 149, 156

ALU Arithmetic Logic Unit. 24

BGS background subtraction. 98

CNN Convolutional Neural Network. 133

CP-logic Causal Probabilistic logic. 2, 33–36, 38–40, 42, 44, 49, 53–56, 154, 157

CPU Central Processing Unit. 20–22, 48, 49, 55, 69, 71, 74, 78, 89, 104, 107, 108, 110, 113, 152

CUDA Compute Unified Device Architecture. 20, 23, 29, 106, 109, 110

CVE Contextual Variable Elimination. 52, 54

DPM Deformable Parts Model. 77, 78, 99, 102, 133, 138

FN false negative. 101

FP false positive. 101

FPGA Field Programmable Gate Array. 11, 150

FPU Floating Point Unit. 21, 24

GFLOPS Number of Giga floating-point operations that can be executed in one second.. 29, 30

GPC GPU Processing Cluster. 24

GPGPU General Purpose computation on the GPU. 20

GPU Graphical Processing Unit. 1, 2, 6, 13, 15–17, 19–25, 27–29, 31–33, 35, 47, 49, 51, 52, 54–57, 68, 69, 71, 73–79, 88, 90, 91, 104–106, 110–116, 118, 119, 124, 125, 129, 150, 152, 156–158, 177

HOG Histogram of Oriented Gradients. 77, 78

HSV hue, saturation, value. 85, 86, 88, 91

ICF Integral Channel Features. 75, 77, 78

IL interest level. 97, 98

KLT Kanade-Lucas-Tomasi feature tracker. 64, 66, 68, 69, 74

LUV Luminosity + UV-color coordinates. 78, 79, 104

mAP Mean Average Precision. 14

ML Machine Learning. 34

MSER Maximally Stable Extremal Regions. 86, 88, 93

ORB Oriented FAST and Rotated BRIEF. 158

PITA Probabilistic Inference with Tabling and Answer subsumption. 34, 50, 52, 54

PLL Probabilistic Logic Learning. 16, 34, 53, 54, 156, 157

PLP Probabilistic Logic Programming. 56, 57

PRISM PRogramming In Statistical Modeling. 34

PTZ pan-tilt-zoom. 94, 96–99, 101, 103, 104, 113, 151, 152

RGB Red Green Blue. 78, 82, 83, 85, 91, 93

SDI Serial Digital Interface. 97

SIFT Scale-Invariant Feature Transform. 61, 158

SM Streaming Multiprocessor. 24, 25

SoC System on a Chip. 29

SURF Speeded Up Robust Features. 61, 97, 158

SVM Support Vector Machine. 78, 89

TPC Texture Processing Cluster. 24

TPU Tensor Processing Unit. 27

UAV Unmanned Areal Vehicle. 16, 113–116, 155

YOLO You Only Look Once. 16, 114, 124, 131, 139, 140, 144, 146, 147, 159

Contents

Abstract iii

Beknopte samenvatting v

List of Abbreviations ix

List of Symbols xi

Contents xi

List of Figures xvii

List of Tables xxiii

1 Introduction 1

1.1 What do we mean by Artificial Intelligence? 2

1.2 What are these constraints? 2

1.2.1 Speed hardware trade-off 5

1.2.2 Hardware accuracy trade-off 11

1.2.3 Speed accuracy trade-off 14

1.3 Contributions 15

1.3.1 Look forward 17

2 Hardware and frameworks **19**

2.1 GPU Computing 20

2.1.1 Using a Graphical Processing Unit (GPU) for general purpose computation 21

2.1.2 GPU Architecture 23

2.1.3 Deep Learning frameworks 25

2.1.4 Conclusion 28

2.2 Embedded platforms 28

2.2.1 Freescale iMX6 28

2.2.2 NVIDIA Jetson Embedded computing board 29

2.2.3 Power consumption vs. computing power 29

2.3 Conclusion 31

3 An OpenCL implementation of a forward sampling algorithm for CP-logic **33**

3.1 Introduction 33

3.2 Preliminaries from Logic Programming 36

3.3 Preliminaries: CP-logic 38

3.4 A forward sampling method for positive CP-theories 40

3.5 A forward sampling method for general CP-theories 43

3.6 OpenCL implementation 45

3.6.1 Our algorithm in OpenCL 45

3.7 Experimental results 47

3.8 Discussion and future work 53

3.9 Conclusions 56

4 Ultra-low-latency automatic endoscopic image orientation stabilisation **59**

4.1 Introduction 60

4.2 Rotation compensation 61
4.2.1 Detecting Corners 65
4.2.2 Tracking 66
4.2.3 Calculate Transformation 66
4.2.4 Interpolation of Estimated Angle 66
4.3 Implementation 67
4.3.1 Using only the Receiver Module (REC) 68
4.3.2 Using both the Receiver and the Source Module (R&S) 69
4.3.3 Compute Server (SRV) 69
4.4 Results 70
4.4.1 Prototype 70
4.4.2 Performance 71
4.5 Conclusion 74

5 GPU implementation of ACF Algorithm 75

5.1 Introduction 76
5.2 Related Work 77
5.3 ACF Person detector 78
5.4 Applications without the use of GPU computation 79
5.5 Application 1: Detection of photovoltaic installations in RGB aerial imaging: a comparative study 80
5.5.1 Introduction 81
5.5.2 Related Work 82
5.5.3 Datasets 83
5.5.4 Suggested Approach 85
5.5.5 Pixel-based colour classification using support vector machines 85
5.5.6 MSER based colour segmentation and shape analysis . . 85

5.5.7 Boosted cascade of weak classifiers 87
5.5.8 Aggregate channel features 88
5.5.9 Results 88
5.5.10 Discussion and conclusion 91
5.6 Application 2: The autonomous hidden camera crew 94
5.6.1 Introduction 94
5.6.2 Related work 95
5.6.3 Hardware setup 96
5.6.4 Camera selection 97
5.6.5 Autonomous PTZ steering 99
5.6.6 Experimental results 101
5.6.7 Conclusion 103
5.7 GPU implementation 104
5.7.1 Feature calculation 104
5.7.2 Model evaluation 105
5.8 GPU implementation speed results 110
5.9 Detector comparison 110
5.10 Application 3: Embedded Real-Time Object Detection for a UAV Warning System 113
5.10.1 Introduction 114
5.10.2 Approach 115
5.10.3 Related work 117
5.10.4 Architecture 118
5.10.5 Evaluation 120
5.10.6 Conclusion 126
5.11 Conclusion 129

6 Fast Simultaneous People Detection and Re-identification in a Single Shot Network 131

6.1 Introduction . . . 131
6.2 Related Work . . . 137
6.3 YOLO-REID . . . 139
6.3.1 Network architecture . . . 139
6.3.2 Triplet loss . . . 141
6.3.3 Multi target training . . . 141
6.4 Person Tracking . . . 143
6.5 Results . . . 144
6.5.1 Detection . . . 144
6.5.2 Person search . . . 144
6.5.3 Computational complexity . . . 145
6.5.4 Person tracking . . . 146
6.6 Conclusion . . . 146

7 Conclusion 149

7.1 Runtime compute trade-off . . . 152
7.2 Hardware accuracy trade-off . . . 154
7.3 Speed accuracy trade-off . . . 157
7.4 Future work . . . 158
7.4.1 Using the GPU for Probabilistic Logic Programming . . . 158
7.4.2 Image processing at ultra low latency . . . 159
7.4.3 Object detection . . . 160
7.4.4 Re-identification . . . 160
7.5 Vision on the future of artificial intelligence in real-world applications . . . 162

Bibliography **165**

About the candidate **181**

List of publications **183**

List of Figures

1.1 Venn diagram showing how the different fields of Artificial Intelligence (AI) discussed in this thesis relate to each other, and where each chapter lies inside the diagram. 3

1.2 Triangle depicting our framework of three different constraints. 4

1.3 Visualization of the runtime-compute trade-off, we cannot change anything about the accuracy. 5

1.4 Execution flow of the parallel sum algorithm. 8

1.5 Comparison of different speedups in function of the number of processors of algorithms exhibiting a different parallel portion. Credit: Daniels220 at English Wikipedia [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons 9

1.6 Visualization of the compute-accuracy trade-off, we cannot change anything about the runtime speed. 11

1.7 Visualization of the speed-accuracy trade-off, we cannot change anything about the compute power. 13

1.8 Example of a speed vs. accuracy trade-off to evaluate different computer algorithms. (Image from `lin2017focal`) 15

2.1 Illustration of the amount of silicon area dedicated to certain task on GPU and Central Processing Unit (CPU). Orange is memory, yellow is control hardware, and green is actual processing cores. (Source: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`) 20

2.2 Historic overview of GPU performance compared to GPU performance. (Source: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`) 21

2.3 The GPU acts as a co-processor to the host-processor (CPU). . 22

2.4 Typical excecution flow of a GPU program. 22

2.5 Address space in OpenCL. 24

2.6 Overview of a V100 architecture. (Image from [103]) 26

2.7 Playing with the power consumption of a system changes the trade-offs that are influenced by the hardware constraints. More power can go into either more speed, or better accuracy. Depending on the hardware platform a better trade-off can be made. 31

2.8 Comparison between power and computing power of the embedded devices we used. 32

3.1 In the application of making a live video montage we are constraint in runtime-speed. 35

3.2 Kleene truth tables. 37

3.3 Effect of different optimisations executed on the GTX295 GPU. 48

3.4 Convergence of the algorithm. 50

3.5 Results for the BLOODTYPE benchmark. 51

3.6 Results for the GROWINGBODY benchmark. 52

3.7 Results for the GROWINGHEAD benchmark. 53

4.1 The endoscope lens rotates freely on the camera attachment. . 59

4.2 The prototype in action. 62

4.3 Low-latency video streaming is crucial in this application, leaving a trade-off between computing power and accuaracy.. 62

4.4 The tracking approach. The dots represent tracked corners in consecutive frames. 63

4.5 Overview of the algorithm. 64

4.6 Comparison of the Harris en Shi-Tomasi corner detecton. The predicted angle is plotted relative to the elapsed time in the test video (footage from real endoscopic surgery). The output of both corner detectors is almost identical. 65

4.7 System architecture. 68

4.8 System architecture. 70

4.9 Different display modes in the prototype. 72

4.10 Angles at different update rates. 73

4.11 Runtime comparison. 73

5.1 Visualization of different the applications, and where they fit on the triangle. 80

5.2 Challenges when dealing with solar panel detection. 80

5.3 The off-line detection of solar panels is accuracy constrained, leaving a trade-off between runtime speed and computing power. 82

5.4 Example of pixel based colour classification using neural networks as learning tool. (top) original image (middle) pixel classification result (bottom) cleaned up segmentation. 84

5.5 MSER based colour segmentation (a) Maximally Stable Extremal Regions (MSER) detected blobs (b) area restriction (c) ratio restriction (d) colour restriction. 86

5.6 Detection of solar panels using a boosted cascade of weak classifiers. 87

5.7 Detection of solar panels using the aggregate channel features technique. 89

5.8 Precision - Recall curves for all techniques tested and validated on the 16000x16000 test image of the Sint-Truiden city centre. 90

5.9 Score processing map for both Viola & Jones and aggregate channel features technique. (top) original input image (middle) boosted cascade result (bottom) aggregate channel features result 92

5.10 While speed has some importance in a real-time system, this application is only soft real-time. The quality of the shots, which is influenced by accuracy is much more important. We are left with a trade-off between runtime speed and computing power. . 95

5.11 Hardware scheme 96

5.12 Top view with cameras, buckets and zones adding to the buckets. 98

5.13 Camera selection results of three buckets during a 10 minute scenario. 101

5.14 Comparison of the time complexity of different detection algorithms on our test footage. 102

5.15 Detection performance on overview footage from a typical home. 103

5.16 Proposed canvases (red) based on people detections (blue) controlling the pan-tilt-zoom (PTZ). Video illustration on http://goo.gl/WIS0P4. 104

5.17 Instead of evaluating each window sequentially each window is assigned to a separate Compute Unified Device Architecture (CUDA) thread. 106

5.18 Comparison of the normal Aggregate Channel Features (ACF) to only the coarse detector and the coarse fine detector. Detectors with the 2048 suffix have a coarse detector trained up to a maximum of 2048 weak classifiers, the others only use 128 weak classifiers for the weak detector. “ACFBaseline” is our baseline ACF CPU implementation. 108

5.19 The stage parallel approach: each thread evaluates a separate decision tree. 109

5.20 Comparison between tested detectors, “ACFOurs” represents our ACF implementations, “YOLOv2” is the standard YOLOv2 object detector. 112

5.21 Comparison of the tested detectors in both speed and accuracy. 113

5.22 Comparison of different object detection algorithms tested on Pascal VOC2007 data (trained on VOC2007 and VOC2012). Image courteously provided by AlexeyAB from his darknet GitHub repository (`https://github.com/AlexeyAB/darknet`). 115

5.23 Because the systems needs to work on a drone, where power consumption and speed are important, the hardware constraint is fixed. We explore the trade-off between speed and accuracy further in this work. 116

5.24 System architecture of a UAV-based decision support system for emergency response 117

5.25 Example decision support interface 119
5.26 Input resolution compared to accuracy 122
5.27 Influence of input resolution on detection frame rate 123
5.28 Influence of video dimensions on detection frame rate 124
5.29 Comparison of the ACF detector to YOLO 124
5.30 Comparison of different detectors in terms of speed 125
5.31 Mean Average Precision vs. frame rate for the discussed object detection algorithms running on an NVIDIA Jetson TX2 127
5.32 Visual comparison a selection of detector on a single frame of the UAV123 dataset. 128

6.1 Illustration of the mapping from image to embedding space. . . 132
6.2 Classic re-identification pipeline, a detector is first run on the input image after which the detections are cut out and sent to the re-identification network. 133
6.3 Comparison between re-identification and person search. A more detailed comparison can be found in the paper [162] of Xiao et al. 134
6.4 Proposed pipeline. 134
6.5 We create a neural network that can do both detection and re-identification at the same time. Left shows the Euclidean distance of a tracked person compared to all other output cells, right shows detection output and centre uses both detection and re-identification to create a tracker. 135
6.6 Visualization of different application, and where they lie on the triangle. 136
6.7 Simplified YOLOv2 network architecture 139
6.8 Overview of the “split end”-architecture: detection and re-identification are preformed separately. 140
6.9 Overview of the “mixed end”-architecture: detection and re-identification layers are merged. 140

6.10 Example of the output of the network. The distance between anchor point and embedding is visualized, brighter means a large distance, darker means a small distance. 142

6.11 Comparison of our YOLO-REID network to standard YOLOv2. 143

6.12 Comparison in speed and accuracy. 145

7.1 Decision process to choose an algorithm that meets constraints. 151

7.2 Visualization of the runtime-compute trade-off, we cannot change anything about the accuracy. 153

7.3 Visualization of the runtime-compute trade-off, we cannot change anything about the speed. 155

7.4 Visualization of the runtime hardware trade-off, we cannot change anything about the hardware economy. 157

List of Tables

2.1 Comparison of some key aspects of the hardware we used. . . . 30

5.1 Comparison of training (given the training sets described in section 5.5.3) and detection times (based on the upscaled 16000 × 16000 pixel image) combined with the complete system configurations used. We used the following hardware: DES1: Intel(R) Core(TM) CPU i7-4500U @ 2.4GHZ, DES2: Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz. 89

5.2 Comparison of different approaches for feature calculation. 111

5.3 Comparison of different approaches for model evaluation. 111

5.4 Comparison of total processing times, with the exception of "Baseline (CPU)" feature calculation is done on GPU. 111

5.5 Comparison of total processing times of the tested detectors on both CPU and GPU. 113

6.1 The re-id performance of our two architectures compared to colour histogram matching, which has a similar computational cost. 144

6.2 Computational cost of different networks. 145

6.3 Comparison of tracking performance against our own trained one class You Only Look Once (YOLO) without re-id and with re-id enabled. 147

Chapter 1

Introduction

Artificial Intelligence (AI) and its many subfields have seen a lot of progress in the last years: many new techniques are becoming available and they are mature enough to be used in the real-world.

However, often they are made to run on a researcher's desktop computer only, without paying much attention to how they can actually be used in the real world.

Parallel to the field of AI, embedded systems and Graphical Processing Units (GPUs) have also seen a lot of progress in the last few years. GPUs are, driven by deep learning and graphics applications, more powerful than ever, and even embedded platforms are becoming capable of resource intensive algorithms. Despite all this, real-world applications still have many constraints that are hard to meet. In this thesis we take look at these constraints, analyse how they differ and what impact they have on each other, and the best ways to work with them. We do this using a framework that outlines which constraints are important in which applications.

Constraints can be unchangeable or "hard", meaning they are dictated by the requirements of the application. Different constraints also have an impact on other constraints. Making one constraint more challenging can have the effect of having to loosen another constraint. The framework of how these constraints relate to each other and which constraints are hard or loose will be used as a guideline throughout this thesis. It will paint a picture of how the many applications in this thesis from a broad variety of subfields of AI, relate to each other. Indeed, although the objective of the applications in this thesis varies a lot, when using them in the real-world, many of the implementation trade-offs

and constraints are shared between them. In section 1.2 we discuss this guiding framework in detail.

1.1 What do we mean by Artificial Intelligence?

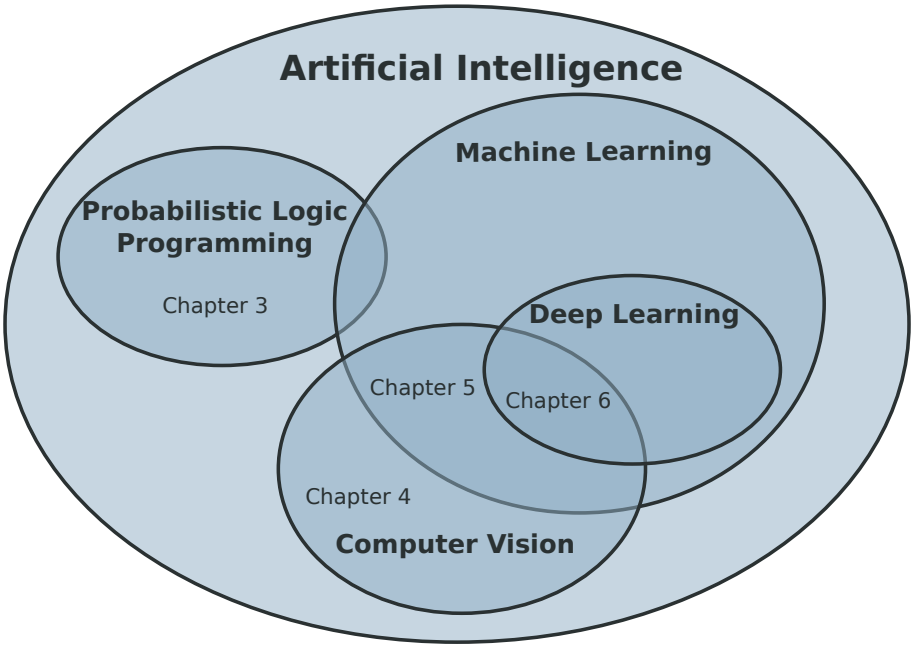
AI is a very broad term. In fact, what is considered as intelligence is not really well-defined. Defining AI means that we first have to define intelligence. Although an interesting topic, defining what intelligence means, and thus what AI is, lies not in the scope of this thesis. In this thesis we instead take a very broad definition of AI, ranging from (probabilistic) logic programming to image processing to computer vision. These topics are well studied and very relevant at the moment. Each of these different subfields have algorithms with their own challenges that are being used in applications with their own constraints. An overview of how these fields relate to each other is shown in figure 1.1.

In this thesis we take a case-based approach to studying these algorithms covering the following topics: In the first chapter titled: "A fast implementation of the Causal Probabilistic logic (CP-logic) Probabilistic Logic Programming language", we begin with a look at how CP-logic theories can be inferred using the GPU. CP-logic is a Probabilistic Logic Programming language, which can be used by domain experts to construct probabilistic-relational models. After that we look at a rotation compensation algorithm for use during endoscopic surgery, which uses feature tracking to determine orientation. From the field of computer vision we first take a look at object detection with the Aggregate Channel Features (ACF) detector, after which we look at person re-identification, person search and tracking using deep learning techniques. Of course these applications only cover a small subfield of what can be considered AI. We hope that with this thesis we cover enough of them to be representative of a larger part of what is AI.

1.2 What are these constraints?

This thesis is about constraints, both for hardware and software, and how they influence different applications of AI when brought into the real world.

During this PhD we look at a broad range of applications that all come with their own constraints. In this section we lay-out the framework of constraints that come back in all of these applications. We identified three groups of constraints that were present at some level in all of them.



3. An OpenCL implementation of a forward sampling algorithm for CP-logic
4. Ultra-low-latency automatic endoscopic image orientation stabilisation
5. GPU implementation of ACF Algorithm
6. Fast Simultaneous People Detection and Re-identification in a Single Shot Network

Figure 1.1: Venn diagram showing how the different fields of AI discussed in this thesis relate to each other, and where each chapter lies inside the diagram.

Concretely we identify the following constraints:

- Runtime speed requirements
- Hardware economy
- Required accuracy of the result

In some applications these constraints are hard, meaning there is no leeway in these constraints. We have to work with them rather than change them. For example, a drone on which we want to do processing might have a maximum take-off weight of few hundred grams, in which case we can't use a heavy computing platform. We are limited by a lightweight platform. Or if we have

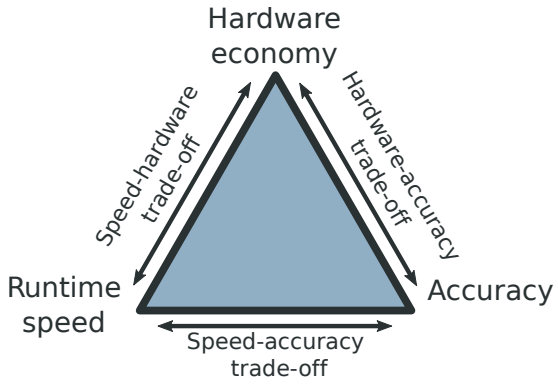


Figure 1.2: Triangle depicting our framework of three different constraints.

an application where latency is essential, the runtime speed constraint is hard. As these constraints are often imposed upon us as design requirements, we cannot change anything about them. The application needs to meet these hard constraints albeit at the cost of other constraints.

In most of the applications in this thesis one of these constraints is hard. This leaves a trade-off between the two other constraints. When two constraints are hard, there is no leeway any more in the remaining constraint. All we can do is try to circumvent the "hardness" by for instance trying to use the same hardware more efficiently.

In the scenario where three constraints are hard, there is no leeway, we can only hope that our application meets the constraints we set.

Figure 1.2 is a graphical representation of these constraints. Each corner depicts one of our three constraints: runtime speed, loss in accuracy and compute power. When one of these constraints is hard, we can make a trade-off between the two constraints opposite to the hard constraint. Take for instance the drone application with a maximum take-off weight (hardware constraint). We can't change anything about the hardware, but we can still make a trade-off between the two other constraints: runtime speed and loss in accuracy, called "speed accuracy trade-off" in the figure. Without changing anything in hardware we can either have an algorithm that runs faster but is less accurate, or an algorithm that is slower but more accurate. (Unless we change the hardware constraint by for instance finding an algorithm that utilizes the hardware more efficiently.)

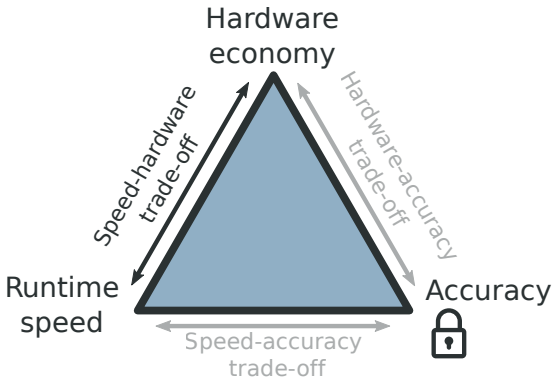


Figure 1.3: Visualization of the runtime-compute trade-off, we cannot change anything about the accuracy.

In total, we can identify three trade-offs, one for each hard constraint. We will discuss these further in the next sections.

One way of dealing with these trade-offs is to simply explore them. Without changing the algorithm we simply accept that if we want more of one thing, we have to pay with another. However, for some trade-offs there is another way. By *optimizing* the algorithm we might find a way to get the best of both worlds, we get better performance on one end, without giving in at the other. We might for instance find a way to make an algorithm faster without scarifying accuracy (e.g. using a more recent person detection algorithm), or we might find a way to utilize the available hardware more effectively (e.g. using machine specific instructions to utilise the hardware more efficiently).

1.2.1 Speed hardware trade-off

In this section we discuss what happens when we have an application where the accuracy constraint is hard. Our algorithm has to perform at a fixed accuracy, the quality of the results has to be the same, no matter the implementation. The runtime speed and hardware economy constraints are not fixed however. We can make a trade-off between these two. High runtime speed means the need for more compute power (and thus more hardware), and a low power platform means a lower runtime speed.

Some practical cases of this kind of trade-off are:

- Applications where critical decisions are taken based on the result of an

algorithm. E.g.: A pedestrian detection system for collision avoidance has to reach a certain accuracy to be considered safe.

- Applications where the quality of the result is very important, but calculations can be done offline and/or hardware cost is not the main concern. E.g.: Person search in a dataset.
- When we have an existing algorithm that works well, and we are happy with the results, but want to change the platform and runtime speed of where and how the algorithm runs. E.g.: Moving a pedestrian detector from a research workstation to an embedded platform.

In all of these applications we have the constraint that the accuracy remains the same. We get to make a trade-off between speed and hardware economy. How we make this trade-off is still open. In some cases we can get away with a faster runtime speed at a lower hardware cost[1]. Some platforms that we used in this thesis can offer us a better trade-off: parallel systems like (embedded) GPUs, servers and workstations, and embedded platforms with dedicated hardware. We will now discuss each of these individually.

Parallel systems

Parallel computing, more specifically GPU computing is often seen as a way to get a high compute performance at a relatively low cost. As it becomes harder to increase clock speeds (and thus speed up sequential execution) adding more cores to a chip seems like a logical solution. This has resulted in parallel processors offering orders of magnitude more performance. However, having more (parallel) compute power does not always result in better performance (speed), as will become clear in this thesis. Some problems are inherently easier to parallelise than others. When we have data dependencies during the execution of an algorithm, part (or sometimes all) of the algorithm cannot run in parallel at all (Amdahl's law [9], see further). No matter how many processors we add, when a next step depends on the state of the current step we cannot start executing the next step without the result of the current step. This is called the *critical path* of our algorithm. The *speedup* we can get from a more parallel processor depends highly on this critical path.

Formally we van define the speedup of an algorithm as follows:

$$S_p = \frac{T_1}{T_p} \tag{1.1}$$

[1] Cost in a broad sense, and not limited to purchase price. This might for instance also be power consumption or weight.

Listing 1.1: Sequential algorithm to add numbers.

```
def add_numbers():
    numbers = [0, 1, 2, 3, 4, 5, 6, 7]
    sum = 0

    for number in numbers:
        sum = sum + number

    return sum
```

Here T_1 is the time it takes to run the algorithm sequentially, T_p is the time it takes to run the algorithm on p processors and S_p is the speedup we can get using p processors. If $\frac{T_1}{T_p} = p$, then the algorithm has a perfect linear speedup. The algorithm has no sequential parts and can use all available processors. The amount of parallelism S_∞ our algorithm exhibits is the amount of speedup we would get if we were to run the algorithm on a theoretical machine with an infinite amount of processors.

Take for instance listing 1.1, where we add all numbers in the array `numbers` together. Using the current algorithm our intermediate result "`sum`" in the for-loop always depends on the previous step. We have to run the algorithm sequentially to get the correct result. Our algorithm does not benefit from parallelism, it has a speedup of one. If we were to naïvely try to run the for-loop in parallel i.e. every thread adds a different number in the list to the variable "`sum`" at the same time, it is unclear what the final result in "`sum`" would be. It would very likely be a wrong value. Multiple threads might try to read or write the `sum` variable at the same time. Say for instance that two threads, a and b, both read the value of sum, modify it and then write the result back. Chances are the value that was in `sum` when thread a was reading was changed in thread b before thread a read it. In that case the result of thread a is based on an outdated value of `sum` and the result of the addition in thread b gets lost, resulting in an incorrect result overall. This problem is called a *race condition*. The difficulty in modifying an algorithm to run in parallel is finding a way to turn a sequential algorithm into a parallel one, while still getting the same results.

Listing 1.2 is a parallel version of our "`add_numbers()`" example. Figure 1.4 gives a visual overview of the execution flow. Instead of adding all numbers sequentially, we add as many numbers as we can in parallel. In each iteration of the for-loop the algorithm adds pairs of numbers together in parallel, until we get the final result. We still have a for-loop, but instead of 8 iterations we

Listing 1.2: Parallel version of the add_numbers example. The `parfor` instruction launches 4 threads each with their own id called `thread_id`.

```
def add_numbers():
    numbers = [0, 1, 2, 3, 4, 5, 6, 7]
    sum = 0

    parfor thread_id in thread(4):
        for i in range(log2(8)):
            result_idx = thread_id * 2^(i + 1)

            if results_idx >= 8:
                break

            left_value_idx = result_idx + 2^i
            numbers[result_idx] = numbers[result_idx] +
                numbers[left_value_idx]

    return numbers[0]
```

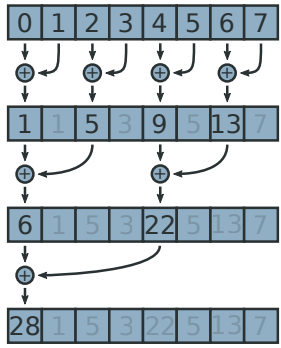


Figure 1.4: Execution flow of the parallel sum algorithm.

now only have to do $\log_2(8) = 3$ iterations. On a processor that is sufficiently parallel we get at speedup of $\frac{8}{3} = 2.6$.

We can use Amdahl's law [9] to determine the theoretical speedup we can get given a speedup of the parallel part of the algorithm s_p, and r_p the ratio of our

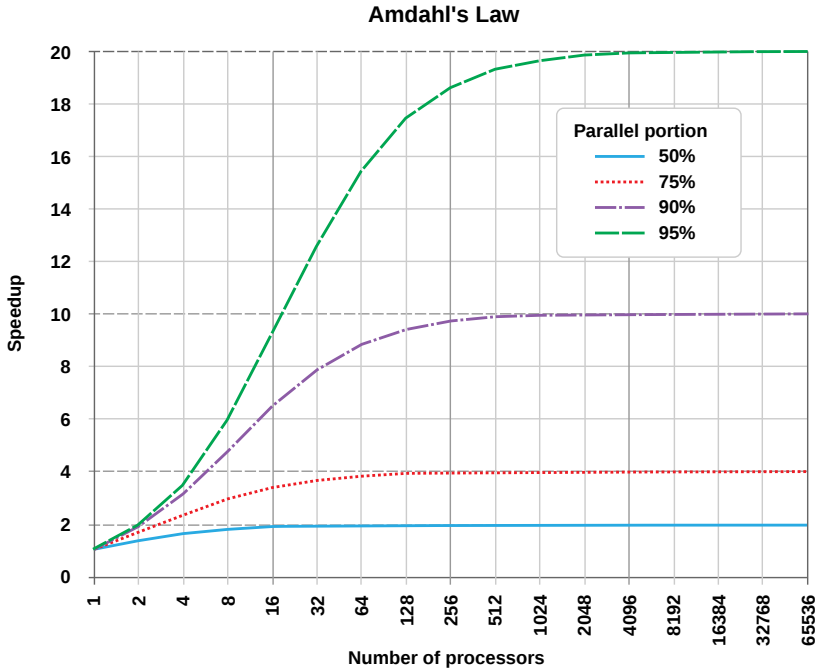


Figure 1.5: Comparison of different speedups in function of the number of processors of algorithms exhibiting a different parallel portion. Credit: Daniels220 at English Wikipedia [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

algorithm that can run in parallel. The formula is given as follows:

$$S = \frac{1}{(1 - r_p) + (r_p / s_p)} \tag{1.2}$$

Figure 1.5 shows speedup in function of number of processors using Amdahl's law when we apply it to algorithms with a different parallel section. After some point adding more processors does not give us a bigger speedup, as long as we have a critical section.

As a toy example let's extend the sum example a bit more. Say that we want to write a parallel program to calculate the sum of first n Fibonacci numbers.

Listing 1.3: Program to calculate the sum of the first n Fibonacci numbers.

```
def calc_fibonacci_sum(n):
    numbers = calc_fibonacci_numbers(n)
    sum = calc_sum(numbers)
```

Listing 1.4: Program to calculate the first n fibonacci numbers.

```
def calc_fibonacci_numbers(n):
    numbers = [0, 1]

    for i in range(2, n):
        numbers.append(numbers[i - 1] + numbers[i - 2])

    return numbers
```

The general algorithm we use is shown in listing 1.3[2]. The algorithm we use to calculate the Fibonacci numbers is shown in listing 1.4. Because every step of the for-loop depends on the previous step this algorithm is hard to parallelise. We say that the `calc_fibonacci_numbers()` functions is the critical section of the program, it cannot be parallelized. For the parallel section, when $n = 8$ we got a speedup of $s_{p,8} = 2.6$. When we run the algorithm with even more input data the speedup increases. When $n = 1024$ our parallel sum example needs $\log_2(1024) = 10$ iterations, resulting in a speedup of $s_{p,1024} = \frac{1024}{10} = 102.4$. The number of required operations in both steps of the program is the same. Calculating the Fibonacci sequence and summing the numbers together requires equal cost, so $r_p = 0.5$. Plugging these numbers into equation 1.2 we get a speedup $S_8 = 1.45$ for $n = 8$, and a speedup $S_{1024} = 1.98$ for $n = 1024$. Notice that while $s_{p,1024}$ is way higher than $s_{p,8}$, because we have a non-parallel section in the algorithm the total increase in speedup (S_{1024} and S_8) is still not that great.

In fact, in the theoretical case S_∞ ($p = \infty$), where the parallel section of the algorithm goes to a speedup of infinity, the total speedup can be found by:

$$\lim_{p \to \infty} S_\infty(s_p) = \frac{1}{1 - r_p} = \frac{1}{1 - 0.5} = 2 \tag{1.3}$$

Our program can only reach a maximum speedup of 2.

[2]Note that this is not the most efficient way to calculate the sum of a Fibonacci sequence. This example only serves as an example to demonstrate Amdahl's law.

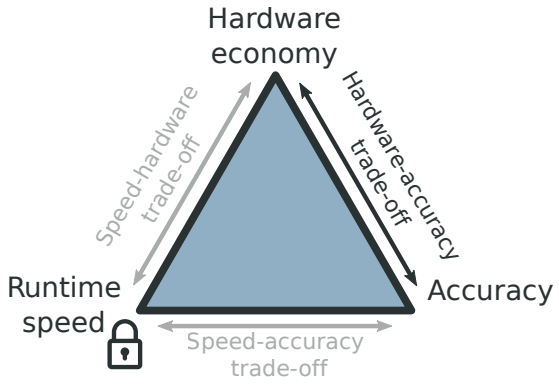


Figure 1.6: Visualization of the compute-accuracy trade-off, we cannot change anything about the runtime speed.

Using this knowledge we can determine whether it is beneficial to add more parallel hardware in favour of a better runtime speed. When the algorithm by itself is not parallel enough, adding more processors is not beneficial. It might however be possible to change the algorithm to reduce the critical section and increase the amount of parallelism, in which case adding more processors can be a good idea.

Dedicated hardware

Purpose-built hardware might also be a good idea when finding a good trade-off between running speed and hardware cost. During this PhD we only touched this subject briefly with the video processing unit of the Freescale iMX 6 in chapter 4. However, when hardware is available on chip to do your specific task or it is worth the effort to do it on Field Programmable Gate Array (FPGA), it might certainly be an interesting avenue that we didn't get into that much during this thesis. Platforms specifically designed for deep learning also exists, examples of these are: Google Edge TPU, Intel's Movidius and recent NVIDIA architectures with deep learning specific hardware.

1.2.2 Hardware accuracy trade-off

Some applications are constrained in speed. External factors dictate that they need to return results in a certain amount of time (latency) to meet specifications. In these applications the runtime speed requirement is considered hard, and we

can explore the trade-off between hardware economy and accuracy (Figure 1.6). Examples of such applications include:

- Low latency video streaming. E.g.: Doing image processing operations on a low latency video stream for use during surgery. The surgeon has to maintain hand-eye coordination which is hard when there is no *direct* visual feedback. In the application in chapter 5 we strife for a latency lower then a single frame interval at a framerate of 60 fps.
- Decisions based on the result of the algorithm have to be made instantly. E.g.: Making last minute adjustment to avoid an obstacle when automatically controlling a drone.

In the world of real-time embedded applications, certain distinctions are made in categories of real-time systems:

Hard real-time For mission critical systems, if the task is not carried out by a certain deadline the complete task is ruined, which can even have safety critical consequences. E.g.: ABS systems in a car, controlling the position of a 3D printer.

Firm real-time The system will not operate like it should if deadlines are not met, but the system can continue operating. As long as not too many deadlines are missed it is only an inconvenience for the user. E.g.: Decoding a video to display in real-time or Artificial Realtiy glasses.

Soft real-time Things are fine as long as the system returns results within a reasonable amount of time. E.g.: Taking a picture from a remote security camera.

In the first two cases, hard and firm real-time systems, we consider the runtime speed constraint to be hard.

When speed is hard, and the algorithm's accuracy is limited by compute power, an obvious way to solve the problem is to introduce more compute power. But is this always the best option? The alternative is: When we notice that our speed constrained application runs too slow we might speed it up by changing the algorithm or its parameters in some way. We might notice that the algorithm runs faster when we use a less accurate calculation in a part of our algorithm. This works well with algorithms that make a prediction or an approximation. Concrete examples of this include: Reducing the number of samples in a Monte Carlo simulation, using fewer weak classifiers in the adaboost algorithm [46], using a neural network that is less deep, using half precision floating point

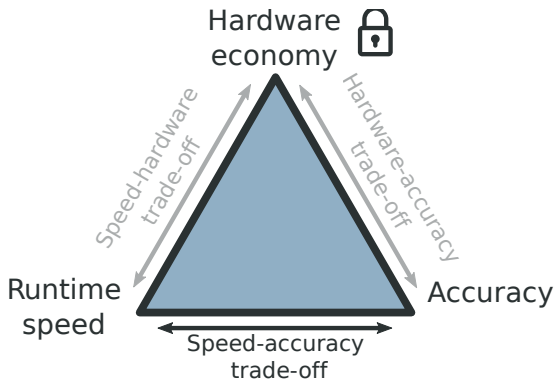


Figure 1.7: Visualization of the speed-accuracy trade-off, we cannot change anything about the compute power.

numbers instead of normal 32 bit floats... These measures ensure that our algorithm runs faster with the same hardware. By introducing more hardware we can undo these constraints and make the algorithm more accurate again.

Another option is to make the algorithm faster by introducing some other heuristics in the algorithm. When optimising a person detector we might for instance say that features can only be read from a certain position in a feature map such that the memory access pattern is optimal to be ported to GPU. Or turn a sequential for-loop into a parallel one that handles multiple steps at the same time, but with fewer intermediate results.

But if these measures were never introduced, is it still possible to improve accuracy by improving hardware? Say we want to use an algorithm from a state-of-the-art research project, but we would like it to become more accurate. Introducing more hardware might not be beneficial here. The algorithm is already tuned to output the best possible performance, no matter the computing cost. Making the algorithm more complex by making it do more precise calculations or making the neural network deeper might not help much. The algorithm is not able to represent a more complex description of the problem. To take advantage of the new hardware, the algorithm often has to be changed. In the case of a neural network it might actually start to overfit after a while and drop in accuracy. All this means that it is not possible to made this trade-off in all applications.

1.2.3 Speed accuracy trade-off

In many applications the hardware platform is already predetermined. A company might want to add a feature to an existing product without changing the hardware. Or the hardware is limited by physical factors. For instance the hardware platform on a drone needs to be low power and lightweight.

In this case we can make a trade-off between runtime speed and algorithm accuracy (figure **??**). Examples of this include:

- Running a pedestrian detector on different resolutions. On a low resolution it will run fast, but will be less accurate, while on a high resolution it will be more accurate but slower.
- Limiting the number of iterations in an iterative calculation. E.g.: Calculating the square root via the Newton-Raphson method. More iterations mean more accuracy at the cost of a slower speed.

Without changing anything to the hardware, we might again be interested in reaching an optimal trade-off, whatever that means. Making this trade-off of course depends on the application. Applications where we are more on the speed-sensitive side of the spectrum include:

- Controlling a system in real time. E.g.: Following a pedestrian from a drone [64].
- Split-second decisions. E.g.: Using a reasoning framework to determine what kind of shot a filming drone has to make.

On the accuracy end of the trade-off we find:

- Applications where waiting a bit longer is not a big issue. E.g.: Automatic passport border control [77].
- Applications that can be done offline. E.g.: Automatic image retouching on a digital camera.

Computer vision algorithms are often evaluated using an accuracy vs. runtime speed trade-off. Figure 1.8 shows an example of such a trade-off. In this case the accuracy of the algorithm is the Mean Average Precision (mAP) (a measure of how well an object detector performs) of the YOLO [114] and ACF [33] detection algorithms. On the x-axis is the framerate. In this case the best performing algorithm, the algorithm that uses the hardware the best, is the

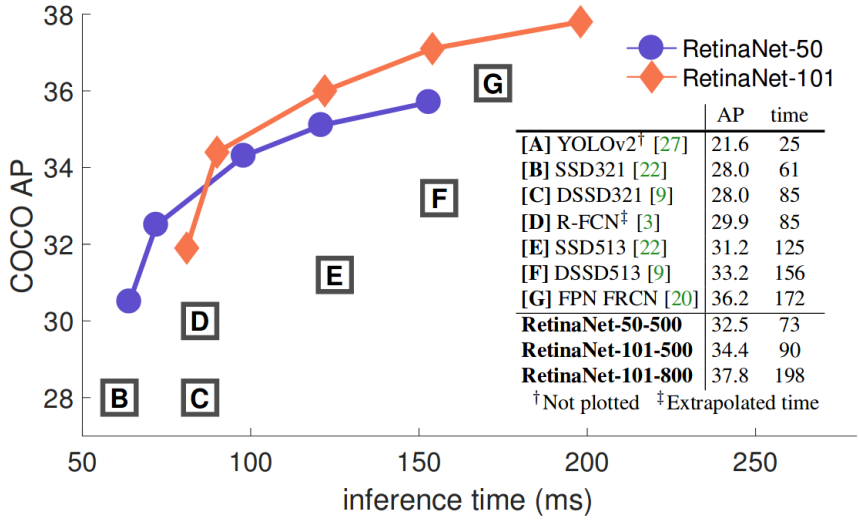


Figure 1.8: Example of a speed vs. accuracy trade-off to evaluate different computer algorithms. (Image from `lin2017focal`)

algorithm in the top right corner. Indeed, we can measure the quality of an algorithm by looking at this speed accuracy trade-off.

In some cases speed and accuracy can be even more intertwined. Take for instance a tracker. When the tracking algorithm is too slow, tracks can be interpolated on skipped frames. A popular way then to measure tracker performance is to let the tracker run in real-time, and skip frames if it is too slow. This is what would happen if we use the tracker in the real-world. A faster tracker then has the advantage that it gets to see more frames, so it has to interpolate less, and wins in accuracy.

1.3 Contributions

Apart from using the different cases as examples of describing the framework that is introduced in this thesis, the cases on their own are of course also big contributions. In this section we highlight these contributions

In chapter 3 we present a forward sampling algorithm for CP-Logic theories using the GPU in OpenCL. Our contributions to the state-of-the-art here are

twofold: We're the first to try to use the GPU for inference of Probabilistic Logic Learning (PLL) theories, which enables us to reach speed-ups of 37× compared to the baseline Central Processing Unit (CPU) implementation, and we're also competitive, and in many cases even better, then the state-of-the-art. Secondly, as it is a forward method, which means that it is not query-based, it can compute the probabilities of a set of queries almost as quickly as the queries themselves.

In chapter 4 we develop an endoscopic counter rotation method, and examined how to integrate it into a digital operating room. We did all this at ultra-low-latency, which is important because research suggest that latency is a key parameter for a surgeon's performance. We look at three different applications on the various different hardware platforms that where available. We find that the one that used a separate machine performed best, and use that due to the safety critical nature of this application, and because hardware is not a constraint.

In chapter 5 we created a GPU implementation of the ACF person detector. We evaluate on two platforms, an embedded GPU platform, and a desktop machine containing a GPU. While the ACF detector does not lend itself easily to big speedups by parallelizing (especially the evaluation step) we still manage to get some significant speed increases on both tested platforms. Compared to the newer deep learning detector YOLOv2 we still manage to get the fastest detections using the GPU ACF method on the embedded platform. Apart from that we also look at some applications:

Application 1: Detection of photovoltaic cells In this application we compare different detection methods to find out the best way to detect solar panels.

Application 2: Automatic camera crew We create a system to automatically follow people in their home using pan-tilt-zoom (PTZ) cameras. We create a system that detects people, and then steers a PTZ camera towards them making a cinematographically pleasing shot. We also create a system to automatically turn on the cameras.

Application 3: Object detection on board of a Unmanned Areal Vehicle (UAV) We evaluate how well object detection (including our technique) can be done on board of a UAV.

In chapter 6 we create a novel solution to the classic person search pipeline. Currently person search is mainly done by fist running a detection network and then separately running a re-identification network. In this work we integrate this into one pipeline built on top of the You Only Look Once (YOLO) detector.

We manage to do re-identification at a negligible cost in addition to the detection that is already done. We mange to create a very fast pipeline with very usable and sufficiently accurate re-identification, which is useful in many applications such as tracking.

1.3.1 Look forward

Throughout this thesis we will use our constraints framework as a guideline to explore and optimise different trade-offs in a number of different applications.

The rest of this thesis consists of the following: Chapter 2 goes a bit deeper into the hardware we used during this thesis, and how different hardware platforms compare to each other.

Chapter 3 explains our work on parallel inference for CP-Logic theories using OpenCL. We explore how a probabilistic logic programming inference system can be implemented in a parallel way using Monte Carlo like simulations on the GPU in OpenCL.

Chapter 4 is about our work on ultra-low-latency automatic endoscopic image orientation stabilisation. Minimally invasive surgery often makes use of endoscopes with an angled lens. When using such an endoscope, the on-screen image tends to rotate wildly, which may confuse the surgeon. A low-impact way of solving this problem is by using computer vision techniques to track the rotation of the incoming footage and then counterrotating the image.

Chapter 5 goes into our GPU implementation of the ACF object detector, and its many applications. While current state-of-the-art results in pedestrian detection are hands down reached by deep learning methods, in terms of running speed traditional hand-crafted feature methods can still be faster. This is especially true on embedded platforms that often require real-time performance, while at the same time having to make do with a limited amount of resources. We compare our implementation to current deep learning methods to explore how relevant hand-crafted feature methods still are.

Chapter 6 explains our work on fast simultaneous people detection and re-identification in a single shot network. A traditional re-identification pipeline consists of a detection and re-identification step, i.e. a person detector is run on an input image to get a cutout which is then sent to a separate re-identification system. In this work we combine detection and re-identification into one single pass neural network. We propose an architecture that can do re-identification simultaneously with detection and classification. The effect of our modification

has only a negligible impact on detection accuracy, and adds the calculation of re-identification vectors at virtually no cost.

We end with a conclusion and a look back in chapter 7.

Chapter 2

Hardware and frameworks

This chapter goes a bit further into depth about the "hardware corner" of our triangle-shaped framework (figure 1.2). While in the rest of this thesis the two other corners, accuracy and speed, vary based on the applications and algorithm, the hardware platforms we use (GPUs and embedded systems) are typically chosen beforehand and remain largely the same. Knowing the hardware really well is also essential when we optimize algorithms at a low level, to make them run more efficiently. For these reasons we dedicate an entire chapter to hardware.

During this thesis we used many target hardware platforms. What they had in common was that they were all constrained in some way. Small embedded platforms for instance often only contain a low power ARM processor with a very limited amount of processing power. This makes developing for them challenging if we want to reach a certain accuracy limited by real-world constraints like latency or frame rate. GPU platforms on the other hand offer lots of resources but use a programming paradigm that makes it harder to write software that uses these resources optimally. In this chapter we give an overview of the different types of hardware we used during this thesis. We take a look at different limitations and programming models that will come back in this thesis.

In this chapter we first take a look at GPU programming and GPU architectures (section 2.1). After that we take a look at the Embedded platforms we used in section 2.2.



Figure 2.1: Illustration of the amount of silicon area dedicated to certain task on GPU and CPU. Orange is memory, yellow is control hardware, and green is actual processing cores. (Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

2.1 GPU Computing

In the world of microprocessor development these days it is clear that the best way to get more performance from processors is to add more computational units instead of increasing clock speed. Increasing clock speed seems to have hit a wall, but transistor size in silicon has continued to shrink according to Moore's law [100]. (At least for now, in the near future it might be harder to keep up [158].) This leads to systems that are more and more parallel. Indeed, the processors with the best performance in terms of floating-point operations are those that can exploit this parallelism in the best way. GPUs have historically always been good at this. Originally GPUs came from the world of computer graphics, where they were purpose-built to quickly render graphics scenes, mainly for video games, which can easily be done in parallel. Starting in the late 2000s with the arrival of programmable shaders for computer graphics the interest in using these processors for other tasks not related to graphics grew. This gave rise to the field know as General Purpose computation on the GPU (GPGPU) [90]. The first adaptions tried to fit their problem in the existing graphics pipeline, implementing their programs in shading languages like the OpenGL Shading Language (GLSL). It didn't take long before languages with a more general purpose target came to market. NVIDIA launched the Compute Unified Device Architecture (CUDA) language specifically for their hardware, and the Khronos Group (the organisation behind OpenGL) came out with OpenCL targeting multiple platforms. This made programming for the graphics processor a lot more accessible.

The general programming paradigm did however remain the same. To program a

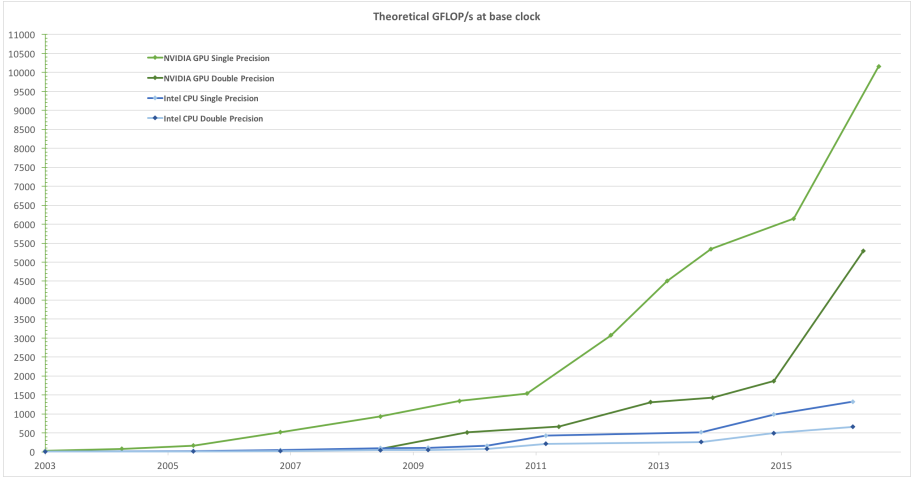


Figure 2.2: Historic overview of GPU performance compared to GPU performance. (Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

processor that does parallel computations one also needs to program it parallelly. GPUs contain huge arrays of parallel processors (around 15 000 single precision Floating Point Units (FPUs) in the latest NVIDIA V100 architecture). However, these cores do not offer the same flexibility as regular CPU cores. They are grouped together in "multiprocessors" that control multiple processing cores at once. They all share (among others) memory, registers, schedulers and program counters, making them less flexible than CPU cores. Compared to CPU cores, GPUs devote way more silicon area to actual processing cores and less so to control flow or registers, which is where GPUs get their tremendous performance (illustrated in figure 2.1). Historically this has always resulted in GPUs leaping ahead in raw performance compared to CPUs (shown in figure 2.2).

2.1.1 Using a GPU for general purpose computation

On a traditional desktop computer the GPU acts as a co-processor to the host-processor (CPU). GPU and CPU are physically connected to each other by a common interface (most often PCI-Express). All communication between CPU and GPU happens over this interface managed by the CPU (figure 2.3). On a traditional setup both CPU and GPU also have their own memory. The CPU uses the host's system's memory, and the GPU has its own dedicated

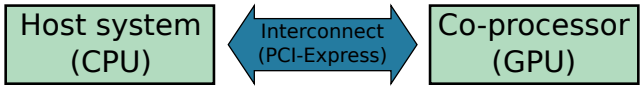


Figure 2.3: The GPU acts as a co-processor to the host-processor (CPU).

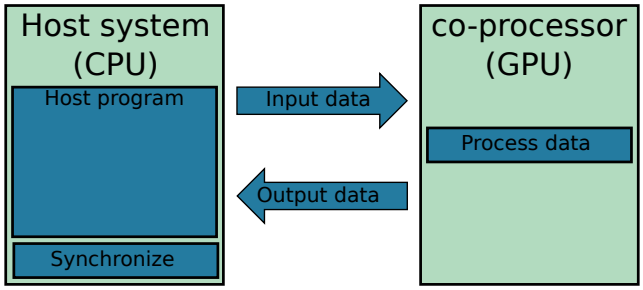


Figure 2.4: Typical excecution flow of a GPU program.

graphics memory. Both of them are typically some type of DRAM. This means that, when the host system hands over some computation to the GPU, the needed input data and output data needs to be copied back and forth. On some systems however, for instance NVIDIA's Jetson TX2 [2], host memory and graphics memory are shared and no memory transfers need to happen.

These implications bring with them a certain high level *execution flow* that is followed when using the GPU as co-processor for computation (shown in figure 2.4). At some point in the host program the algorithm has some input data that is ready for parallel processing on the GPU. This can be input data to the program or data that has already been processed by the host program. When the host program has this data, it gives the computing framework (which handles interaction between host and GPU) the command to copy the data over to GPU memory on some previously allocated space. After that, the host sends the next command which is to start a GPU program to process the data. Finally, the host sends the command to copy the output data back from GPU to CPU to get the final results. All commands sent from host to GPU are asynchronous meaning that the host does not have to wait on the GPU to finish computation. If we do want to wait, for instance if we want to continue with the results we got from GPU computation, we can send a synchronization command which will block until the GPU has finished all computation.

Programs running on GPU are called *kernels* in both OpenCL and CUDA. Kernels are short functions taking pointers to input and output buffers to do

computation on data. In some sense a kernel is similar to the body of a for-loop. A kernel is launched a predefined number of times as requested by the host program. Each instance of this kernel, called a "work-item" in OpenCL or "thread" in CUDA is then analogous to one iteration of a for-loop. We do however have to keep in mind that these instances can run in parallel and that multithreading artefacts like for example race conditions (explained in section 1.2.1) can be an issue. Listing 2.1 shows an example of an OpenCL kernel. We got two input vectors `vector_a` and `vector_b` which are added together and stored in `result`. `vector_size` is the length of the vectors and tells the program which threads do not need processing, in case it is launched for more threads[1]. `global_id` represents the current work-item of the current instance of the kernel, and is in our analogy similar to the iterator in a for-loop. Different work-items (or threads) exist in groups called work-groups (or blocks), work-items in the same work-group can co-operate more closely: they share faster memory (see later) and can synchronize without having to relaunch the kernel.

In most C code the compiler figures out whether a variable should be kept in a register or stored in DRAM. In GPU programming frameworks like CUDA or OpenCL this is not the case. The programmer has way more control over which memory space is used. Figure 2.6 gives an overview of memory spaces in OpenCL. CUDA memory spaces are similar but use different names. Starting from the bottom of the figure there is the host memory. This is the normal DRAM used by the host program. The GPU has no access to this memory. Data first has to be copied to the global or constant memory if we want to use it on the GPU. Global and constant memory also reside in DRAM but this time on the GPU side. Global memory is read-write, and constant memory is read-only. Moving up even further we have local memory. Local memory ("shared memory" in CUDA) is memory that is shared between work-items in the same work-group. It is generally stored in caches or registers on chip and is therefore really fast. It is most often used when different work-items need to work together, for instance when reducing an array of data to its sum. Private memory ("local memory" in CUDA) is memory that is local to a work-item. It is also stored in registers and is used for local variables inside the kernel.

2.1.2 GPU Architecture

To explain the inner workings of a modern graphics processor, in this section we will explain as an example the architecture of NVIDIA's Volta V100 GPU, at the time of writing NVIDIA's most recent GPU. Figure 2.6 gives an overview

[1] Sometimes more threads than necessary are launched to get a number of threads that is a power of two.

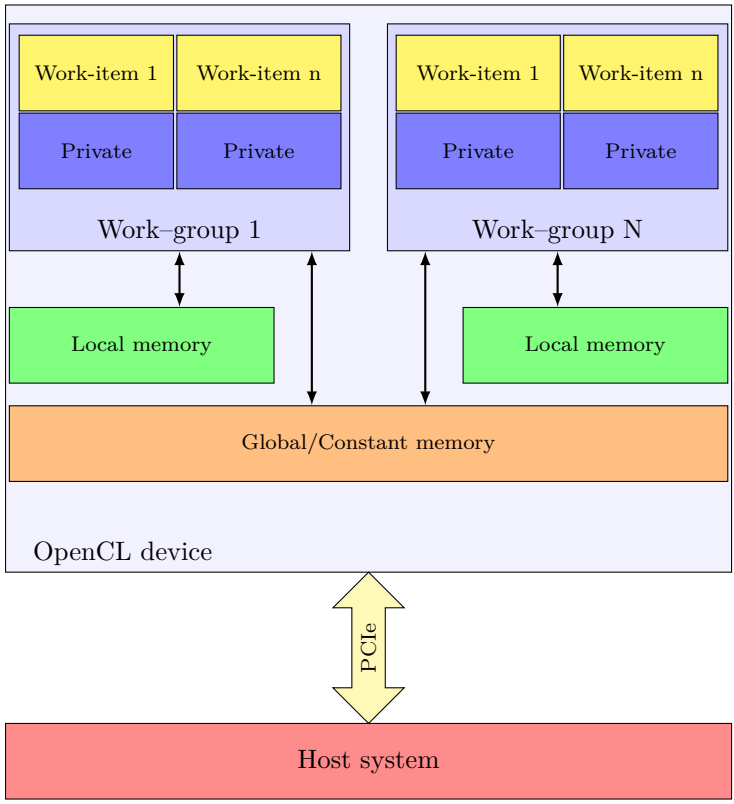


Figure 2.5: Address space in OpenCL.

of the architecture of the V100. The GPU is divided into six GPU Processing Clusters (GPCs) which can be thought of as self-contained processors containing all features of a GPU. The GPCs themselves are divided into seven Texture Processing Clusters (TPCs), which are groups of Streaming Multiprocessors (SMs) that share graphics related paraphernalia like texture caches and geometry processors. A SM finally contains the actual processing cores, combined with control hardware and L0 caches (for private memory) and L1 caches (for local/shared memory). A single SM is able to execute four warps (groups of threads that are executed together) consisting of 32 different threads at the same time. One SM contains 64 32 bit FPUs, 64 32 bit Arithmetic Logic Units (ALUs), 32 64 bit FPUs and 8 tensor cores (hardware specifically optimized to do tensor arithmetic for Deep Learning applications), which can all be used at the same time. Before the Volta architecture all threads in a warp had

Listing 2.1: Example of an OpenCL kernel.

```
__kernel void vector_add (
    __constant float *vector_a,
    __constant float *vector_b,
    __global float *result,
    const int vector_size)
    {
        const int global_id = get_global_id(0);

        if (global_id < vector_size)
        {
            result[global_id] = vector_a[global_id] +
                                vector_b[global_id];
        }
    }
```

to execute in locked step, meaning that when there is branch divergence, for instance when executing an if statement where some threads take one branch and others the other, all threads need to execute both branches. Since Volta however the SMs are equipped with more sophisticated control hardware where each thread can have its own program state. More information can be found in NVIDIA's whitepaper [103].

Powerful GPU's like NVIDIA's V100 are definitely most suited for the most complex algorithms which require a huge amount of processing power. Cases in this thesis that require such a tremendous computing power are the "YOLO-ReID" detector in chapter 6 during training, and inference with deep backend, and our GPU implementation of ACF in chapter 5 when processing a huge amount of video streams.

2.1.3 Deep Learning frameworks

With the rise of deep learning many high-level frameworks containing some commonly used GPU operations have become popular. These frameworks contain a high-level interface to underlying GPU code in a high-level language like python or Lua. They offer an easily accessible toolbox to implement deep learning networks complete with surrounding algorithms like gradient descent. Some of these frameworks include:

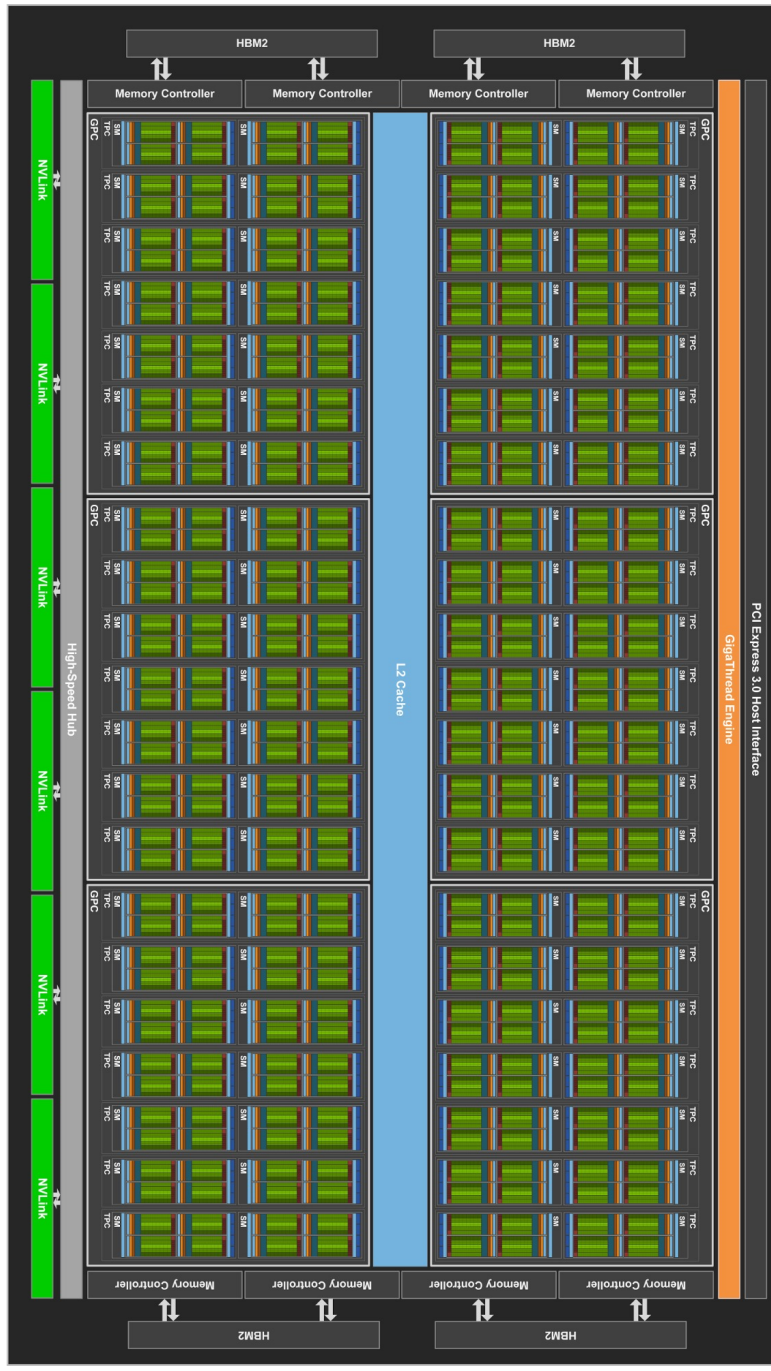


Figure 2.6: Overview of a V100 architecture. (Image from [103])

TensorFlow TensorFlow[2] is the deep learning framework developed by Google. Because of this, most of their research uses it, making it a very popular framework. Apart from GPU processing, it also supports Google's specialized Tensor Processing Unit (TPU) processing nodes in Google's data centres. It uses a graph based design where users first have to construct a graph representing the network architecture, which is then used for neural network training or inference (or other tasks).

CNTK Microsoft Cognitive Toolkit[3] is Microsoft's deep learning framework. It's written in C++, offers a Python API, and uses like Tensorflow a graph to define the network architecture.

Theano Theano[4] is a deep learning framework developed at the University of Montreal. It uses a Python API and is written in Python. Like PyTorch, it uses a NumPy like syntax to define operations on tensors.

Keras Keras[5] is a high level neural network that works on top of other frameworks (TensorFlow, CNTK or Theano). It offers a very user-friendly Python API.

Caffe (2) Caffe[6] was initially developed at the University of California, Berkeley. It's an open source deep learning framework implemented in C++, and provides a Python interface. Neural networks are defined by a configuration file instead of using a programming language to define the architecture. In 2017 Facebook announced Caffe 2[7] , adding additional features such as recurrent neural networks to Caffe. Recently Caffe 2 and PyTorch merged into a single project mainly supported by Facebook.

PyTorch PyTorch[8] is a relatively new framework, mainly developed by Facebook AI Research. It uses an easy to follow programming model where the network is defined by a series of operations on tensors. The framework is then capable of calculating gradients on the tensor operations, making it capable of learning tasks. Instead of constructing a computation graph (like in Tensorflow[9]), one can construct a network imperatively. In recent years it has been rising in popularity tremendously, especially for research. With the arrival of its first stable release, and the merge with Caffe 2, it promises to become one of the major deep learning frameworks.

[2] https://www.tensorflow.org/
[3] https://www.microsoft.com/en-us/cognitive-toolkit/
[4] http://deeplearning.net/software/theano/
[5] https://keras.io/
[6] http://caffe.berkeleyvision.org/
[7] https://caffe2.ai/
[8] https://pytorch.org/
[9] Tensorflow has recently introduced an "eager mode" which supports imperative network definition.

TensorRT TensorRT[10] is NVIDIA's framework for high performance neural network inference. Unlike the other frameworks here it is not capable of training. It contains a network optimizer to do inference of neural networks as fast as possible on NVIDIA hardware.

2.1.4 Conclusion

The GPU can offer a tremendous amount of computing power at a relatively low cost compared to normal processors. We do however need to use a programming paradigm that is somewhat harder to work with. Frequent programming patterns are often however already implemented making it easier to use the GPU without much expertise in GPU programming. Whether or not a GPU might be useful in an applications highly depends on the amount of parallelism that can be found in the application. The remaining chapters of this thesis will all use GPU computing in some form.

2.2 Embedded platforms

The type of hardware that is most suitable to solve a specific problem largely depends on the problem at hand. Embedded systems are special-purpose devices encapsulated in a device to do a specific task. Generally they have fewer resources then a traditional desktop setup which makes them a challenging target to deploy demanding algorithms. Popular examples of these devices which we used during this PhD include the iMX6 and the NVIDIA Jetson TX2. In this section we will briefly explain the architecture of both of these devices, and point out the hardware constraints of these platforms.

2.2.1 Freescale iMX6

The Freescale (now NXP) iMX6 was together with NVIDIA's Jetson TK1 one of the first embedded processors that offered low power ARM cores coupled with a GPU that could be used for general purpose computation. The iMX6's Vivante GC2000 GPU supports a limited implementation of the OpenCL standard called OpenCL EP (Embedded Profile) [102]. These limitations include:

- A global work size (number of threads that can be launched) of 65536 threads.

[10] https://developer.nvidia.com/tensorrt

- A work-group size of (depending on the number of instructions in the kernel) 16 – 256.
- The kernel is limited to 512 instructions.

These limitations make programming for this GPU quite challenging.

The Vivante GC2000 GPU contains four shader cores each capable of doing four (vectorized float) instructions at the same time, giving it a maximum performance of 16 Number of Giga floating-point operations that can be executed in one second. (GFLOPS) using multiply add (MAD) instructions. The development board has a power consumption of 5W [3].

2.2.2 NVIDIA Jetson Embedded computing board

Since the Jetson TK1 released in 2014 based on the Kepler architecture, NVIDIA has been releasing embedded System on a Chip (SoC) versions of its GPUs. These platforms contain an ARM processor paired with a smaller version of its desktop GPUs. The latest in this series is the Jetson AGX Xavier which uses the latest Volta architecture described in section 2.1.2. Other embedded GPU processing boards include the Jetson TX1 and Jetson TX2 based on NVIDIA's Maxwel and Pascal architecture respectively. Table 2.1 gives an overview of the relevant specifications of these devices.

2.2.3 Power consumption vs. computing power

Figure 2.8 shows a comparison of the devices we developed for in terms of power consumption and GPU computing power. As we would expect there's an obvious correlation between the two: more computing power generally means more power consumption. Playing with the power consumption of a system changes the trade-offs that are influenced by the hardware constraints. More power can go into either more speed, or better accuracy. Depending on the hardware platform a better trade-off can be made (figure 2.7). When we have a more powerful platform (i.e. more GFLOPS) we can either make the application run faster, or make it more accurate.

A simple way to make the application run faster at the cost of more power consumption (and thus satisfies a harder compute constraint) is to increase the clock speed (up to some limit determined by the heat output and stability of the hardware)t co. Without any algorithmic changes the algorithm will

[11] NVIDIA quotes the number of tensor cores instead of CUDA cores.

	iMX6 (Vivante GC2000) [6]	TK1 [4]	TX1 [2]	TX2 [2]	AGX Xavier [5]	V100 [103]
Architecture	N/A	Kepler	Maxwell	Pascal	Volta	Volta
"CUDA" Cores	4 (shader cores)	192	256	256	512[11]	5120
Memory	4 GiB (shared)	2 GiB	4 GiB	8 GiB	16 GiB	32 GiB
Theoretical FP32 GFLOPS	32	326	511	665	1300	15700
Power (W)	5	5	10	7.5	30	300
GFLOPS/W	6.4	65.2	51.1	**88.7**	43.3	52.3

Table 2.1: Comparison of some key aspects of the hardware we used.

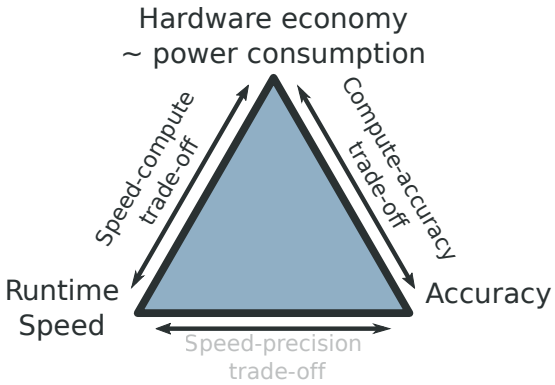


Figure 2.7: Playing with the power consumption of a system changes the trade-offs that are influenced by the hardware constraints. More power can go into either more speed, or better accuracy. Depending on the hardware platform a better trade-off can be made.

return results faster. As we explained in section 1.2.1, adding more parallel hardware (e.g. by using a different platform), does not however always have this immediate result of a faster system.

Likewise, when we are given more compute power we can often also increase the accuracy of the algorithm. More parallel hardware might mean that we can run a deep network, use more samples, or use better precision floating-point operations. By increasing clock speed, we can go via the speed constraint on the triangle to the speed accuracy trade-off, and trade speed for accuracy. We might for example do more iterations in the algorithm to get a more precise result.

2.3 Conclusion

In this chapter we looked at different hardware used during this project. We used many types of hardware that where either constrained in terms of performance (section 2.2) or programming model (section 2.1). In general, we can say that the right device depends on the type of program at hand: complex algorithms requiring a lot of computing power will have a hard time running on low power systems, while low power systems can be the only option in certain applications. Dedicated accelerators like GPUs might be necessary if problems require a huge amount of computation. Although it has become way better with more high

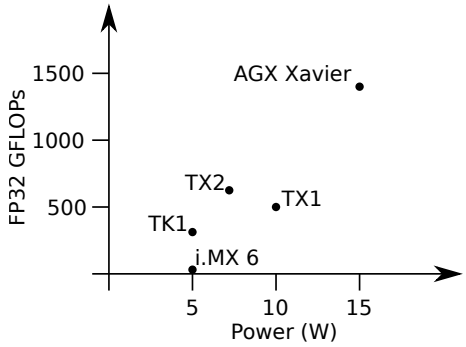


Figure 2.8: Comparison between power and computing power of the embedded devices we used.

level libraries, for some novel applications (custom applications where no pre-made library exists), these devices do still bring a more difficult programming paradigm. GPU programming is however not the end of the story. For even more efficiency it might be a good idea to use programmable hardware (FPGAs) or even hardware solutions (ASICs). This does off course come at an even higher development cost. In [135], the authors compare the GPU development cost to the cost of FPGA development.

Throughout this thesis the hardware constraint will come back as a very important parameter in developing an application that meets all requirements.

Chapter 3

An OpenCL implementation of a forward sampling algorithm for CP-logic

The core of this thesis consists of a number of cases that highlight the different aspects of implementing algorithms from the field of AI in the real world. This first case is about an implementation of a forward sampling algorithm for CP-logic on the GPU. We present an approximate query answering algorithm for the Probabilistic Logic Programming language CP-logic. It complements existing sampling algorithms by using the rules from body to head instead of in the other direction. We present an implementation in OpenCL, which is able to exploit the multicore architecture of modern GPUs to compute a large number of samples in parallel, and demonstrate that this is competitive with existing inference algorithms.

This work was previously published in the International Journal of Approximate Reasoning [150] and the proceedings of Probabilistic Logic Programming workshop [149].

3.1 Introduction

Probabilistic graphical models, such as Bayesian networks [106], have enjoyed great success in a variety of applications. However, their limited expressiveness

makes certain patterns of dependencies between random variables difficult to express, and may also lead to models that are too unwieldy to interpret or maintain. In an effort to improve on this, the area of Probabilistic Logic Learning (PLL) leverages concepts and methods from Logic Programming (and/or other logical formalisms) to develop languages with better knowledge representation properties. In other words, the goal is to make models that are more readable, more compact and more modular. Unlike the propositional representation offered by Bayesian networks, PLL models tend to be first-order, using the power of logical variables and logical formulas to describe the relations between a large number of random variables by a comparatively small number of expressions.

An important family of PLL languages has originated from the Distribution Semantics, originally developed by Sato [125]. Examples of languages in this family are Sato's own PRogramming In Statistical Modeling (PRISM) language [125], the Independent Choice Logic [109] and CP-logic [154]. Among these languages, CP-logic offers the most general syntax and will therefore be our focus in this chapter, even though our results could apply equally to other languages in this family. In all of these languages, a user constructs a set of Logic Programming-like rules, which contain a number of probabilistic parameters. The precise value of these parameters can be supplied by the user or learned from data. Once their values are known, the model can then be used for a variety of inference tasks, such as prediction or explaining observations. A core inference task, which is needed in many applications and also plays a key role in the Machine Learning algorithms that are typically used to estimate the probabilistic parameters, is that of computing the probability of a query. Because, for instance, the Machine Learning (ML) algorithms need to perform this task numerous times, an efficient implementation is crucial. Here, the additional expressivity of PLL languages may prove problematic.

Different algorithms already exist for computing the probability of a query in CP-logic. The Problog system (`http://dtai.cs.kuleuven.be/problog`) supports the CP-logic language with algorithms based on weighted CNFs [43] and Binary Decision Diagrams [73]. A separate family of inference algorithms for CP-logic [119,122] is based on more traditional Logic Programming methods such as SLG resolution. One of these, the Probabilistic Inference with Tabling and Answer subsumption (PITA) algorithm for CP-logic, is currently part of the XSB Prolog system (`http://xsb.sourceforge.net`).

Both families of algorithms compute the exact probability of a query. However, because the high computational complexity of this task can be problematic for real applications, several approximate algorithms have also been developed. One possibility [73,118] is to compute only a subset of all proofs of the query, as opposed to computing all proofs to produce an exact probability. An alternative

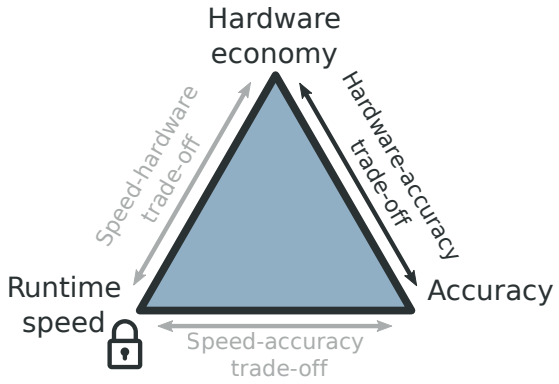


Figure 3.1: In the application of making a live video montage we are constraint in runtime-speed.

is the Monte Carlo algorithm MCINTYRE [120], which repeatedly samples an SLD proof tree of the query. All these methods use the rules of a CP-logic theory in a backwards way, i.e., going from the head to the body.

In this chapter, we propose an alternative sampling algorithm, which uses the rules in a forwards way, from body to head. As we will argue, this is a useful complement to the backwards methods. Moreover, because different samples are sampled independently, our method can easily be parallelised. We demonstrate this by developing an implementation that runs in parallel on GPU hardware.

While this work does not have a particular application in mind, many applications do exist. An example of this is automatically making video montages using cinematographic rules. Work from my colleague Bram Aerts [8], used probabilistic logic programming to automatically decide which video feed to select in a movie scene with multiple feeds to create a pleasing result. This system could also be used for live video streams, in which case fast inference is crucial.

This is an example of an application that is constrained in time. The systems needs to deliver results in time to make a decision before it is too late. Using the triangle-framework introduced in chapter 1, we set "runtime-speed" as a hard constraint, which leaves a trade-off between hardware and accuracy (figure 3.1). We *explore* this trade-off by finding the best setting between compute power and accuracy, and also *optimize* the algorithm to get better accuracy at the cost of no more hardware, by using it more efficiently. How we make this trade-off and optimizations will become clear in this chapter.

3.2 Preliminaries from Logic Programming

Before recalling the definition of CP-logic itself, we briefly summarise some relevant concepts from (non-probabilistic) Logic Programming. A *vocabulary* Σ consists of a set of predicate symbols and function symbols, each with an associated arity. Function symbols with an arity zero are called *constants*. Throughout this article, we consider only *relational* vocabularies in which all function symbols are constants. A *term* is either a constant or a variable $x, y, z, \ldots$ drawn from some infinite set of available variables. An *atom* is an expression $P(t_1, \ldots, t_n)$ where P is a predicate symbol of arity n and the t_i are terms. Formulas are built from atoms using the standard connectives $\wedge, \vee, \neg, \forall, \exists$.

A Logic Program is a finite set of rules r of the form:

$$\forall \vec{x}\ A \leftarrow \phi$$

where A is an atom (denoted $head(r)$), ϕ a formula (denoted $body(r)$) and $\vec{x}$ a tuple of variables that contains all free variables of A and ϕ. It is common to restrict ϕ to a conjunction of literals. A rule is *ground* if it does not contain any variables. In the absence of function symbols of arity > 0, each program can be transformed to a (finite) ground program by replacing each variable by all available constants in all possible ways, turning a single rule (with a non-empty tuple $\vec{x}$) into a set of rules (with empty $\vec{x}$), turning $\forall$ into $\wedge$, and turning $\exists$ into $\vee$. The resulting ground program is called the *grounding* of the original program.

An *(Herbrand) interpretation* I for a vocabulary Σ assigns to each n-ary predicate P a set P^I of n-tuples of constants. Such an interpretation can equivalently be viewed as the set of all ground atoms $P(\vec{c})$ for which $\vec{c} \in P^I$. For an atom A, we write $I \models A$ when $A \in I$. By the usual induction over the logical connectives (e.g., $I \models \phi \wedge \psi$ if both $I \models \phi$ and $I \models \psi$) this notion is extended to arbitrary formulas in vocabulary Σ.

The semantics of a *positive* logic program (i.e., one in which no negation $\neg$ appears) is uncontroversially defined by means of a least fixpoint construction. For a program P, we define the operator T_P which maps each interpretation I to the interpretation $I' = \{head(r) \mid r \in P \text{ and } I \models body(r)\}$. The *model* of P is then the least fixpoint of this operator T_P, which can be constructed by iteratively applying T_P to the empty interpretation $\{\}$.

For programs containing negation, different semantics exist. CP-logic makes use of the well-founded semantics [143], for reasons explained elsewhere [154]. This semantics uses three-valued interpretations $\mathcal{I}$, in which atoms and formulas can be **Unknown** in addition to **True** or **False**. Such a three-valued interpretation

ϕ	ψ	$\phi \wedge \psi$
True	**True**	**True**
True	**Unknown**	**Unknown**
True	**False**	**False**
Unknown	**Unknown**	**Unknown**
Unknown	**False**	**False**
False	**False**	**False**

ϕ	$\neg\phi$
True	**False**
Unknown	**Unknown**
False	**True**

Figure 3.2: Kleene truth tables.

$\mathcal{I}$ corresponds in a natural way to a pair of two-valued interpretations (I, J) with $I \subseteq J$: the atoms $A \in I$ are those for which $A^{\mathcal{I}} = \mathbf{True}$; the atoms $A \notin J$ are those for which $A^{\mathcal{I}} = \mathbf{False}$; and the remaining atoms $A \in J \setminus I$ are those for which $A^{\mathcal{I}} = \mathbf{Unknown}$. We say that a three-valued interpretation $\mathcal{I}$ is *less precise* than $\mathcal{I}'$, denoted $\mathcal{I} \leq_p \mathcal{I}'$, if whenever $A^{\mathcal{I}'} = \mathbf{Unknown}$ also $A^{\mathcal{I}} = \mathbf{Unknown}$, and whenever $A^{\mathcal{I}'} \neq \mathbf{Unknown}$ then $A^{\mathcal{I}'} = A^{\mathcal{I}}$. Viewing $\mathcal{I}$ and $\mathcal{I}'$ as pairs (I, J) and (I', J') of two-valued interpretations, this corresponds to $I \subseteq I'$ and $J' \subseteq J$.

Using the standard Kleene truth tables (see Fig. 3.2) the three-valued truth assignment to atoms can again be extended to arbitrary formulas. When the three-valued interpretation $\mathcal{I}$ is viewed as a pair (I, J) of two-valued interpretations, this truth assignment $\phi^{\mathcal{I}}$ can also be characterised in a different way. For a formula ϕ, denote by $\phi^{(I,J)}$ the result of evaluating all positive occurrences of atoms (i.e., those within the scope of an even number of negations) in I, all negative occurrences of atoms in J, and then simply applying the usual induction over the logical connectives. Then $\phi^{\mathcal{I}} = \mathbf{True}$ if and only if both $\phi^{(I,J)} = \mathbf{True}$ and $\phi^{(J,I)} = \mathbf{True}$; $\phi^{\mathcal{I}} = \mathbf{False}$ iff both $\phi^{(I,J)} = \mathbf{False}$ and $\phi^{(J,I)} = \mathbf{False}$; and $\phi^{\mathcal{I}} = \mathbf{Unknown}$ iff $\phi^{(I,J)} = \mathbf{False}$ but $\phi^{(J,I)} = \mathbf{True}$. (Note that because $I \subseteq J$, it is impossible that $\phi^{(I,J)} = \mathbf{True}$ and $\phi^{(J,I)} = \mathbf{False}$.)

For a given interpretation J, we can define an operator $U_P(\cdot, J)$ which maps each I to the interpretation $I' = \{head(r) \mid r \in P \text{ and } body(r)^{(I,J)} = \mathbf{True}\}$. Such an operator $U_P(\cdot, J)$ interprets all negative occurrences of atoms by the fixed interpretation J and acts like the standard T_P operator on the remaining, positive occurrences. As with T_P, we are interested in the least fixpoint of $U(\cdot, J)$. By C_P, we denote the operator that maps each interpretation J to this least fixpoint, i.e., $C_P(J) = lfp(U(\cdot, J))$.

Using this C_P, we can now define the well-founded model $WFM(P)$ of P as follows. An *alternating fixpoint* of C_P is a pair (I, J) of interpretations such that $C_P(I) = J$ and $C_P(J) = I$. The well-founded model of P is the least

precise alternating fixpoint of C_P.

3.3 Preliminaries: CP-logic

A theory in CP-logic is a finite set of *CP-laws* of the form:

$$\forall \vec{x}\ (A_1 : \alpha_1) \vee \cdots \vee (A_n : \alpha_n) \leftarrow \phi.$$

Here, ϕ is a formula of first-order logic such that the free variables of ϕ belong to $\vec{x}$ and the A_i are atoms, such that the tuple of variables $\vec{x}$ contains all free variables in ϕ and the A_i. The α_i are non-zero probabilities with $\sum \alpha_i \leq 1$. Such a rule expresses that ϕ causes some (implicit) non-deterministic event, of which each A_i is a possible outcome with probability α_i. If $\sum_i \alpha_i = 1$, then at least one of the possible effects A_i must result if the event caused by ϕ happens; otherwise, the event may happen without any (visible) effect on the state of the world. For a CP-law r, we refer to ϕ as $body(r)$, and to the sequence $(A_i, \alpha_i)_{i=1}^n$ as $head(r)$. The body may be omitted for events that are vacuously caused.

The semantics of a theory in CP-logic is defined in terms of its grounding, so from now on we will restrict attention to ground theories. Moreover, we assume without loss of generality that the sum $\sum_i \alpha_i$ of all probabilities in any $head(r)$ is precisely 1, since this can always be achieved by adding a fresh atom as an additional disjunct. A final assumption, made for simplicity, is that we assume that each $body(r)$ is a conjunction of literals.

Example 1. *Suzy and Billy may each decide to throw a rock at a bottle. Suzy throws with probability* 0.5 *and if she does, her rock breaks the bottle with probability* 0.8*. Billy always throws and his rock hits with probability* 0.6*.*

$$(Throws(Suzy) : 0.5). \tag{3.1}$$

$$(Throws(Billy) : 1). \tag{3.2}$$

$$(Broken : 0.8) \leftarrow Throws(Suzy). \tag{3.3}$$

$$(Broken : 0.6) \leftarrow Throws(Billy). \tag{3.4}$$

This theory consists of two vacuous laws and two conditional laws. The second law is deterministic, i.e., it always has the same effect.

The semantics of CP-logic can be defined as a straightforward instantiation of Sato's distribution semantics [124]. An *instance* of a CP-logic theory T is a logic program that is constructed by choosing a single atom A_i from the head

of each rule r of T. In other words, if we denote this chosen atom by $\sigma(r)$, then the instance T^σ is the logic program that consists of the rules:

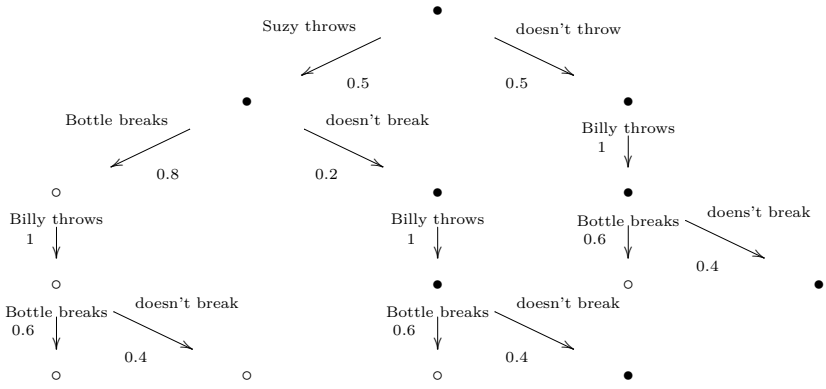
$$\sigma(r) \leftarrow body(r), \qquad \text{for all } r \in T.$$

To each such instance, we associate a probability π_σ, which is the product $\Pi_{r \in T} \alpha_r$, where each α_r is the probability of $\sigma(r)$ in $head(r)$. This probability distribution over instances is then mapped to a probability distribution over interpretations by applying the well-founded model semantics to these logic programs. In other words, we define a probability distribution π_T over interpretations I as:

$$\pi_T(I) = \sum_{\sigma \text{ s.t. } WFM(T^\sigma)=I} \pi_\sigma. \tag{3.5}$$

This semantics is only well-defined for CP-theories which have the property that each instance T^σ has a two-valued well-founded model. Therefore, CP-logic only allows such theories. This can be ensured by imposing a syntactic condition such as *stratification*.

In [154], an alternative—but equivalent—characterization of the CP-logic semantics was developed. Following the work of Shafer [129], it makes use of probability trees. In particular, the concept of an *execution model* of a CP-theory is defined. This is a probability tree in which each node s is labelled with an interpretation $I(s)$. Such trees are constructed, starting from a root node s_0 in which all atoms are false (i.e., $I(s_0) = \{\}$), by "firing" rules whose body holds. The following is an execution model for Example 1. States s in which the bottle is broken (i.e., $I(s) \models Broken$) are represented by an empty circle, and those in which it is still whole by a full one.



To each branch b of such a tree corresponds a set of instances of the theory, namely all those which make the same choices as b for all the rules that actually fire in b. For the rules that do not fire in b, it does not matter which choices the instances make. As shown in [154], the sum of the probabilities of all these instances is precisely the probability of the branch, and they all have the interpretation $I(l)$ associated with the leaf l of the branch as their WFM. For positive theories, i.e., those without negation, this property is rather obvious. In the general case, however, some additional care is required when constructing the execution models. We discuss this in Section 3.5.

A CP-theory may have many execution models, which differ in the order in which they fire rules. The differences between these trees are irrelevant, in the sense that they all produce the same probability distribution in the end, namely π_T as defined by equation (3.5) [154].

3.4 A forward sampling method for positive CP-theories

The probability tree characterization of the semantics of CP-logic suggests a straightforward sampling method. To approximate the probability $\pi_T(Q)$ of a query Q, we can simply perform repeated random walks in the tree and report the ratio $\tilde{p}$ of how often we end up in a leaf s such that $I(s) \models Q$ versus the total number of walks. The more walks we take, the more precise this estimate.

To turn this idea into a concrete algorithm, we must decide which of the execution models of the theory will be used for the random walks. For efficiency reasons, we construct the tree by first building a list of all rules that are applicable in a given node, and then firing all of these rules in sequence, thereby traversing a number of nodes in which we do not have to evaluate any rule bodies. This results in Algorithm 1, which produces a single random walk when called with the CP-theory T as its first argument and the interpretation $\{\}$ that assigns **False** to all atoms as the second.

At the end of this procedure, the interpretation I is saturated, in the sense that all rules $r \in T$, such that $I \models body(r)$, have been fired, and therefore I contains at least one true atom from each such $head(r)$.

The following theorem proves the correctness of this procedure.

Theorem 1. *Let T be a positive CP-theory and let F be the interpretation that assigns false to all atoms. Let $\tilde{P}_T$ be the probability distribution over all possible runs r of the probabilistic Algorithm 1 when called as* `Saturate`(T, F). *For each*

Algorithm 1 saturates a given interpretation I by applying rules from a set R

1: **procedure** SATURATE(T, I) ▷ Updates I
2: $T' \leftarrow T$
3: $Fired \leftarrow \{\}$
4: **repeat**
5: $ToFire \leftarrow \{r \in T' \mid I \models body(r)\}$
6: **for all** $r \in ToFire$ **do**
7: $a \leftarrow RandomChoice(head(r))$
8: $I(a) \leftarrow$ **True**
9: $T' \leftarrow T' \setminus ToFire$
10: **until** $ToFire = \{\}$
11: **function** RANDOMCHOICE($(a_1, \alpha_1), \ldots, (a_n, \alpha_n)$)
12: $r \leftarrow$ random floating point number $\in [0, 1]$
13: $sum \leftarrow 0$
14: **for all** $i \in [1, n]$ **do** ▷ If $\sum_{j<i} \alpha_j \leq r \leq \sum_{j \leq i} \alpha_j$, then return a_i
15: $sum \leftarrow sum + \alpha_i$
16: **if** $r \leq sum$ **then**
17: **return** a_i

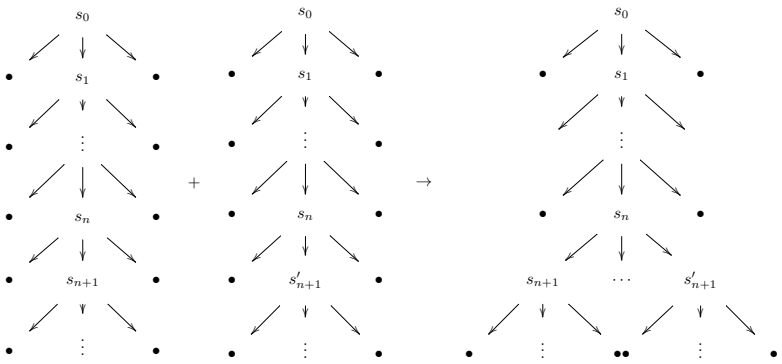
possible run r, let n^r be the number of iterations of the inner for-loop (line 6), and, for each $1 \leq i \leq n^r$, let I_i^r be the value of I at the start of the ith iteration of this loop. Then there exists an execution model χ of T with a one-to-one mapping f between the branches of χ and the possible runs of the algorithm, such that, for each branch $b = (s_1, \ldots, s_{n^r})$ of χ and the corresponding run $r = f(b)$, the probability of b is equal to $\tilde{P}_T(r)$ and $I(s_i) = I_i^r$ for all $1 \leq i \leq n^r$.

Proof. For a run r, we construct part of an execution model containing a branch b such that $f(b) = r$ as follows. We start from a root node s_0 and assign $I(s) = F$ (as must be the case for a root node in an execution model). We iteratively extend the node s_i by firing the rule r of line 6: if the head of r is $((a_1, \alpha_1), \ldots, (a_n, \alpha_n))$, then the children of s_i are nodes s_i^j, for $j \in 1..n$, with $I(s_i^j) = I(s_i) \cup \{a_i\}$ and the probability of the edge from s_i to s_i^j is α_j. The node s_i^j such that RANDOMCHOICE($head(r)$) returned a_j is then chosen as s_{i+1}, and this construction is repeated. It is obvious by induction that, for each i, $I(s_i) = I_i^r$. Moreover, the probability tree thus constructed can be extended to an execution model by applying the usual construction process to those nodes that do not occur in the sequence $(s_i)_i$.

We now have established a one-to-one mapping f between runs and branches of a number of different (potentially as many as there are runs) execution models

of T . What remains to be shown is that there exists a *single* execution model χ such that the branches of χ are precisely all $f(r)$ for which r is a possible run of the algorithm.

Whenever two runs r, r' of the algorithm coincide on the first n probabilistic choices returned by RandomChoice, then the initial fragments $(s_0, \ldots, s_n)$ and $(s'_0, \ldots, s'_n)$ of the sequences $(s_i)_i$ and $(s'_i)_i$ constructed in the preceding paragraph will also be identical. It follows that we can merge the two corresponding partial probability trees into a single probability tree:



(3.6)

In this picture, the bullets denote nodes that are constructed but not expanded, because they do not belong to the sequence $(s_i)_i$ ($(s'_i)_i$ respectively). Moreover, n is the maximal number for which the initial fragments up to n of $(s_i)_i$) and $(s'_i)_i$ coincide. It follows that we can merge all branches $f(r)$ for which r is a possible run into a single probability tree χ. By construction, this single probability tree is a complete execution model of the theory. The fact that the probabilities with which RandomChoice returns a particular atom are the same as the labels α_i placed on the corresponding edge of this execution model concludes the proof.

□

With this result in hand, we can compute the probability of a given query by repeatedly sampling this distribution. This results in the following Algorithm 2.

So far, we have only considered positive CP-logic theories. When negation is allowed to appear, some more work is needed.

Algorithm 2 approximates the probability of a query Q according to π_T

```
1: function SAMPLE(T, Q)
2:     q ← 0, n ← 0 ▷ Nb of samples in which Q holds and total nb of samples
3:     repeat
4:         I ← the interpretation such that all atoms are False
5:         Saturate(T, I)
6:         if I ⊨ Q then q ← q + 1
7:         n ← n + 1
8:     until desired accuracy reached                    ▷ See Section 3.6.1
9:     return q/n
```

3.5 A forward sampling method for general CP-theories

Negation in a CP-theory is interpreted by the WFM semantics. In [154], the execution model semantics was also extended to this case. Because the WFM uses three-valued interpretations, each node s in an execution model is now also labeled with a three-valued interpretation $\mathcal{I}(s)$, in addition to the two-valued $I(s)$. The true atoms of $\mathcal{I}(s)$ and $I(s)$ are always the same, but false atoms of $I(s)$ may be unknown in $\mathcal{I}(s)$. At the root r of the tree, $\mathcal{I}(r)$ assigns **Unknown** to all atoms.

The key difference with the positive case is that rules are now only allowed to fire if their body is true in both $I(s)$ and in $\mathcal{I}(s)$ (in particular, rules which are true in $I(s)$ but unknown in $\mathcal{I}(s)$ may not fire). The effect of firing a rule is the same as before: one atom from the head of the rule becomes true, both in $I(s)$ and $\mathcal{I}(s)$. In this way, an atom that was originally unknown in some $\mathcal{I}(s)$ may become true in a child $\mathcal{I}(s')$. There is also a way in which atoms that are originally unknown may become false. This is done by a "look ahead"-mechanism, that makes atoms false when there is no longer any way in which they could still be caused. The details are as follows.

The three-valued interpretation $\mathcal{I}(s)$ in a node s is not derived from the three-valued interpretation $\mathcal{I}(s')$ associated to the parent s' of s, but it is instead constructed from the two-valued interpretation $I(s)$. In particular, $\mathcal{I}(s)$ is defined as the limit of a sequence $(\mathcal{I}'_i)_{i\geq 0}$ of three-valued interpretations. This sequence starts from the interpretation $\mathcal{I}'_0$ that coincides completely with $I(s)$ (and therefore has no unknown atoms). Given a $\mathcal{I}'_i$, we then select a rule r that has not yet fired and whose body is either **True** or **Unknown** in $\mathcal{I}'_i$. We derive $\mathcal{I}'_{i+1}$ from $\mathcal{I}'_i$ by changing the truth value of all atoms in the head of r that are still **False** to **Unknown**. In this way, we end up making some of

the atoms that are **False** in $I(s)$ **Unknown**, but not (necessarily) all of them. $\mathcal{I}(s)$ is then defined as the limit of this sequence. It can be shown that, by constructing $\mathcal{I}(s)$ in this way, an overestimate is obtained of the atoms that still might be caused in a descendant of s: atoms that are **False** in $\mathcal{I}(s)$ are also false in $I(l)$ for any leaf l that can be reached from s. Relating this construction to the concepts of Section 3.2, it was shown in [154] that in the special case of deterministic CP-theories, this $\mathcal{I}(s)$ is in fact equal to $C_T(I(s))$. From this result, it follows that, also for theories containing negation, the execution model characterisation of the semantics coincides with the Sato-style distribution semantics characterisation of equation (3.5).

Even though the three-valued interpretation $\mathcal{I}(s)$ is constructed from $I(s)$, and not from the three-valued interpretation $\mathcal{I}(s')$ of the parent s' of s, it can nevertheless can be shown that $\mathcal{I}(s)$ is always more precise than $\mathcal{I}(s')$, in the sense that its true and false atoms, respectively, are a superset of those of $\mathcal{I}(s')$. This allows us to postpone the expensive operation of actually computing $\mathcal{I}(s)$: if the body of a rule r is true in $\mathcal{I}(s'')$, for some ancestor s'' of s, then we can be sure that it is still true in $\mathcal{I}(s)$. Therefore, we can safely fire r, thus proceeding to a child of s, without ever having to compute $\mathcal{I}(s)$ at all.

A further improvement on this is that it is also safe to fire a rule r in a child s of s' even if its body is not true in $\mathcal{I}(s')$ itself, but is true in the three-valued interpretation $\mathcal{I}'$ that we can construct from $\mathcal{I}(s')$ by switching the atom a that was chosen on the transition from s' to s from **Unknown** to **True** (in other words, $\mathcal{I}'$ has the same **False** atoms as $\mathcal{I}(s')$ and the same **True** atoms as $I(s)$). This is because the real $\mathcal{I}(s)$, that we are trying to avoid having to construct, must also be more precise that this $\mathcal{I}'$, and therefore all rule bodies that are true in $\mathcal{I}'$ must also be true in $\mathcal{I}(s)$.

This leads us to Algorithm 3. Here, we are now calling the procedure `Saturate` with a three-valued interpretation as its argument. In this case, the expression $I \models body(r)$ in Line 13 of Algorithm 1 has to be interpreted as that the truth value of $body(r)$ has to be **True** according to I (i.e., **Unknown** is not enough).

Theorem 2. *For a CP-logic theory T that may contain negation, the same correspondence holds between runs of the non-deterministic Algorithm 3 and branches of an execution model of T, as formulated in Theorem 1 for Algorithm 1 and execution models of a positive CP-logic theory.*

Proof. The procedure SHRINK of Algorithm 3 is a straightforward implementation of the construction process used to define the three-valued interpretation $\mathcal{I}(s)$. From this, together with Theorem 1, the result immediately follows. □

Algorithm 3 samples single branch of execution model of T, returning the leaf

function SINGLESAMPLE(T)
  $\mathcal{I} \leftarrow$ interpretation that maps each atom appearing in T to **Unknown**
  repeat
    $Saturate(T, \mathcal{I})$ ▷ Atoms go from **Unknown** to **True**
    $Shrink(T, \mathcal{I})$ ▷ Atoms go from **Unknown** to **False**
  until fixpoint reached
  return $\mathcal{I}$
function SHRINK($T, \mathcal{I}$) ▷ updates $\mathcal{I}$ by falsifying atoms
                                ▷ that can no longer be derived
  for all atoms a such that $\mathcal{I}(a) =$ **Unknown do**
    $\mathcal{I}(a) \leftarrow$ **False**
  repeat
    $ToFire \leftarrow \{r \in T \mid body(r)^{\mathcal{I}} \neq \textbf{False}\}$
    for all atoms a in some $head(r)$ with $r \in ToFire$ **do**
      $\mathcal{I}(a) \leftarrow$ **Unknown**
    $T \leftarrow T \setminus ToFire$
  until $ToFire = \{\}$

3.6 OpenCL implementation

To compute a good approximation of the probability of a query, our sampling algorithm may need a large number of samples. Because different samples are sampled independently, the sampling process can be executed in parallel to achieve a significant speed-up. We demonstrate this in OpenCL.[1]., a recent framework for programming parallel hardware (typically, but not exclusively, GPUs) in a platform-independent way.

3.6.1 Our algorithm in OpenCL

To compute n samples, we create n work items, each running a kernel consisting of Algorithm 3. To make this possible, the host code first copies the CP-theory T to the global memory of the OpenCL device. It then also allocates the data structures that the work items need for their operation: an array `atoms` in which they can store the truth value of each atom, initialized to **Unknown**; an array `rules` in which they can store the truth value of each rule body (**True**, **False**, **Unknown** or "already fired"), also initialized to unknown; and a sequence of random numbers for the required random decisions. These arrays are all in global memory, because they grow too large for local/private memory in

[1] `http://www.khronos.org/opencl/`

many of our benchmarks. Below, we discuss a number of ways in which our implementation does make use of local/private memory.

When a work item finishes, it also leaves its end result in global memory. Once all work items have finished, the host copies these individual samples back to main memory and joins them together in order to answer the query. To improve the performance of this basic OpenCL algorithm, we can apply the following optimisations.

MEM: Each work item $w_1, \ldots w_n$ needs its own array $(a_1^{w_i}, \ldots, a_m^{w_i})$ in which to store the truth value of the atoms $(a_1, \ldots, a_m)$. Instead of storing these arrays as $\big((a_1^{w_1}, \ldots, a_m^{w_1}), (a_1^{w_2}, \ldots, a_m^{w_2}), \ldots, (a_1^{w_n}, \ldots, a_m^{w_n})\big)$, it is more efficient to store them as $\big((a_1^{w_1}, \ldots, a_1^{w_n}), (a_2^{w_1}, \ldots, a_2^{w_n}), \ldots, (a_m^{w_1}, \ldots, a_m^{w_n})\big)$. In this way, coalesced memory access allows all work items to access their own truth value for the same atom a_i in a single transaction.

INI: Initializing the arrays `atoms` and `rules` can be done as the first operation of the kernel, to avoid copying two large arrays of **Unknown** values from host to device memory.

PRI: The array `atoms` can be cached in private memory to reduce the time spent accessing global memory.

RAN: Instead of generating random numbers on the host and transferring them to the OpenCL device, we can also run a random number generator on the OpenCL device, so that we only have to transfer a single seed. We will use the MWC64X OpenCL random number generator.[2]

RED: Instead of copying the results of the individual samples back to the host, we can run a second kernel `Reduce` on the OpenCL device to aggregate the results there, so only the final count has to be transferred back. This is done using the standard method of iteratively reducing n results of the sampling algorithm to $n/2$ results by having $n/2$ work-items that each combine 2 results. The intermediary data structures needed for this are stores in local memory.

CHU: While the OpenCL device is constructing samples, the host is idling. To avoid this, we can opt for a chunking strategy, in which we compute our n samples in chunks of $m \ll n$ at a time. While the OpenCL device is computing a chunk of samples, the host can transfer the results of the previous chunk back to host memory and aggregate them. Because this is a relatively cheap operation, it should be finished before the ith chunk of samples is done, so the host should be ready in time to initiate the

[2] `http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html`

next chunk. In this way, we no longer need a separate `Reduce` kernel and keep the OpenCL device continually busy generating samples. A possible downside of this approach is that we now incur multiple times the overhead of starting a kernel on the OpenCL device and waiting for it to finish.

All of the above optimisations can be combined, with the exception of the mutually incompatible RED and CHU. In addition, we also experimented with caching part of the theory (in particular, the heads of rules) in local memory. However, due to limited size of local memory this was only possible for very small benchmarks. Moreover, it did not significantly improve performance. We have therefore not considered this further.

Instead of running a fixed number of iterations, we can also stop the algorithm once a desired accuracy has been reached. This is particularly useful in combination with the optimisation CHU, because we can then check the accuracy after each chuck. The Problog Monte Carlo algorithm [73] uses a stopping criterion based on the central limit theorem:

$$2z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{k}} \leq \delta,$$

where $\hat{p}$ is the measured probability of the query, z_x is the x percentile of a standard normal distribution, k is the number of samples and δ is the desired width of the confidence interval. The MCINTYRE algorithm uses the same criterion, with the additional requirement that $k\hat{p} > 5$ and $k(1-\hat{p}) > 5$ to ensure that approximation provided by the central limit theorem is trustworthy. We will use the same criterion with the same parameters of $\alpha = 0.05$ (so $z_{1-\alpha/2} = 1.96$) and $\delta = 0.01$. With these parameters, the worst case value for k (reached for $\hat{p} = 0.5$) is 38416.

3.7 Experimental results

Source code and graphs of the experiments described in this section can be downloaded from:

`http://www.cs.kuleuven.be/~joost/PLL/`

All our experiments were run on a Linux machine with an Intel Core i7 965 CPU (3.20GHz) and an NVIDIA GeForce GTX 295 (GT200) GPU. This GPU has 896 MiB of global memory and consists of 30 streaming multiprocessors,

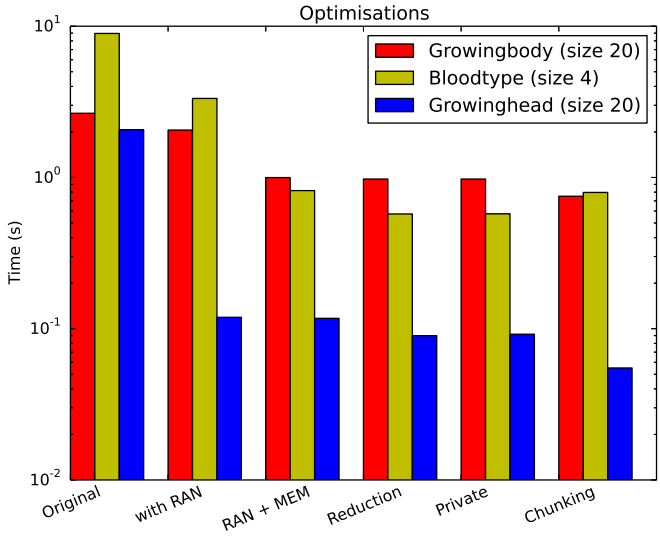


Figure 3.3: Effect of different optimisations executed on the GTX295 GPU.

each with 16 KiB of local memory and consisting of 8 individual cores operating at 1242 MHz. When executing OpenCL code, each work item occupies a single core, with work items in the same work group running on the same streaming multiprocessor. Each core has an additional 8 KiB of private memory available. Our experiments use the BLOODTYPE, GROWINGHEAD and GROWINGBODY benchmarks from [95]. The latter two benchmarks consist of ground programs. However, the BLOODTYPE benchmark first needs to be grounded before we can apply our algorithm.

Figure 3.3 investigates the effect of the different optimisations proposed in the previous section. As a baseline, we also include a plain C version of our algorithm, that runs on a single CPU core and computes the required number of samples in a sequential `for`-loop. We compare this baseline to the following variants of our OpenCL implementation on GPU.

Name	Applied optimisations
GPU original	—
RAN	RAN
RAN+MEM	RAN, MEM
Reduction	RAN, MEM, RED
Private	RAN, MEM, RED, PRI
Chunking	RAN, MEM, CHU, PRI

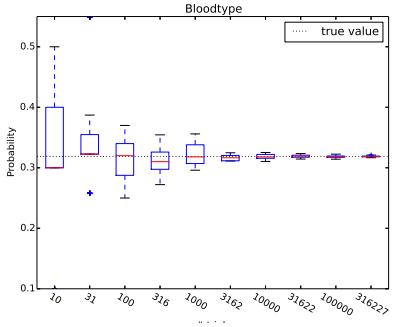
Recall that the optimisations CHU and RED are incompatible. Therefore, the final two variants are two alternative ways of fully optimising the code. For each version, we have run a single instance of the three benchmarks.

Figure 3.3 shows first of all that relatively little is gained by simply porting the CPU code to GPU in a naive way. Indeed, for one of the benchmarks, the additional overhead of copying data and starting an OpenCL kernel even leads to a slow-down instead of a speed-up. This figure also shows that the optimisations have a significant impact. In general, more optimisations are better, even though it is not always the same optimisation that leads to the biggest gain. Of the two fully optimised variants, *Chunking* outperforms *Private* on two of the three benchmarks. When comparing it to *GPU original*, the smallest gain is observed in the GrowingBody benchmark, where *Chunking* is only about 4× faster. For Bloodtype and GrowingHead, the speedup is 15× and 37×, respectively. When compared to the C version on a single CPU core, the speedups are, respectively, 19×, 22× and 10×. We conclude that use of the GPU and our optimisations is indeed worthwhile. Our remaining experiments will be conducted using the *Chunking* variant.

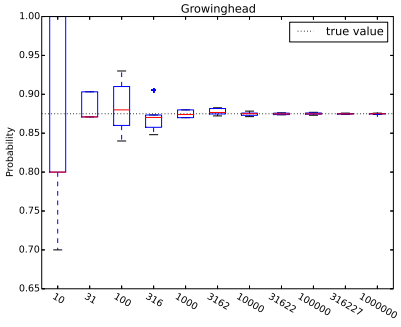
The convergence behavior of our algoritm is examined in Figures 3.4a, 3.4b, 3.4c. These figures show boxplots for the probability estimates produced by 10 different runs of our algorithm, each time using a fixed number of samples. The benchmarks used here are the same as before. As these results show, even a relatively small number of samples can already produce reasonably good approximations of the true probability. This is most pronounced for Bloodtype, where 10 samples already provide a very good estimate (even though the stopping criterion of Section 3.6.1 would run until about 350 samples). For GrowingBody, the stopping criterion suggests about 17 000 samples, which is indeed about where the results of the algorithm begin to seem accurate. The same can be said for GrowingHead, where the stopping criterion suggests about 30 000 samples.

Finally, we compare our algorithm to existing query answering methods for CP-logic:

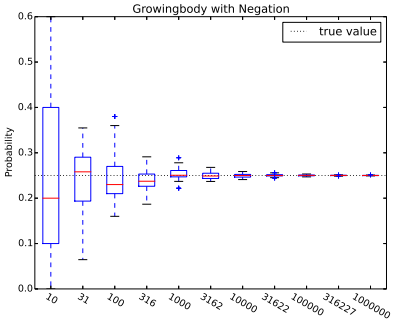
- The approach of [94], which transforms CP-logic into a Bayesian network



(a) BLOODTYPE



(b) GROWINGHEAD



(c) GROWINGBODY

Figure 3.4: Convergence of the algorithm.

and then applies the Variable Elimination (VE) algorithm or Contextual Variable Elimination (CVE) [95]

- The default and Monte Carlo inference algorithms of the ProbLog system [73]
- The PITA algorithm [121] that is built into XSB Prolog
- The MCINTYRE backwards sampling algorithms [120].

For the k-best [73] and k-optimal [118] algorithms, we were unable to run the most recent version (in YAP 6.3.3) on our system, even after contacting the authors. We were able to run an older version of k-best (in YAP 6.2.2), but this does not support negation of arbitrary atoms and was therefore unable to handle the GROWINGBODY benchmark. The k-optimal algorithm is not available in this version and could not be included in our experiments.

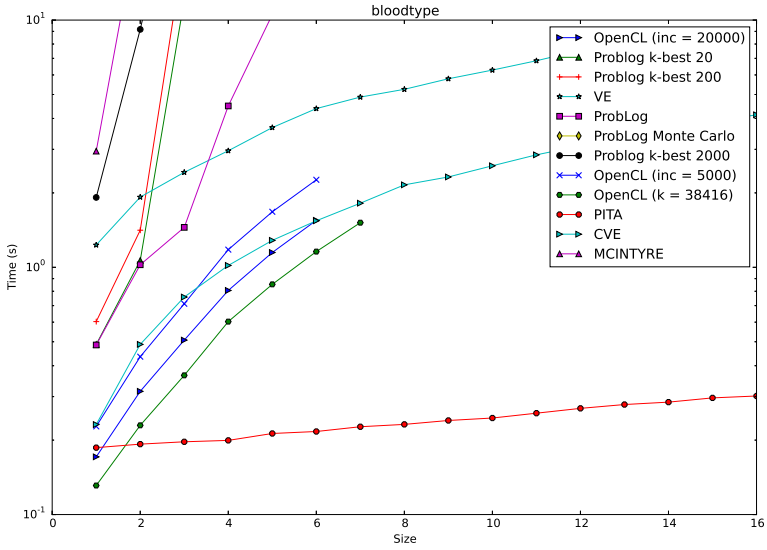


Figure 3.5: Results for the BLOODTYPE benchmark.

The results for the three benchmarks are shown in Figures 3.5, 3.6 and 3.7. Here, each data point is the lowest runtime observed in three runs of the systems. Three variants of our OpenCL algorithm are included in the comparison: *inc = 20000*, *inc = 5000* and $k = 38416$. The first two variants use the MCINTYRE stopping criterion, where *inc* is the chunk size (i.e., the number of samples that are run at once). A smaller *inc* allows the total number of samples computed to be closer to the actual minimal value for which the stopping criterion is satisfied, but may increase the likelihood of GPU processors having to wait for data transfers. Our experiments show that the large chunk size consistently performs better. Our third variant does not actually use the stopping criterion at all, but just uses its worst case value of $k = 38416$ as a fixed number of samples. For k-best, we have included experiments with $k = 2000$, $k = 200$ and $k = 20$. As the results below show, the precise value of k has only a limited impact on the way in which this algorithm compares to the others.

For BLOODTYPE, the sampling approaches perform quite poorly in general. While several exact inference methods (PITA, CVE) are able to handle problems up to size 16 in a second or less, MCINTYRE (not visible in the graph, because all measured runtimes exceed 10s), Problog Monte Carlo and the three version of k-best already require more than $100s$ for size 4. Our algorithm does better and

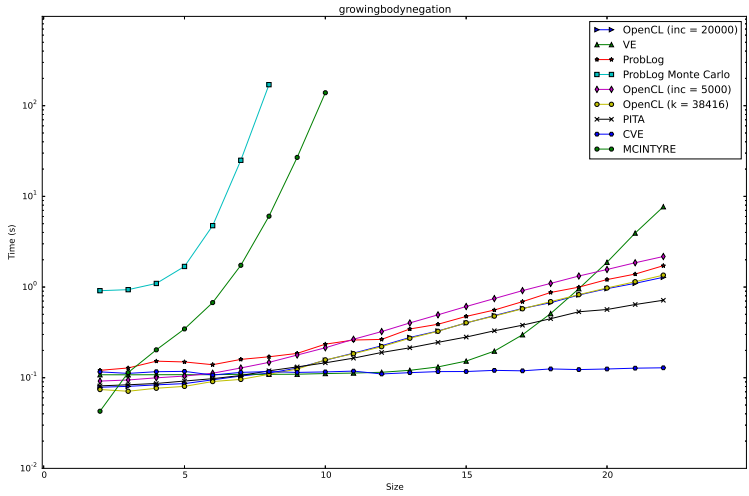


Figure 3.6: Results for the GrowingBody benchmark.

can handle up to size 6 in about a second. After that, however, the theory no longer fits in the global memory of our GPU. We have also run some experiments with a Tesla C 2075 GPU, which has 6 GiB of global memory. This allows instances up to size 11 to fit in GPU memory. The runtimes for these problem sizes were still in the order of a few (< 5) seconds, for all three variants of our algoritm, following the Contextual Variable Elimination (CVE) curve. As this benchmark is the only non-ground one, the times reported for our algorithm include the time needed to ground the theory ($\sim 0.01s$–$0.1s$).

A similar phenomenon can be seen for the GrowingBody benchmark. Again, CVE and PITA perform best, comfortably handling problem sizes ≥ 20, while the sampling methods of MCINTYRE and Problog MC already struggle to reach size 10. Here, however, our sampling approach does significantly better. While its runtimes do grow faster than that of CVE and PITA, it is still able to handle sizes ≥ 20 in under a second.

Finally, the GrowingHead benchmark is clearly better suited for the sampling methods. Here, Problog MC is the worst of the sampling methods, being (slightly) outperformed by the exact inference of PITA, but with the gap narrowing for the larger problem sizes. For $k = 20$, the k-best algorithm runs slightly faster than PITA, but produces an estimate that is almost 0.2 below the actual probability. MCINTYRE performs best in this benchmark, handling

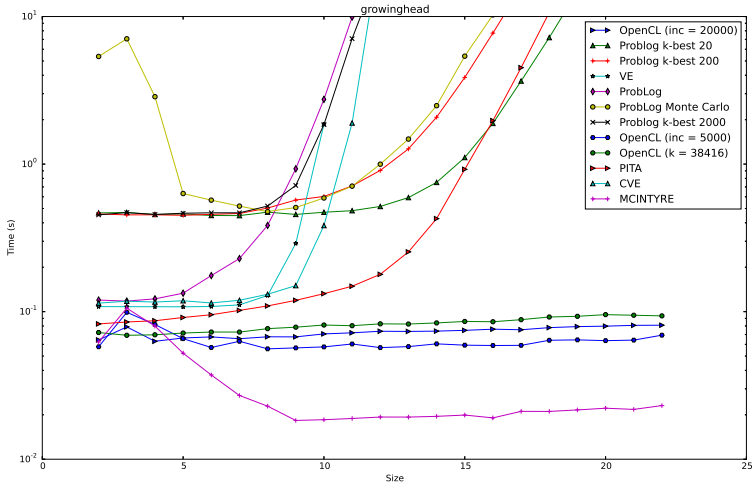


Figure 3.7: Results for the GROWINGHEAD benchmark.

problem sizes ≥ 20 in a fraction of a second. Our algorithm is somewhat slower, probably due to the overhead of copying data to the GPU, but, like MCINTYRE, it also shows an almost constant runtime (around $0.1s$) for this benchmark.

3.8 Discussion and future work

Sampling methods have a long history as ways of approximating complex computations. Typically, the development of a new sampling method requires a significant amount of domain knowledge and creativity in order to be effective. PLL may help to improve on this situation, by allowing the development of generic sampling methods, that can be applied to a wide range of probabilistic models. For instance, [70, 71] developed a general framework for constructing sampling algorithms, which they apply both to estimate the number of models of a DNF formula and to estimate the failure probability of a network with possibly faulty edges. In CP-logic, representing such a network is almost trivial (see e.g. [30]); moreover, the Problog system is able to reduce the problem of computing its failure probability to that of computing the probability of a DNF—and to solve this latter problem, precisely the sampling method of [70] may

be used [131]. In this way, the Problog PLL system has in effect incorporated Karp et al.'s methodology.

The fact that the efficacy of sampling methods typically depends on having a large number of independently drawn samples of course makes these methods in general a suitable target for GPU hardware. This has been investigated for instance in the context of physics [96], biochemical networks [171] or finances [7]. A general study of a number of different sampling methods on GPU from a statistical point-of-view was performed in [80]. Their results demonstrate that remarkable speed-ups (from 30× to 500×) can indeed be achieved. Here again, we may hope that eventually PLL methods will be able to offer a uniform framework which is able to reproduce the results from these special-purpose methods.

While the suitability of GPU hardware for sampling algorithms has thus already been demonstrated, we are not aware of any attempts to exploit this possibility in the context of PLL. Our efforts, using the OpenCL framework, are therefore, to the best of our knowledge, the first in this respect. When we compare our work to the existing studies of sampling on GPU, we find that focus of these methods is somewhat orthogonal to the results we have presented in this paper. This is due to the fact that logic programming concepts play a significant role in our method, whereas existing approaches are more statistical in nature. As a consequence, the bulk of our algorithm is concerned with the computation of a single sample, which is typically considered to be an atomic operation in more statistically oriented work. On the other hand, our method directly samples from the target distribution in a very straightforward way, whereas statistical methods often sample from a different distribution in order to reduce the number of samples that is needed. In future work, it could be examined whether these methods can also be integrated with our work. In particular, this could prove very useful when the goal is not to compute an *a priori* probability $P_T(\phi)$ but an *a posteriori* $P_T(\phi \mid \psi)$ where ψ itself has a low probability. However, this is beyond the scope of the current paper.

When comparing our sampling method on GPU with other inference methods for the PLL language of CP-logic, we have first of all shown that a careful optimisation of the code is important in order to achieve decent performance. However, with these optimisations in place, our implementation was generally able to outperform the existing PLL sampling methods. A single exception was the GrowingHead benchmark, where our algorithm shows similar performance to MCINTYRE, but was a constant few hundredths of a second slower, probably due to the overhead of using the GPU.

When comparing the sampling methods in general to the exact inference algorithms, we saw that the PITA algorithm and the CVE algorithm both

had clearly better performance on one of the benchmarks. However, for the other benchmarks, they performed significantly worse.

We can conclude from these results that:

- Certain benchmarks are more suited towards exact methods while others favour sampling methods, which demonstrates again that both kinds of algorithms have their place;
- Our method is an improvement on the state-of-the-art of sampling algorithms for CP-logic, delivering a more robust performance across different benchmarks.

A particular challenge in our experiments is that we have had to compare the performance of our method, running mainly on GPU, with existing algorithms that only use the CPU. Hence, a bias could have been introduced into our results by using a comparatively more or less powerful GPU. To avoid this, we have chosen to use similarly priced processors: at the time of writing, both the CPU and GPU that were used are worth around $100-$200. Moreover, they are commonly found together in typical desktop machines.

One limitation of such low-end GPUs is that they typically have a relatively small amount of memory available, causing our approach to become useless when the CP-logic theory grows too large. However, this problem does not appear to be too severe in practice: we only encountered it for one of our three benchmarks and, even there, we were able to reach a problem size where several other approaches (standard Problog inference, the k-best method and MCINTYRE) had already either timed-out or run out of (main) memory as well. The problem can of course be mitigated further by switching to a higher end GPU: by using one in the price range of $\pm$\$1500 instead of $\pm$\$150, we could reach problem size $n = 11$ instead of $n = 7$ for this benchmark.

Our method not only differs from existing sampling methods by using GPU hardware, but also by not being query-oriented: each sample constructs the entire interpretation at one of the leaves of an execution model. In our experiments, we have used this interpretation only to check that a single query is satisfied. However, with essentially the same effort, we can compute the probability of a set of queries at once. By contrast, all of the other algorithms that we have considered would need to be run for each query separately. In this sense, the experiments that we have conducted therefore represent the worst case for our algorithm, namely, that in which only a single query is of interest. The ability to compute the probability of multiple queries at once is useful in, e.g., applications where the goal is to find the most probable of a set of atoms, or to compute their entire joint probability distribution.

The fact that our method is not query-based also opens up interesting possibilities for future work. For instance, our method could be applied in cases where query-based methods have problems with an infinite number of explanations. Suppose a player repeatedly rolls a die until he gets the number six. Computing the expected number of rolls is not a trivial task for algorithms that depend on a completely constructed proof tree, as they may end up in an infinite recursion. Our approach does not suffer from this problem, even though our current ground-and-solve method obviously cannot cope with the fact that this example would generate an infinite grounding. However, there appears to be no conceptual difficulty in switching to a lazy grounding method, where we would load the non-ground theory directly onto the GPU and then generate instantiations of the rules only when they can actually be fired.

A second interesting topic for future work is that our algorithm may also be easier to extend to new language features, because it more closely follows the execution model semantics of CP-logic. This may in particular be the case for language features that extend the expressive power of the heads of CP-logic rules. For instance, while the head of a CP-logic rule is currently a choice between a fixed number of alternatives, we could allow the set of possible alternatives and/or their probabilities to be dynamically determined, based on the interpretation $I(s)$ in the state s where it is executed. A similar feature can be found in the P-log language [14], which has some similarities to CP-logic, but does not follow Sato's distribution semantics. (A comparison between CP-logic and P-log can be found in [154].) A second example is that, instead of allowing only a discrete choice in the head of rules, we could also allow a value to be selected from a continuous distribution. Finally, recent work [153] has investigated the possibility of extending CP-logic with negated atoms in the head. Integrating such features into our forward sampling algorithm seems much easier than extending one of the proof-based methods.

3.9 Conclusions

We have presented a sampling method for the expressive Probabilistic Logic Programming (PLP) language of CP-logic. It differs from the existing state-of-the-art in two ways: it makes use of the OpenCL framework to take advantage of massively parallel GPU hardware; and it is a forward method, i.e., it is not query-based. As is common for OpenCL programs, the efficiency of our implementation depends to a large extent on which optimisations have been applied: in one case, our most optimised variant was 37 times faster than the original.

With the necessary optimisations in place, our algorithm proved to be an improvement on existing sampling methods (MCINTYRE and Problog Monte Carlo), providing a more robust performance across different benchmarks. In addition to its greater efficiency, our algoritm can also compute the probabilities of a set of queries almost as quickly as that of a single query. Our results also show that neither the exact inference algorithms nor the sampling methods are able to consistently outperform the other, suggesting that both have their place among PLP inference methods.

One limitation of our current work is that we have focused on ground CP-theories, which means that grounding is a potential bottleneck. In future work, a lifted method could be developed, which would allow rules to be stored on the GPU in a first-order format. During the sampling process, these could then be instantiated on the fly. This could also help to avoid the problems we observed with memory limitations on the GPU. Moreover, it would also allow us to handle problems with an infinite grounding.

In the example in the introduction we had runtime-speed as a hard constraint. For an automatic cinematographer that needs to work on live recordings, fast inference is essential. The runtime-speed hard constraint leaves a trade-off between hardware cost and accuracy. Figure 3.4 shows the trade-off between number of samples and how close the resulting probability lies to the true probability. Indeed, increasing the number of samples also increases the accuracy. Using a GPU with more computing power will be able to process more samples in the same amount of time, giving us a trade-off to *explore* between the amount of hardware and the accuracy of the output. The Monte Carlo-like method we use in this work is inherently easy to parallelize. Every pass through the CP-Logic theory is independent of other passes making inter thread communication unnecessary. Only at the end of the theory we have to accumulate the results into a global mean. By *optimizing* the algorithm even more we were able to get a better trade-off between needed hardware and accuracy. We essentially get accuracy for free.

Algorithms with no inter thread communication such as this one, are easy to parallelize. Up to a point (determined by Amdahl's law), adding more hardware comes at an increase in speed. As is evident from the results of our different parallel optimization, making the algorithm more parallel clearly results in better performance.

Chapter 4

Ultra-low-latency automatic endoscopic image orientation stabilisation

As a second case, in this chapter we study a video counter rotation algorithm for use during endoscopic surgery. A wildly different application compared to the previous ones that still employs some of the same constraints.

This work was previously published in the IADIS International Journal on Computers Science and Information Systems [147], in the proceedings of Computer Vision and Image Processing 2016 - Part of the Multi Conference on Computer Science and Information Systems 2016 [148] and demonstrated on the Benelux Conference on Artificial Intelligence [151] where it won best demo award.

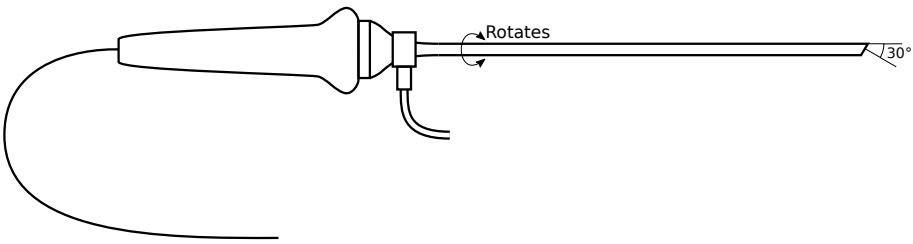


Figure 4.1: The endoscope lens rotates freely on the camera attachment.

4.1 Introduction

Minimally invasive surgery is a type of surgery where small incisions are made in order to operate at a remote location in the patient's body. These incisions make room for the required instruments, as well as an endoscope which acts as the surgeon's eyes during the procedure. The endoscope is commonly equipped with an angled lens (typically set at 30 degrees, see figure 1), in order to enable the surgeon to get a broader view. Moreover, this also makes it possible to look behind objects by turning the endoscope, thus revealing perspectives which would be concealed with a normal endoscope. While this extra information provides a significant benefit to the surgeon, the angled lens is also a source of confusion and disorientation, since the act of rotating the endoscope also rotates the on-screen image [22]. For certain kinds of procedures, this confusion can be avoided by placing a small amount of water into, e.g., the abdominal cavity in order to provide the surgeon with a reference "horizon". In situations where this is not possible, the surgeon may choose to use an endoscope that allows its lens to be rotated independently of the endoscope itself (see figure 1). A downside of this solution, however, is that the movement required to turn only the lens while keeping the endoscope steady is quite cumbersome and often requires both hands of the operator, while also restricting camera movement.

A more promising approach is to solve this problem by postprocessing the video footage coming from the endoscope. Such postprocessing may consist of automatically counter-rotating the output of the endoscope, or of drawing a virtual "horizon" on screen, which informs the surgeon of the current rotation of the image. In both cases, the core problem is that of determining the current rotation angle. Generally speaking, there are two ways of doing this: either by using some kind of hardware gravity sensor [61], or by using the images captured by the endoscope. While the use of a gravity sensor has the advantage of being drift free, an obvious disadvantage is that it requires a special endoscope equipped with such a sensor. In addition, gravity sensor endoscopes do not work if they are held above the patient, looking down.

By contrast, the image-based approach has the advantage that counter-rotation can be implemented entirely in software. Indeed, by tracking the translation between consecutive frames of the video feed, the orientation of the video can be automatically determined. Such an approach has already been investigated [99], but—to the best of our knowledge—it has never been implemented in a system used for live surgery. One of the main problems that need to be tackled in order to make this possible is that the video feed needs to be rendered at minimum latency. Indeed, for the hand-eye coordination of the surgeon, latency is a crucial parameter [113]. A recent study found that a latency above 130ms has detrimental effects on surgeon's performance [76].

The company eSATURNUS has developed a state-of-the-art digital operating room environment, called NUCLeUS TM , which operates at extremely low latency: it streams 1080p60 video with a latency of $<$ 16ms, i.e., less than one frame. In this paper, we investigate how a video-based rotation tracking algorithm can be successfully integrated in this system. To do this, we have to tackle the unique challenge of combining high-framerate, high-resolution images with the requirement of ultra-low latency. We investigate the implementation of the algorithm on embedded hardware as a method for coping with these challenges.

Before we go deeper into the inner working of the implementation of our counter rotation algorithm, like we did in the last chapter, we will first apply our triangle framework that was introduced in chapter 1 to it. Clearly this application is constrained in time, resulting in a trade-off between accuracy and hardware economy (figure 4.3). We will *explore* this trade-off by implementing the system on three different distributed hardware configurations with a wildly varying amount of performance. In a sense, speed and tracking accuracy are also intertwined. Tracking accuracy is mostly affected by the interval at which the tracker manages to update. When we do not get enough tracking updates the Kalman filter fails to predict the correct position and the angle starts to drift.

This research has resulted in the development of a prototype, that can be seen in action at: http://www.youtube.com/watch?v=2tMk0u0yWc0. Figure 4.2 shows a screenshot of this video.

The remainder of this chapter is organized as follows. In section 4.2, we lay out the algorithm we used for our implementation, while section 4.3 explains how the algorithm is implemented on resource constrained hardware. Finally, section 4.4 discusses results in terms of performance and accuracy

4.2 Rotation compensation

The requirement for ultra-low latency forces us to consider a relatively simple approach to computing the rotation of the video. For this reason, we avoid feature matching techniques, such as Scale-Invariant Feature Transform (SIFT) [88] or Speeded Up Robust Features (SURF) [16], and have chosen to use a tracking algorithm instead.

The general idea of such a tracking approach is illustrated in figure 4.4. The left-hand left side of this figure shows two consecutive frames of the incoming video. A number of distinctive points in the image (also called “corners”) are extracted and these are then tracked across subsequent frames. From the

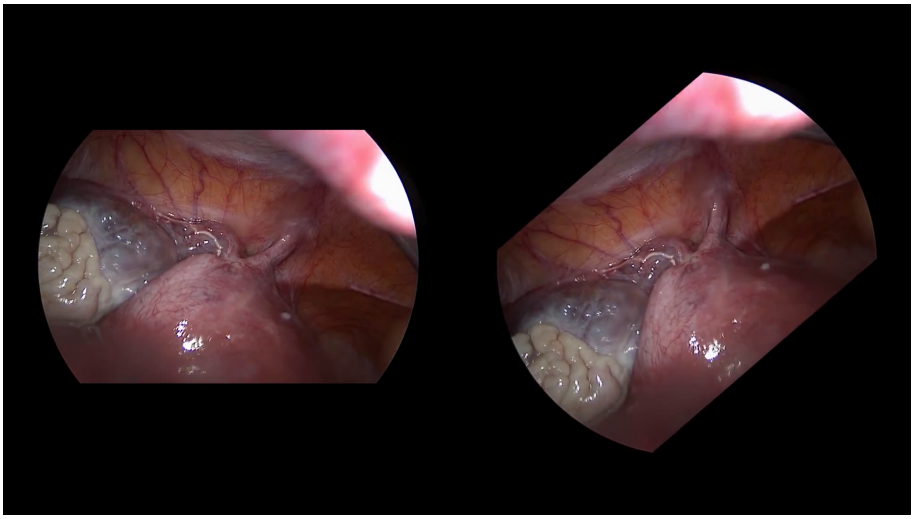


Figure 4.2: The prototype in action.

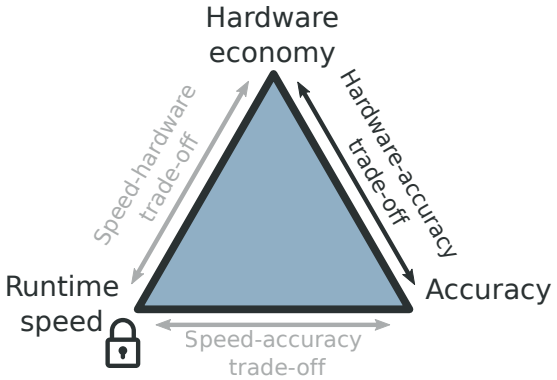


Figure 4.3: Low-latency video streaming is crucial in this application, leaving a trade-off between computing power and accuaracy..

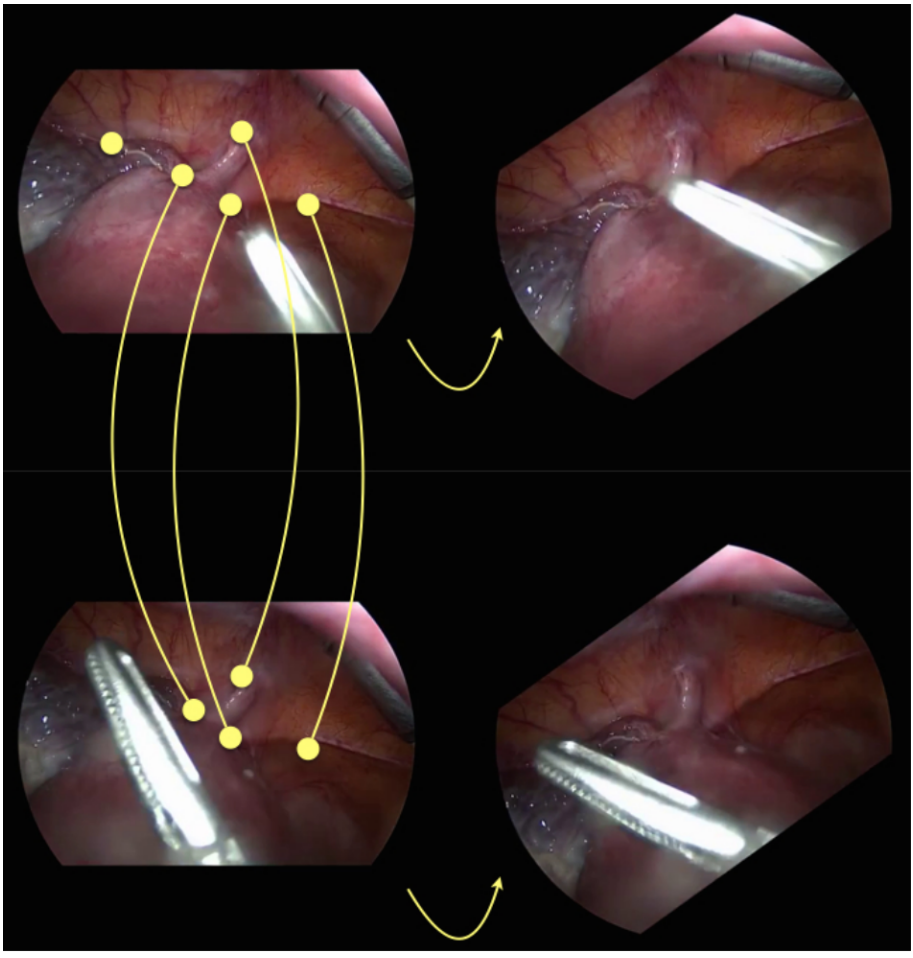


Figure 4.4: The tracking approach. The dots represent tracked corners in consecutive frames.

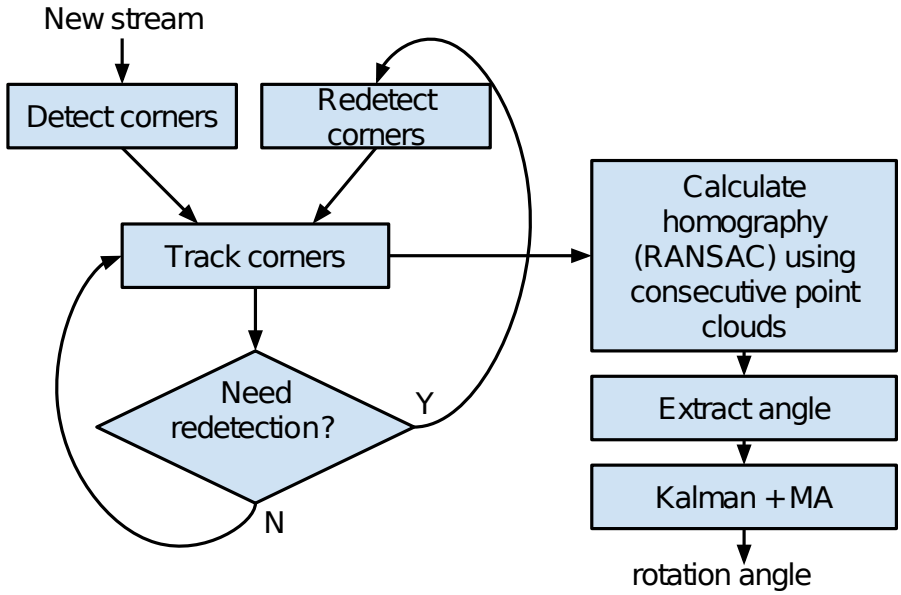


Figure 4.5: Overview of the algorithm.

movement of the corners from one frame to the next, the rotation angle between the two frames can be calculated. This angle can then be used to rotate the frame back to the reference orientation. The resulting "straight"frames are shown on the right-hand hand side of this figure.

To implement this approach, we have chosen to use a Kanade-Lucas-Tomasi feature tracker (KLT) tracker [89,139]. The general structure of this algorithm is shown in figure 4.5. We will first clarify how these different parts work together, and why they are needed, before we explain each part in detail.

Processing starts when the operator enables the rotation rotation compensation. This is done by pressing a button on the endoscope, while holding it level. At that point, the system takes the most recent frame of the stream and applies the corner detection algorithm. Once the corners have been extracted, the system can start tracking them through the next frames. However, over the course of the video, the system may lose track of some of the original corners. Once the number of tracked corners drops below a certain threshold ("need redetection" step in figure 4.5), the corner detection algorithm is again applied and the process continues. Meanwhile the corners are used to calculate the homography

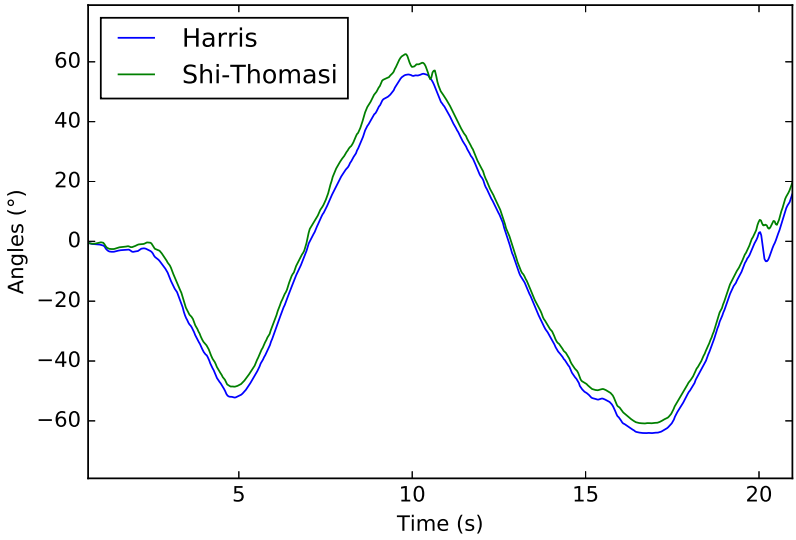


Figure 4.6: Comparison of the Harris en Shi-Tomasi corner detecton. The predicted angle is plotted relative to the elapsed time in the test video (footage from real endoscopic surgery). The output of both corner detectors is almost identical.

which is used to calculate the rotation angle. This angle then passes through a Kalman and moving average filter which gives us the final rotation angle. We now discuss each of these components in more detail.

4.2.1 Detecting Corners

To detect corners in the endoscopy images, we use Harris corners [54], which are known to produce good results for video footage in which there are small changes between consecutive frames [127]. We have also tried Shi and Tomasi's "Good features to track" [130], and found that they produce almost identical results on typical endoscopy footage (see figure 4.6). As suggested in the OpenCV manual, we track only corners that have a "quality level" of at least 0.01. On average, this results in about 170 corners being tracked.

4.2.2 Tracking

To track corners from one frame to the next, we use a pyramidal implementation of the KLT tracker as described in [21]. The implementation makes use of a scale space pyramid built on top of the original image. Each new layer is constructed by downscaling the previous layer by some constant factor. Feature tracking then commences at the top—i.e., lowest resolution—layer. These rough results are then gradually refined by descending in the pyramid.

4.2.3 Calculate Transformation

In this step, two point clouds—one from the previous frame and one from the current frame— are compared to each other in order to calculate the transformation between the two frames. We calculate the homography between both clouds using OpenCV's RANSAC [44] implementation which yields a transformation matrix of the form:

$$T = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{4.1}$$

For laparoscopy video, we can assume that this transformation is Euclidean (i.e., $h_{31} = h_{32} = 0$ and $h_{33} = 1$). Therefore, we can extract the rotation angle as:

$$\alpha = \arctan2(h_{21}, h_{22}) \tag{4.2}$$

4.2.4 Interpolation of Estimated Angle

Because our algorithm is run on a resource constrained system (see Section 4.3), the tracker may not be able to compute a rotation angle for each frame in the video, especially when dealing with 1080p60 video. Therefore, it may be necessary to drop a number of frames between successive estimations of the angle. To make sure that the video nevertheless rotates smoothly on the screen, we use a Kalman filter, in combination with a simple moving average filter, to interpolate between two successive updates.

The details are as follows. We use a constant velocity motion model Kalman filter, assuming that the change in rotation angle is constant over time. Therefore, it uses the following linear model to predict the angle α_t at any given time point

t.

$$\begin{bmatrix} \alpha_{t+1} \\ \dot{\alpha}_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & \delta_t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_t \\ \dot{\alpha}_t \end{bmatrix} \tag{4.3}$$

Here, δ_t is the time elapsed between time points t and $t+1$ (because the angle computation might have to skip frames, δ_t may not be constant).

After making the new prediction α_{t+1}, the Kalman model is then updated using a weighted average of α_{t+1} and the new measurement, i.e., the angle that was computed by the KLT tracker (the observation vector contains only the new angle). The weights depend on a process covariance matrix Q and a measurement covariance matrix R. We have empirically determined the following values for these matrices:

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 77 \end{bmatrix} \tag{4.4}$$

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.5}$$

The high value for velocity in the process covariance matrix indicates that the predicted velocity is often off by a large margin, when the surgeon makes sudden movements. When frames have to be dropped, the Kalman filter may start to drift away from the real rotation angle. This can happen in particular when there are sudden camera movements that violate the Kalman filter's assumption of a constant $\dot{\alpha}_t$. In such a case, the Kalman filter may have to perform a large correction when a new angle finally does arrive. This can give a jerky effect in the video. Therefore, we pass the resulting angle to a moving average filter which further smooths out the jerkiness. This filter keeps track of the last n samples (in our case, $n = 10$) and is updated at a constant interval. Our implementation does this in a separate thread. The output of the moving average filter is used as the final result.

4.3 Implementation

The general architecture of the NUCLeUS Digital Operating Room is shown in figure 4.7. Video is captured by the source module, which transmits a direct, full HD feed to the receiver module, which decodes this feed and displays it on screen. There is also a secondary, lower-resolution SD feed, which can be used for other purposes, e.g., it can be streamed over the internet to medical staff to follow the procedure from a remote location.

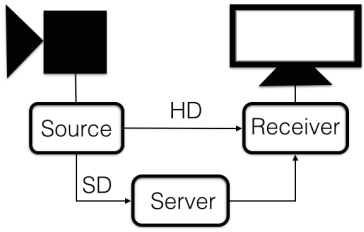


Figure 4.7: System architecture.

This architecture provides us with different options for implementing the tracking algorithm:

REC The computation of the angle happens entirely on the receiver module.

R&S The KLT tracker runs on the source module, while the Kalman and moving average filters run on the receiver.

SRV The angle computation happens on the separate compute server.

We describe each option in detail below.

4.3.1 Using only the Receiver Module (REC)

The receiver module contains the quad core iMX6 processor with Vivante GC2000 GPU that was described in chapter 2.2.1.

The REC approach uses this GPU to run the KLT tracker. When a new frame comes in, it is first downscaled to a resolution of 800x448px and then copied to GPU memory space. The GPU then calculates each level of the image pyramid of the frame, as well as its derivatives in both x and y directions, using the Scharr operator [126]. OpenCL's float4 type is used as much as possible in order to benefit from the GPU's vectorised instructions, which allow 4 floating point computations to be carried out in a single operation. A row of the convolution mask consists of 5 floating point values. The first four entries of every row are stored in a float4, while the remaining entry is stored in a single float. Similarly, we read the first four pixels (with respect to the mask) of the image as a single float4 and do an element-wise multiplication with the vector of the mask. In this way, 16 bytes can be read at the same time and all of the floating point capabilities of the GPU are exploited. In all cases, a single work item is launched for every pixel in the output image.

Once the image pyramid is computed, the actual tracking starts. We launch one work-item for each corner in the previous frame. On a typical desktop GPU, this would use only a fraction of the available compute units. However, since the Vivante GC2000 GPU only has four compute units, each capable of running four floating point operations at the same time, this parallelization strategy is able to keep the available hardware occupied. Inside the kernel, the displacement of each feature is calculated using the KL tracking algorithm. Once an updated position for the highest level of the pyramid is calculated, this position is refined on the next level of the pyramid, using the position of the level above as a starting point. This is repeated until we reach the lowest level of the pyramid.

Finally, we transfer the updated locations of the corners back to the CPU and use OpenCV to calculate the transformation between the current and previous point cloud. We then apply the Kalman and moving average filter.

4.3.2 Using both the Receiver and the Source Module (R&S)

This variant splits the computation over the receiver and source module: the source calculates the rotation angle and the receiver applies the Kalman and moving average filter. One advantage of using the source module to calculate the rotation angle is that we can make use of the already downscaled secondary video stream that is available on this module, while in the REC approach, the full HD stream still has to be downscaled on the receiver. Because the source module does not have a GPU, this implementation uses the OpenCV CPU version of the KLT algorithm instead of our own OpenCL EP implementation. The resulting angle is transmitted to the receiver module as part of the header of the image that is sent, so no additional communication delay is incurred.

4.3.3 Compute Server (SRV)

This approach performs the tracking on the separate desktop machine that gets the secondary SD feed. The resulting angle is sent to the receiver module, which uses it to rotate its direct HD feed and then renders the results on-screen. Because the angle and HD image arrive through different paths, they need not be synchronized: the receiver always uses the most recent angle it has received to rotate the most recent image. Because of this, the angle computation cannot increase the latency with which the video feed is rendered. If the angle computation is too slow, this will have as its only effect that the rotation will use an angle that is slightly off. The Kalman filter on the receiver module can compensate for this effect, while the additional moving average filter avoids

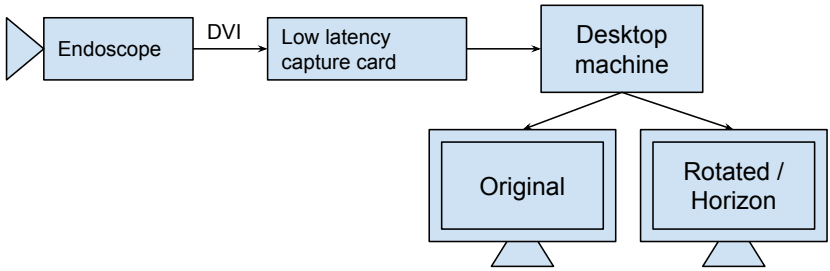


Figure 4.8: System architecture.

jerkiness upon receiving a delayed angle. If the compute server is powerful enough to keep up with the framerate, however, no interpolation of the angle is necessary. In this case, the Kalman filter just smooths out sudden movements, at the expense of causing the rotation angle to lag slightly behind.

4.4 Results

In this section, we first shortly describe a prototype we made, after that we evaluate our different implementations in terms of both performance and accuracy.

4.4.1 Prototype

To demonstrate the system described in this chapter we also made a prototype to show at various events. This prototype won the best demo award at the 2015 BNAIC conference in Hasselt, Belgium [152]. In the section, we will briefly illustrate this demo setup. Some of the experiments in section 4.4.2 are also done on this prototype. The algorithms used in the prototype are the same as those implemented in de Nucleus OR environment described earlier in this chapter. We do however use different hardware. We use a single desktop machine to do all the processing and render the result on two displays. An overview of the setup is shown in figure 4.10. A special purpose capture card is used to capture video from the endoscope (over DVI) at low-latency.

The first display shows the original video coming from the endoscope, and the second shows the tracked corners, the rotated video feet or the horizon projection (shown in figure 4.9). As a test subject for demonstration purposes, we found that using a bell pepper demonstrates our algorithm very well.

4.4.2 Performance

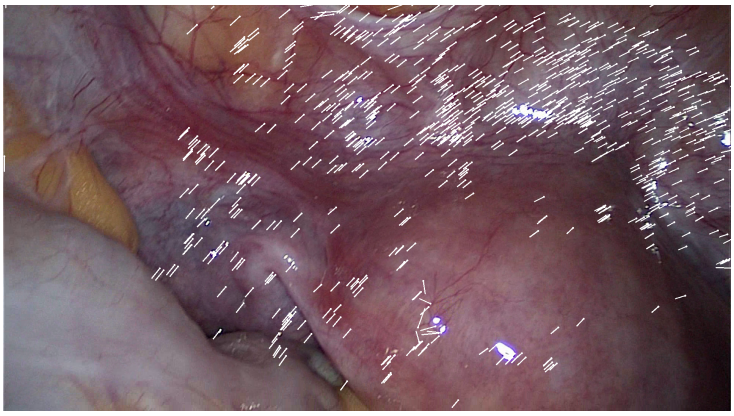
If the computation of the rotation angle cannot be performed fast enough to keep up with the framerate of the video, our approach is to drop frames from the angle computation. In this way, we avoid increasing the latency with which the video is rendered, but this obviously comes at the cost of lowering the accuracy of the computed angle. Figure 4.10 examines the effect of increasing the interval between two angle computations on the accuracy of the angle. The baseline corresponds to computing an angle for each frame of the 30fps video. We can therefore achieve the baseline by computing each angle within 33ms. As the time needed for each frame increases, we have to drop more frames and the accuracy deteriorates. As figure 9 shows, we can go as low as 5fps and still have a reasonable approximation of the baseline.

Figure 4.11 shows the performance of our algorithm on the different system configurations discussed in section 4.3, measured over 40 trials.

The REC approach is infeasible, as it takes an average of almost 28s per frame. The Vivante GPU that is available on this module is therefore clearly not powerful enough to perform the required computation.

Shifting the computation of the rotation angle to the source module (R&S) provides a significant improvement. This is due to the fact that the source module's CPU (an Intel atom Z530) is more powerful than the processors that are available on the receiver, as well as the fact that we avoid having to downscale the video twice. However, the average computation time of 0.43s means that we could still only compute 2 angles per second. As we can see from figure 9, this is an unacceptably poor approximation of the real rotation angle. Moreover, the worst execution time for this approach was substantially slower (0.89s) than the average. Therefore, this approach has to be ruled out.

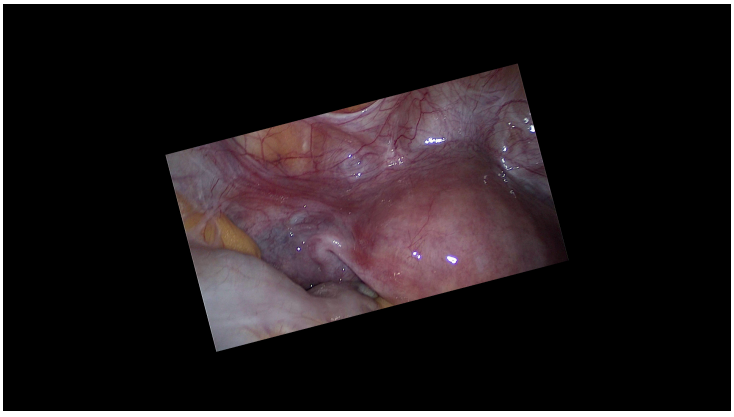
We can conclude that the approach with a separate compute server (Intel Xeon E5-2630 in our tests) is most feasible in this case. While the downside of this approach is obvious—i.e., the need for additional hardware—the gain in runtime outweighs the cost for this latency-sensitive application. Indeed, with an average runtime of 11ms and a worst runtime of 28ms, this solution is able to keep track with the 30fps framerate of the video. Taking also into account the network transfer of the low-resolution image feed to this compute server



(a) Visualization of tracked corners.



(b) Horizon projection.



(c) Rotated image.

Figure 4.9: Different display modes in the prototype.

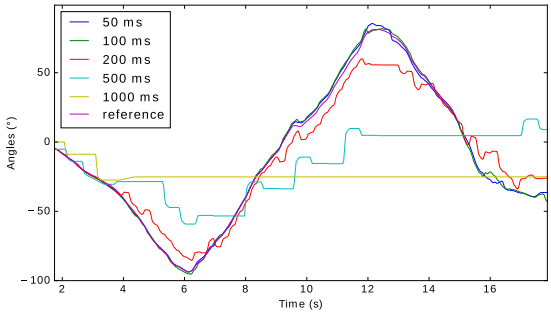


Figure 4.10: Angles at different update rates.

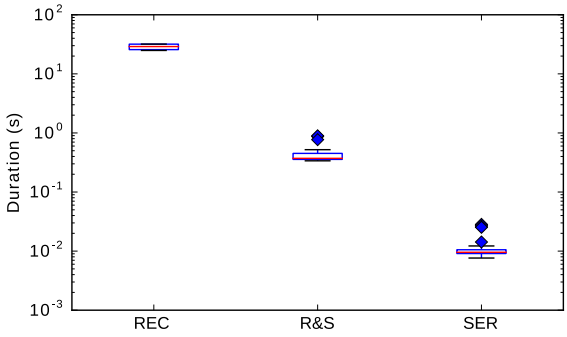


Figure 4.11: Runtime comparison.

and the communication of the angle from the server to the receiver, the latency with which the computed angle arrives at the receiver module is approximately between 50 and 100ms. This slight lag in rotation angle is not noticeable in practice.

As discussed before, the video itself is rendered from the direct HD feed, so angle computation does not affect its latency at all. However, the GPU that is available on the receiver module of NUCLeUS cannot actually counterrotate the HD video in 60fps. One solution is to work with 1280x720@30fps video instead. The other is to simply render the incoming HD feed as is, but to impose a "horizon" on it that informs the surgeon of the current rotation. We have implemented both approaches.

4.5 Conclusion

To cope with the disorientation arising from the use of angled endoscopes, we have developed an orientation stabilisation method, based on the KLT tracker (Lucas and Kanade, 1981; Tomasi and Kanade, 1991), and examined how to integrate this into the NUCLeUS Digital Operating Room. This state-of-the-art system aims at providing an ultra-low-latency solution, which is important since research suggests that latency is a key parameter for surgeon's performance. The aim of this paper is therefore to investigate methods of integrating orientation stabilisation into this system without adversely affecting this latency.

We have developed two embedded implementations of the KLT tracker: an OpenCL EP implementation running on the GPU in the receiver module and an implementation running mainly on the CPU in the source module. Our experimental results demonstrate that neither of these implementations is fast enough to allow a reasonable approximation of the rotation angle. However, the IP-based architecture of NUCLeUS also allows a third option, namely that of running the bulk of the computation on a separate desktop machine. This option allows the latency with which the video is rendered to remain unchanged, while the rotation angle that is computed never lags more than a few frames behind.

As mentioned in the introduction this application fits in our framework as being constrained in time, leaving a trade-off between hardware and accuracy. Indeed, while the amount of computing power directly influences the speed at which the algorithm runs, a lack of speed results in a less accurate tracking result. When the angle estimator does not get enough "real" samples it becomes harder and harder to make an accurate prediction of the true angle. This is true in general in real-time applications. Slower algorithms generally give more accurate results on a single point in time. However, if the result is used in a time dependent application such as a tracker, it is often better to use a less accurate algorithm that is able to give more updates. Due to the design of our system the impact on video latency (the constraint that was hard) was always kept to a minimum.

In this work we tried three different hardware configurations, which each resulted in different rotation accuracy. We reached the conclusion that, in a crucial application such as this, the configurations without a compute server suffered too much in accuracy (loss in tracking) that it was better to still use a compute server with sufficient resources. Indeed, when it is not possible to get enough updates rotation accuracy also suffers.

Chapter 5

GPU implementation of ACF Algorithm

This chapter is about our GPU implementation of the ACF pedestrian detector. At the time when this research was carried out, ACF was a very popular pedestrian detector known primarily for its speed at a decent accuracy. ACF is a fast variant fitting into the channel features based detectors family which had a good potential for high accuracy by extending the amount of features. Other detectors in this class including among others: Integral Channel Features (ICF) [35] which formed the base of channel based detectors, squared channel features [167], filtered channel features [169] and rotated channel features [168]. While it is already very fast, there are some applications (think large-scale person detection or embedded hardware) that could benefit from an even faster detector. While there were more accurate algorithms at the time, ACF still had a lot of potential as a first "filter" before detections are passed to a more accurate deep learning detector (or classifier).

When this research was carried out, fast deep learning detectors like YOLO [115] and SSD [86], where just becoming practical. We therefore compare our GPU implementation of ACF to these. With this research we tried to answer the question if handcrafted detectors where still worth it in an environment where GPUs are available everywhere.

Portions of this work were published in the proceedings of the VISAPP 2018 Conference [146].

5.1 Introduction

Traditional handcrafted methods for pedestrian detection (like Histogram of Oriented Gradients (HOG) [28], Aggregate Channel Features (ACF) [32], Deformable Parts Model (DPM) [39]) which where the state-of-the-art just a few years ago are nowadays in many cases surpassed by the rise of Deep Learning in terms of accuracy. However, on embedded platforms traditional methods are still quite relevant. Applications such as pedestrian safety around self driving cars [141], Unmanned Areal Vehicles [137], some surveillance applications... often demand real-time performance with only a limited amount of computational resources. This meant that in the past, deep learning was not suitable for use on embedded platforms, traditional detectors like ACF where the most suitable solution. However, with the arrival of platforms like the Jetson TX2, which offer a really powerful GPU in an embedded low-power package. Deep learning on embedded platforms has become more feasible.

The goal of this paper is to make a fair comparison of the old hand crafted methods to newer deep learning methods on a platform like the Jetson TX2. For this, we need a good GPU implementation of a cutting edge pedestrian detector that uses hand crafted methods. In this chapter we report on our efforts on this GPU implementation, we go deeper into how the ACF algorithm can be parallelized so it can be used on a GPU.

To evaluate our implementation we compare it to the state-of-the-art Deep Learning object detector YOLO [115] in both accuracy and speed.

While this GPU implementation of ACF is not about a specific application, we do use it in many applications. Viewed in the scope of our constraints framework of chapter 1, depending on the application, different constraints are important. All of these applications can however benefit from a fast an accurate detector. Independently of the application or hardware, we can generally make a trade-off between speed and accuracy by varying different parameters in the algorithm:

Stride ACF uses a sliding window that evaluates the model at each possible position in the image. The stride between these windows is typically determined by the shrinking factor. If we want a faster detector we can increase this stride so that fewer windows have to be evaluated. This increases speed at the cost of accuracy.

Fewer weak classifiers ACF uses a variation of the AdaBoost algorithm [47]. A series of weak classifiers are evaluated in a cascade to make one strong classifier. Using a cascade of classifiers can speed up the result significantly, for instance in DPM [41]. Once a possible window position is ruled out,

we can stop evaluating early. For windows that are not ruled out, all weak classifiers are evaluated. We can make the algorithm faster by reducing the total number of weak classifiers in a strong classifier. When there are fewer weak classifiers, fewer have to be evaluated, resulting in an increase in speed, at the cost of accuracy.

These are ways to *explore* the speed accuracy trade-off. When we know our hardware, we might however want to *optimize* the algorithm further so that is runs faster at no extra cost. This optimization is discussed further in this chapter.

The remainder of this chapter is organised as follows. Section 5.2 explores the existing literature on person detection. Section 5.3 describes how the ACF detector works in detail. Section 5.7 explains our GPU implementation. Section 5.8 talks about the results of our ACF detector, and section 5.9 compares it to other (deep learning) object detection approaches. Sections 5.5, 5.6 and 5.10 explains the different applications we did in more detail, and also applies our constrains framework to them. We conclude in section 5.11.

5.2 Related Work

Pedestrian detection is a well studied problem, for which a lot of different approaches have been proposed. Currently methods that reach state-of-the-art accuracy almost all make use of deep neural networks. Detectors such as Fast-RCNN and Faster-RCNN use a two-stage approach. In the first stage a number of regions are emitted from a Region Proposal Network, which are then classified to further determine to which class, if any, the object belongs. Although these detectors have gained a lot of speed improvement over the years, they are still not sufficiently fast for real-time detection, let alone for embedded implementations. In recent years, a large speed gain was made by tackling the object detection problem as a single-stage approach (SSD, YOLO, YOLOv2)... The YOLOv2 detector [115] uses a single shot network to at the same time predict object class as well as bounding boxes. The output image is divided into a set of anchor points, each containing a detection with different anchor boxes. The SSD [86] detector uses a similar approach using only one network for both detection and region proposal.

Up to a few years ago, hand-crafted feature-based methods were the state-of-the-art in object detection. Detectors like Viola and Jones [156], Histogram of Oriented Gradients (HOG) [28], ICF [35], ACF [32] and Deformable Parts Model (DPM) [39] are some examples of detectors that use these kind of features. Viola and Jones and ICF calculate an integral intensity image, and use some kind of

Haar wavelets to generate possible feature values. HOG, ACF, ICF and DPM make use of so called HOG like features. Multiple histograms, each representing a small part of the image are calculated on the image gradient. Each bin in the image then represents a separate feature layer. The DPM detector learns a detector for different parts of the object which makes it more invariant to pose changes. In all of these detectors, the calculated features are then used to train a classifier using Support Vector Machine (SVM) or AdaBoost. To cover the entire image a sliding window approach is used to evaluate all possible detection windows in the image on different scales.

In this chapter we choose to focus further on the the ACF person detector for a few reasons: ACF is in itself, on CPU already quite fast, which means that it is often used as a person detector on embedded platforms. Porting ACF to GPU is something that to the best of our knowledge has not been done before. The authors in [104] explain how the Viola and Jones face detections algorithm can be ported to GPU, which is some ways similar to ACF.

The GPU implementation is an extension of our own CPU implementation of ACF, which is already faster than Dollár's Matlab implementation.

5.3 ACF Person detector

To be able to follow along with our GPU implementation of the ACF algorithm we will first give a brief overview of the ACF algorithm itself.

The ACF person detector uses an AdaBoost classifier which uses "ACF features" to classify image patches. The entire image is searched using a sliding window approach.

In total the ACF features consist of ten channels: Luminosity + UV-color coordinates (LUV) colour / intensity information, gradient magnitude and histograms of Oriented Gradients (HOG). They are calculated as follows: Red Green Blue (RGB) colour information coming from an image source is converted to the LUV colour space, a gaussian blur is applied and the resulting Luminosity (L) and chroma values (U and V) are used for the first three channels. The gradients (in all directions) of the image are calculated from the luminosity channel. The magnitude of the gradient, after again applying a gaussian blur is the fourth channel. The six remaining channels each represent a different bin (containing a set of orientations) in the gradient orientation histogram. A separate histogram is calculated for each patch of $n \times n$ pixels (often 4x4) in the gradient images, this means that the resulting feature channels will be downscaled by a factor of n (know as the shrinking factor). To make sure

that all channels have the same dimensions, the LUV and gradient magnitude channels are also downscaled by the shrinking factor. Each gradient magnitude in the $n \times n$ patch for which a histogram is calculated is placed in the two closest bins using linear interpolation according to its orientation.

Using these features a classifier can be trained to detect objects like people. In our implementation we are only interested in speeding up the evaluation phase as it is the only part that needs to run in real-time, and also the only part that will run on embedded hardware. For this reason we will only explain how evaluation of an ACF model is performed, and omit the training phase details.

For classification ACF uses a variation of the AdaBoost [46] algorithm. A series of weak classifiers (decision trees) are evaluated in a cascade (one after another) to make one strong classifier. Every decision tree adds or subtracts a certain value (determined during training) to a global sum which represents the detection score for a certain window, as seen in equation 5.1.

$$H_N(X) = \sum_{n=1}^{N} h_n(x) \tag{5.1}$$

Decision trees are evaluated sequentially. If at any point N the global score H_N reaches a value below a certain cut-off threshold, the evaluation for that particular window is stopped. Only for windows that never go below this threshold all decision trees are evaluated. Stopping early with the evaluation means that much fewer decision trees have to be evaluated making the evaluation much faster. Only windows with a high score (where the object likely is present) are evaluated fully. After evaluating each window in this fashion using the sliding window approach, Non-Maximum-Suppression (NMS) is applied which gives us our final detection boxes.

5.4 Applications without the use of GPU computation

Before we dive in to how to do all this on the GPU, in the next two sections, we first describe two applications that use ACF without GPU computation. Both of these applications have a hardware speed trade-off (shown in figure 5.1).

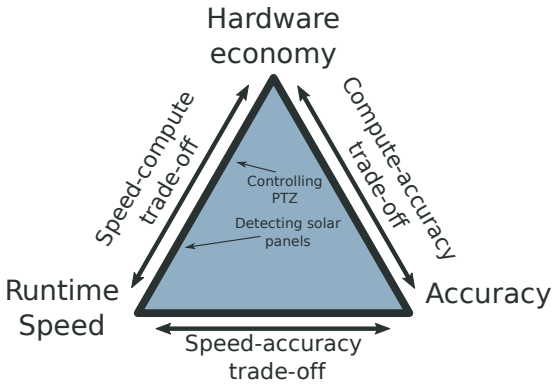


Figure 5.1: Visualization of different the applications, and where they fit on the triangle.

5.5 Application 1: Detection of photovoltaic installations in RGB aerial imaging: a comparative study

This work was previously published in the proceedings of the 2016 GEOBIA conference [112] and was preformed together with my colleague Steven Puttemans.

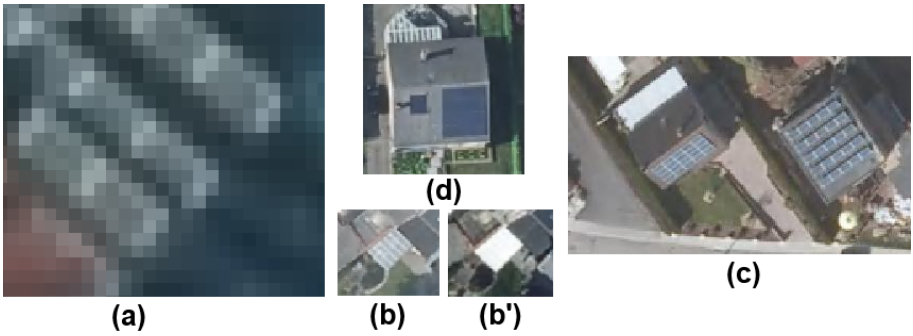


Figure 5.2: Challenges when dealing with solar panel detection.

5.5.1 Introduction

Solar panels provide a solution in generating energy from non-polluting resources and therefore placing a solar panel installation is subsidized and encouraged by governments and electricity grid administrators. However, due to the given funding and tax reduction, these 'greener' energy generating alternatives also give raise to malicious fraud. Our client, an electricity grid administrator, wants to use the power of computer vision to track down these fraud cases to ensure that no financial benefits are given to people that are not correctly registered at the grid administrator with their solar panel installation.

Fully automated solar panel detection in RGB images yields some major challenges, as seen in figure 5.2. First of all a solar panel is an object shape with only a few distinct visual features like shape and colour, due to its simple shape and colour distribution. Secondly the images that are freely available on the FGIA portal have a very limited resolution (25cm/pixel) resulting in a solar panel size of only 4×7 pixels, which is very little considering the real dimensions are around 1×1.5 m. This is clearly visible in figure 5.2 (a). Due to the material properties of solar panels, specular reflections tend to change the visual properties of the panels, depending on the position of the sun. This could lead to solar panels occurring as a flat white overexposed part of the image. This difference can be seen in figure 5.2 (b) and (b'). The orientation in which the solar panels are placed also varies (south, south-east, south-west), raising the need of a full 360° rotational search of the image for object instances. Combined with the different roofs' slants (30°, 45°, 60°, etc.), panels get optically deformed and thus appear shorter or longer in 2D images, despite the fixed physical size, as seen in figure 5.2 (c). Finally, solar panels come in different materials (mono- and polycrystalline, full black, etc.) introducing even more intra class variance into the problem. For example, due to the small resolution per panel, full black installations appear as a black square, yielding no visual features to train computer vision techniques, as seen in figure 5.2 (d).

Our goal is to develop a computer vision based approach that is able to automatically detect solar panels in aerial imagery and return the location of these installations with a high certainty. This in turn can avoid putting in huge amounts of manual labour to manually locate solar panel installations for fraud detection.

Problem type

In this work we compare different methods of detecting solar panels in areal imagery. Processing areal images can easily be done off-line, because there is

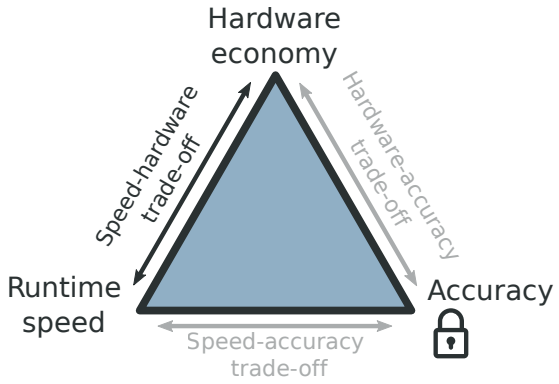


Figure 5.3: The off-line detection of solar panels is accuracy constrained, leaving a trade-off between runtime speed and computing power.

no need to process them in real-time, so runtime speed is not that important. Arguably, hardware economy is also not that important. Because the application is able to run off-line, we are only limited by hardware cost. This is an example of an application that can run on a powerful compute server. Accuracy, on the other hand is important. While there is no strict accuracy requirement we need to reach in these applications (this work is more about how feasible this approach is), the higher the accuracy, the more useful the system becomes. Runtime speed and hardware can change, but we would rather not give in on accuracy.

We therefore say that, in this application, accuracy is a hard constraint. We can *explore* a trade-off between speed and the amount of hardware needed. Although we can get a better trade-off by using different algorithms (which we explored in this application), the choice of algorithm is still largely dictated by the accuracy it reaches. Figure 5.3 depicts a system that is constrained in accuracy.

5.5.2 Related Work

The automated analysis of solar panel installations from aerial images using techniques of the computer vision fields, limits itself to the analysis of solar panel efficiency and defects, using RGB and thermal cameras, as seen in [81,123,140], while automated solar panel detection and localisation seem to be unexplored. While object detection is a well studied problem in the field of computer vision, many other fields still have not discovered the power of these state-of-the-art

techniques in autonomous object detection and localisation. The community of computer vision has however already performed object detection research in the field of aerial imagery, focussing on roads [60], buildings [93] or vehicles [49], always using state-of-the-art computer vision algorithms and thus giving perfect example cases that can be expanded to solar panel detection and localisation.

While the intuitive way would be to use segmentation based approaches, where the RGB input image is transformed to a colour space where separating object pixels from background pixels using strict (learned) thresholds is easier, they tend to fail when a wide range of other objects in the images have similar colour ranges. Furthermore, these techniques only take into account object colour information. Object categorisation techniques like [33, 34, 41, 155] tend to go one step further and use object properties like colour, shape, texture, etc. to uniquely define features that describe the object class, which are then used to successfully separate objects from the background.

5.5.3 Datasets

For developing and testing our suggested approaches, we used the freely available medium-scale (25cm/pixel resolution) aerial footage from FGIA (Flanders Geographical Information Agency) covering the grid area of Flanders where our electricity grid administrator is active. From this publicly available dataset, a set of 2500 individual solar panels were manually annotated and a set of more than 150000 random negative samples (containing everything except solar panels) were collected and used to train the models used by our state-of-the-art object detection algorithms.

To test the four suggested approaches an aerial image of 2×2 km of the city centre of Sint-Truiden was obtained. At a resolution of 25cm/pixel this results in a 8000×8000 pixel image, which is up-scaled using a bi-cubic operator to 16000×16000 pixels to ensure the solar panels are covered by enough pixels per panel. Inside this test image 313 solar panel installations where manually annotated, by drawing polygons around the installations, to use as ground truth when validating the fully automated object detection techniques suggested in this work.

The complete dataset, including training data, test data and annotations can be found for research purposes at `http://www.eavise.be/SolarPanelDataset/`.



Figure 5.4: Example of pixel based colour classification using neural networks as learning tool. (top) original image (middle) pixel classification result (bottom) cleaned up segmentation.

5.5.4 Suggested Approach

To find an optimal solution for fully automated solar panel detection in aerial images, a comparative study was performed. In the following subsections each of the four state-of-the-art approaches will be explained and discussed in detail. Section 5.5.9 will then take a closer look at the accuracy and time complexity achieved by each individual technique.

5.5.5 Pixel-based colour classification using support vector machines

Our pixel-based colour classification, as seen in figure 5.4, uses the internal colour area of each solar panel (blue-grey colour range) without looking at the bright edges of the panel, in order to ensure that the pixel colour distribution of the training pixels are separable in the hue, saturation, value (HSV) colour space. We manually collected 1000 internal solar panel pixels and 2000 randomly selected non-solar panel background pixels. Both solar panel and non-solar panel pixels are transformed to the HSV colour space, a space where separating colours is easier than inside the RGB colour space. Based on this training data, a support vector machine classifier with linear kernel is trained, able to autonomously separate solar panel from non-solar panel pixels.

When a test image is presented to the support vector machine classifier, each pixel is processed and is given a certainty score, indicating how certain we are that the pixel is actually part of a solar panel installation. This certainty score allows us to set a minimal certainty threshold, generating a binary image as seen in figure 5.4. On top of that, binary image opening and closing operators can be applied to remove noise, followed by contour detection and contour filling to achieve a cleaner result.

5.5.6 MSER based colour segmentation and shape analysis

Maximally stable extremal regions [92] is a technique used to detect blobs inside a given image. By systematically increasing the threshold on a given greyscale input image, from very sloppy to very strict, we create a set of sequential binary images. Inside those images the algorithm looks for regions that stay stable over the different thresholds and then approximates those regions by their fitted ellipse. Due to the higher response of solar panels in the blue channel compared to the red and green channels, we only process the blue channel data, removing the need to explicitly convert the RGB image to a greyscale image. Applying

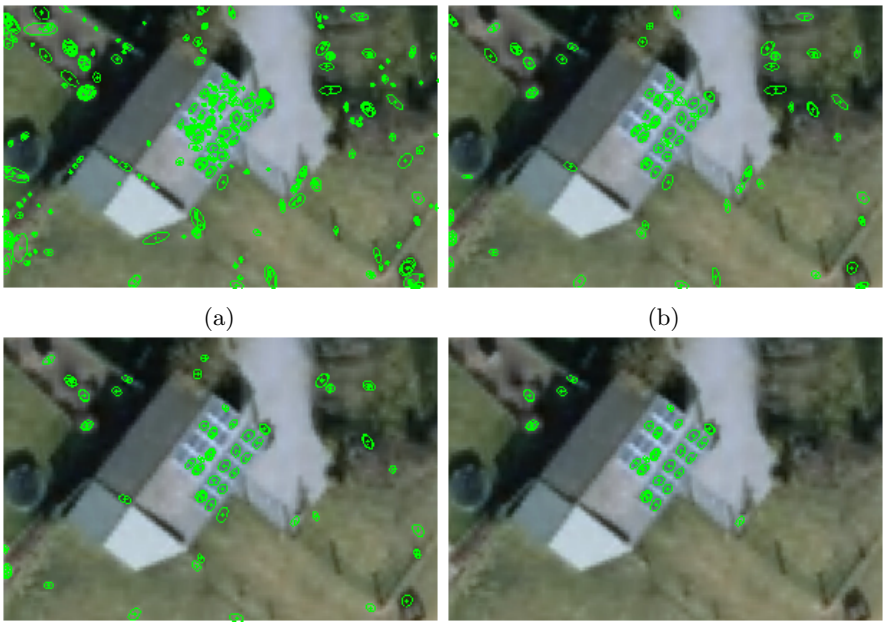


Figure 5.5: MSER based colour segmentation (a) MSER detected blobs (b) area restriction (c) ratio restriction (d) colour restriction.

the Maximally Stable Extremal Regions (MSER) algorithm generates a selection of blob candidates, as seen in figure 5.5a. To further filter the obtained regions, we start by discarding blobs of an incorrect size (see figure 5.5b), then look for blobs with a axis ratio that deviates maximally 30% of the ideal 1.5 : 1 ratio (see figure 5.5c) and finally we apply HSV colour segmentation on the remaining blobs, using the technique discussed in subsection 5.5.5. The known size range of the blobs can be explained by the fact that aerial imagery is taken on a constant hight, while the limited ratio deviation is explained by the fixed physical size of the solar panels. Finally, since our solar panel installation training set contains solar panels with a known and shared colour range in the HSV colour space, we can allow colour based segmentation for further filtering.

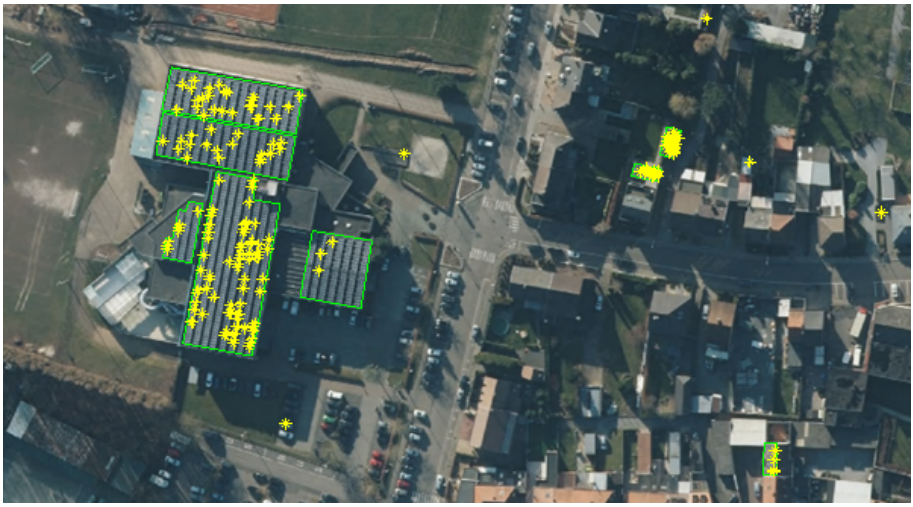


Figure 5.6: Detection of solar panels using a boosted cascade of weak classifiers.

5.5.7 Boosted cascade of weak classifiers

The previous techniques have only a very limited training time, since most processing is done on the fly when providing new test samples. This is different for object categorisation techniques, where a model is learned from a set of positive object samples and a large set of random background samples. From each training sample a set of specific features is learned that is smartly combined into a model, able to separate objects from non-object patches inside the image.

For our first object categorisation approach, we use the framework by [155], based on a boosted, using AdaBoost [45], cascade of weak classifiers (simple decision trees). This technique was originally developed for efficient face detection, but studies [110,111] have proven that this technique still achieves top-notch results in other application fields, focussing on more general object detection test cases.

The boosted cascade framework used does not incorporate colour information, but rather looks at the structure and texture of objects a greyscale image. It describes local binary pattern features [83], features that look at local gradient information in training samples. Since solar panels have a colour range that has a higher response in the blue channel, we decided not to use a greyscale image, but the blue channel instead, explicitly forcing the framework to use colour information.

A downside of the boosted cascade classifier framework, is that it trains a model

for a fixed orientation. For this we explicitly rotated all solar panel examples to a horizontal position, resulting in a model able to detect horizontal solar panels. However, solar panels occur in different orientations. To cope with that, we simply rotate the input image over different angles, with a predefined angle step, and then warp back the retrieved detections. This allows us to build a full 360 degree capable solar panel detector using a single orientation model.

When running the detector on a test image, the detector will apply a fully rotational sliding window based evaluation of the image, triggering a detection at each position that gets classified as an object by the trained model. An example of such a detection output can be seen in figure 5.6.

5.5.8 Aggregate channel features

ACF [33,34], the technique that we optimized for GPU in the beginning of this chapter, is in fact an extension to the this technique. Where the boosted cascade framework ignores colour information from the start, using ACF confirmed our premise that using colour information for certain object detection tasks can yield serious advantages and yield a higher detection accuracy. This is one of the main reasons why we decided to try this technique for our solar panel detection case.

Running this more recent object categorisation detector on top of a given test image generates similar output as the previous technique, as seen in figure 5.7. However, keep in mind that both figure 5.6 and 5.7 are a sample detection output at specific detection thresholds of the algorithms. Deciding which algorithm performs better will be done thoroughly in section 5.5.9.

5.5.9 Results

We started out with comparing our four state-of-the-art algorithms in processing time, as seen in Table 5.1. We can see a clear difference in training time between the more basic HSV pixel segmentation and the MSER approach on the one hand, and the object categorisation techniques on the other hand. While the boosted cascade and the aggregate channel features approach take quite a bit longer to train on the given dataset, this task should only be done once, because a trained model can be reused as many times as we desire. However, when looking at detection time, we notice a steady increase in processing time when computational complexity of the algorithm increases. Where the standard pixel based segmentation takes only 10 seconds for a 16000×16000 pixel image, the basic object categorisation framework already takes 600 times that long.

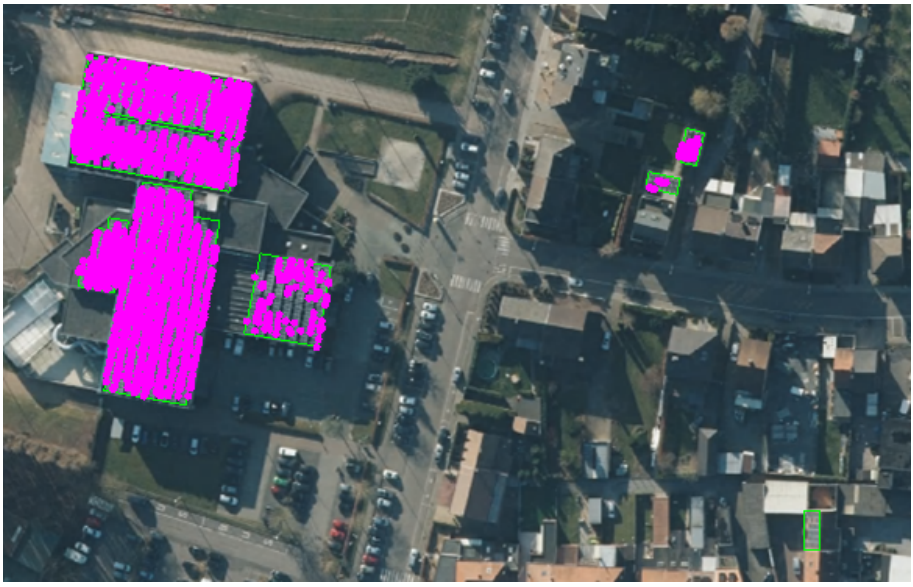


Figure 5.7: Detection of solar panels using the aggregate channel features technique.

Table 5.1: Comparison of training (given the training sets described in section 5.5.3) and detection times (based on the upscaled 16000 × 16000 pixel image) combined with the complete system configurations used. We used the following hardware: DES1: Intel(R) Core(TM) CPU i7-4500U @ 2.4GHZ, DES2: Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz.

Technique used	**Training time**	**Detection time**	**System**
HSV + SVM	10 sec	10 sec	DES1
MSER	0 sec	100 sec	DES1
Boosted Cascade	3.5 hour	10 min	DES2
Aggregate Channel Features	36 min	6 hour	DES2

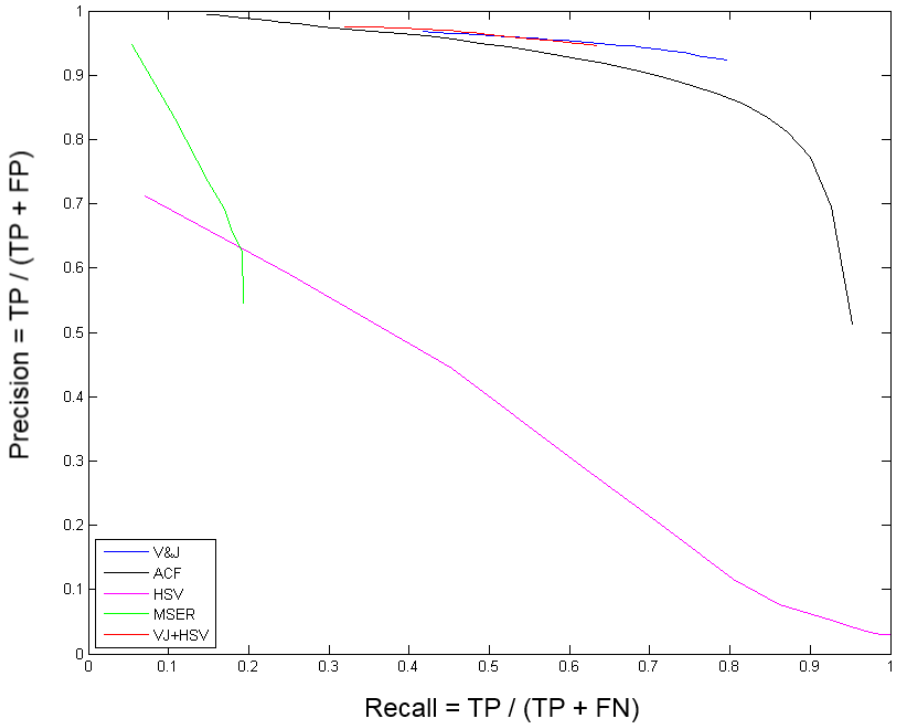


Figure 5.8: Precision - Recall curves for all techniques tested and validated on the 16000x16000 test image of the Sint-Truiden city centre.

However, these timings should be interpreted with caution because they highly depend on the available hardware, which is also specified in Table 5.1. Furthermore, the implementation of the aggregate channel features technique was developed in-house and was at the time of doing the experiments not yet optimized for parallel processing. The results of this then only server as a demonstration of the accuracy of ACF without looking at the results terms of speed. Taking into account the results we got at the beginning of this chapter, using a fast GPU can result in an additional speed up of about 3.8 times.

One of the main reasons why object categorisation techniques take a lot more time, is because these techniques are trained for a specific orientation. In order to be able to detect objects in every possible orientation, we rotated the original image for a full 360 degrees, with a single degree step. This again can be heavily optimized when desired, by reducing the number of rotation steps or for example rotating the model instead of rotating the image.

To evaluate the accuracy of the developed techniques, we suggest to use precision recall curves, used to compare the actual detection output with the manually obtained ground truth polygons. To generate these curves, seen in figure 5.8, the generated detection maps are first downscaled to a 4000×4000 pixels, combining scores of the detections obtained on the same locations. On top of the resulting score map, a threshold is applied (which is the varying parameter used to generate the different precision-recall values for each algorithm). This is followed by a dilation (make detection centres as large as solar panel dimensions) and erosion (remove detections that are lonely and not grouped) operations resulting in a clear binary image. This binary image is then compared to calculate the amount of true positives, false positives and false negatives, which are in turn used to calculate correct precision and recall values.

Figure 5.9 shows the comparison of that binary output which is in turn compared to the manual annotations. By using the three channels of an RGB image, knowledge of both results can be visually combined. Detection centres are visualised as dots on the red channel, the ground truth is visualized on the green channel and finally the detection output regions are visualised on the blue channel. Combining those three colour channels yields a set of pixel based classification labels. Cyan labels indicate true positive detections, black labels indicate true negative detections, blue labels indicate false positive detections and green labels indicate false negative detections. The visualisation is done for both the boosted cascade and the aggregate channel technique.

The precision-recall curves clearly show that boosted cascade based object categorisation techniques such as Viola & Jones and ACF outperform the other, more basic computer vision techniques. We notice that the boosted cascade technique performs still a bit better than the aggregate channel technique, which indicates that adding extra feature channels like colour and other gradient filters is overkill for solar panel detection. To test this theory we added HSV pixel based segmentation as a post-processing step to our boosted cascade detector, displayed as the red curve in figure 5.8. We see almost no increase in efficiency which made us to decide to drop this extra processing step, in order to obtain a smaller computational complexity and thus a faster running time.

5.5.10 Discussion and conclusion

Our goal is to test our GPU-optimized ACF implementation and evaluate it against other techniques on a concrete application: automatically detecting photovoltaic solar panel installations in RGB aerial images, considering the very challenging conditions due to the limited resolution, the limited amount of

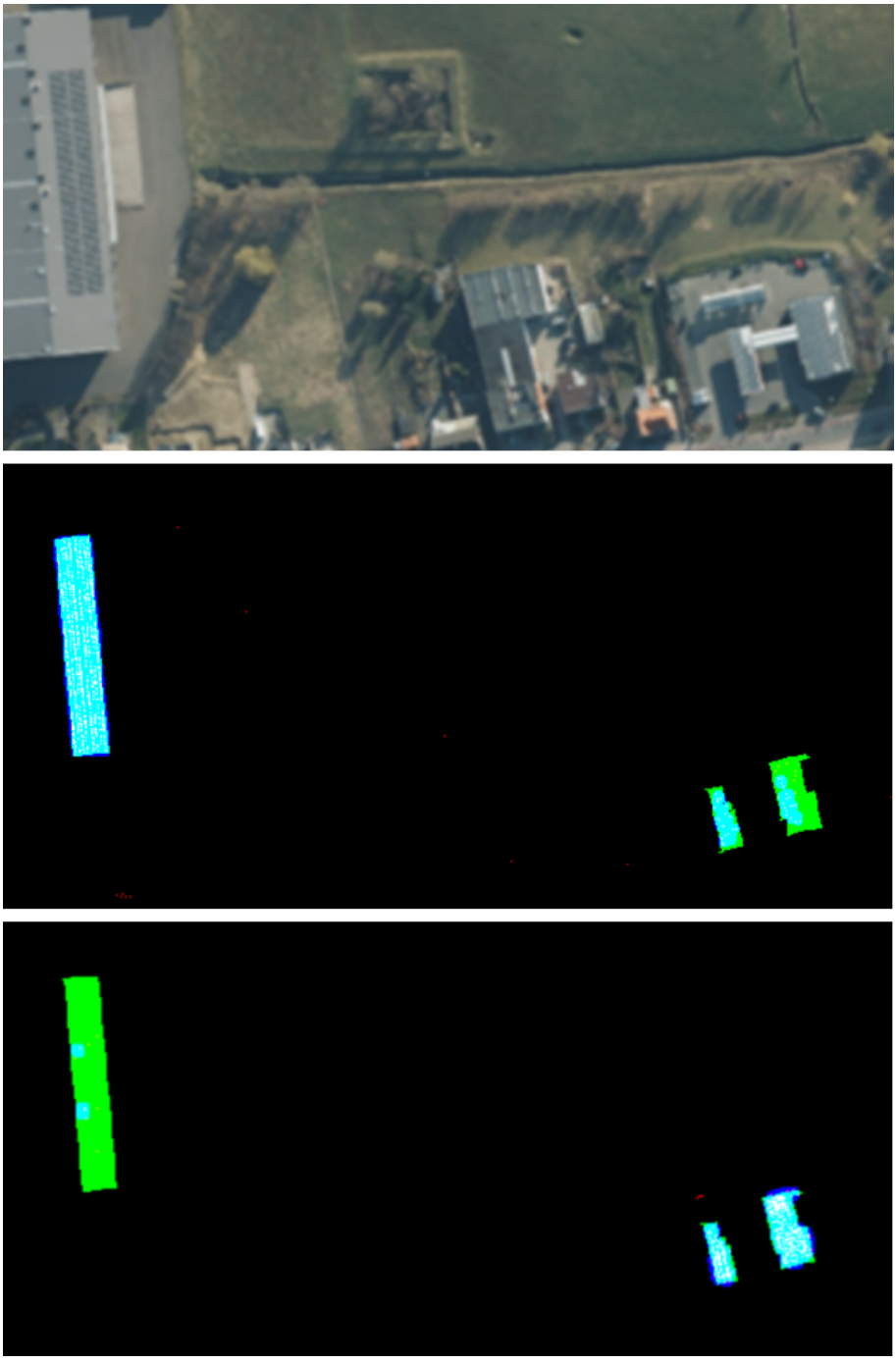


Figure 5.9: Score processing map for both Viola & Jones and aggregate channel features technique. (top) original input image (middle) boosted cascade result (bottom) aggregate channel features result

visual features, the existence of specular reflections and the different orientations of the solar panel object class.

Using pixel based segmentation for solar panel detection yields only moderate results. In any given test image, it is impossible to only separate blobs that are pure solar panels, since the colour distribution of the panels, also tends to return in many background patches. Furthermore, there are several solar panel types that tend to deviate from the average colour distribution, like the full black type, generating even more missed detections.

Applying smarter feature based techniques, like the MSER based approach, seems promising and even yields decent results in specific areas, but still has quite a fast drop in accuracy when considering larger regions where solar panels need to be detected, as seen from the resulting precision-recall curve based on the 2×2 km area of Sint-Truiden.

However, our tests clearly showed that object categorisation techniques like Viola & Jones and aggregate channel features can outperform more basic techniques with the only downside that training data needs to be collected and that the single training setup for building the model takes a bit longer. The advantage Viola & Jones has compared to ACF might have to do with the features it uses. Viola & Jones's features might be able to capture texture better then ACF's.

We proved that the tested techniques are a working solution for efficient and fully automated solar panel detection in RGB based aerial imagery for fraud detection. For a fully automated system, achieving a precision of 93% at a recall of 80% is not bad at all. Certainly if you take into account some considerations. Due to the heavy deformations a roofs' slant angle can introduce to the solar panel object, training multiple models to cope with the missed detections could be a first improvement. Furthermore, we performed the research on mid resolution images, at 25cm/pixel. However, there are already measurements of the same area at a resolution of 8cm/pixel available, which would again yield an accuracy increase. Since our training set consists mainly of industrial solar panel arrays, it is also quite understandable that the model is not able to cope with the larger deformations found on solar panels placed on domestic housing.

Finally, one could argue that the fully automated system should not be able to detect 100% of the solar panels with 100% precision. In most set-ups, an operator is still available for slight semi-automated input. We could provide that percentage of object that we did not found, or that yield a lower certainty, to the operator and ask him to define these cases of doubt by manual inspection. This suggested semi-automated approach would still reduce the manual labour drastically and reduce the time needed to process larger datasets.

5.6 Application 2: The autonomous hidden camera crew

This work was previously published in the proceedings of the 2017 MVA conference [144]. This research was carried out together with my colleague Timothy Callemein, my part mainly involved PTZ steering and detection.

5.6.1 Introduction

Interesting events occur in each household, both negative such as death, divorce, and disease, as well as positive ones like birth, adoption, and marriage. An independent video production house wants to make a reality TV show that will get families on TV in these situations, hoping to yield emotional and socially inspiring footage. Take for example first-time parents questioning how other couples manage to get up at night to care for their newborn child.

Recording a household with cameras is not groundbreaking in itself. Various TV programs have already explored this concept. However, the production house desired an innovative system that was able to automatically record using on-site hardware without the presence of a human camera crew. In this paper, we describe the solution we developed for this, composed of a system to decide which camera will be recording (section 5.6.4), and a controller for the PTZ to take cinematographic medium shots (as described in section 5.6.5). Remark that the video streams produced by our system form the input for a human film editor. In section 6.5 we present the results, both quantitatively and qualitatively by validating it using an experienced camera crew.

Problem type

In this work we try to follow people in their home using automatically controlled PTZ cameras. The PTZ cameras are controlled based on the detections of a person detection algorithm. Intuitively one would think that, because this is an application that needs to work in real-time, the runtime speed is the main constraint. However, since the type of shot that we are focussing on in this application is a steady shot (meaning that the shot should not change for a certain period of time), the system we will describe here is only "soft real-time". It doesn't matter that much if we are a bit earlier or later when changing shots, as long as we can make a good shot we are happy. When working on this project the production house even expressed the heuristic that, it is better to stay on the same shot a little bit longer when it is no longer the best shot to see the

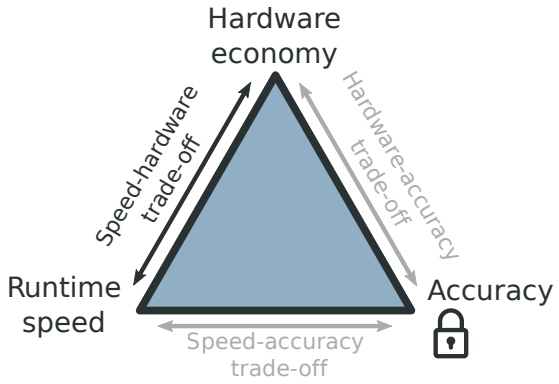


Figure 5.10: While speed has some importance in a real-time system, this application is only soft real-time. The quality of the shots, which is influenced by accuracy is much more important. We are left with a trade-off between runtime speed and computing power.

reason why it should be changed. For instance: When people walk outside of the scene, we want to see them leave the scene.

A more important concern is the quality of the shot. When we change to a new shot we want the quality of the shot to be good. The right people in the scene should be in the shot (following certain rules). To get the right people in the shot we need to know where they are, and for that we need a high quality detector. Detection accuracy in this application is much more important than either speed or required hardware. While hardware is not unimportant, when making setups at people's homes we used compute servers that where placed in hidden places like people's basements, making compute power less of an issue.

Figure 5.10 shows an accuracy constraint system in our triangle framework.

5.6.2 Related work

There are different camera network techniques available like [82,134], that mainly focus on security applications, mostly involving far away cameras capturing an overview of an environment. In this paper however, cameras will follow cinematographic rules that will be positioned at eye height and fairly close to what they are capturing. The presented system will decide on the spot whether or not to record, making it critical to be fast and reliable, as opposed to surveillance video that is permanently recorded and stored for future analysis. Other techniques like [29,37] that follow cinematographic rules edit the video

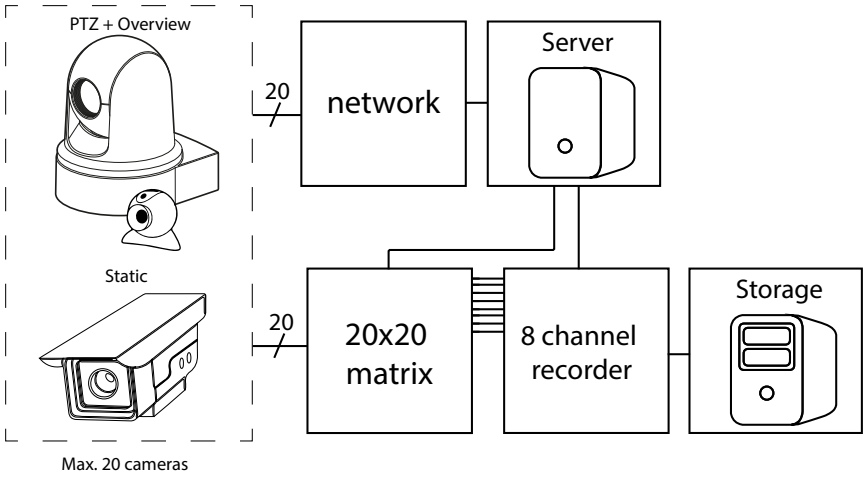


Figure 5.11: Hardware scheme

streams to produce one video montage or only focus on one room. Our system will select multiple cameras placed in multiple rooms and record them minimizing data collection providing only useful material that still needs manual editing. This relatively new research topic is being actively researched [25].

Automatically controlling a PTZ camera to follow people is not a new concept. In [65] a system is proposed for lecture recording that also takes into account some cinematographic rules. The lecturer is tracked in the PTZ video (no overview camera is used) using a combination of a detector and a tracker. They use a PID loop to continuously keep the lecturer on a fixed position in the frame. While such a system works well for lecture recording, recording people in their house is much more challenging. Our system is able to create shots of multiple people, conform to different cinematographic rules and take pleasing static shots.

5.6.3 Hardware setup

Figure 5.11 illustrates the hardware setup comprised of multiple Full HD cameras to be recorded. Although the recorder is capable of capturing eight channels simultaneously, we expanded this number by adding a matrix allowing up to 20 cameras in total.

The chosen ProRes LT 422 codec captures at an average of 102 Mbps, amounting

up to 1 TB per continuous camera stream per day. The proposed camera selection method will reduce the needed storage, extending the recording time greatly, and reducing the search space of the human editor.

Two types of cameras are used, static and PTZ cameras, both with a dedicated Serial Digital Interface (SDI) recording interface and IP interface for image processing and PTZ control. We add an overview camera to each PTZ, to detect actors and steer the PTZ as described in section 5.6.5.

We develop a semi-autonomous calibration procedure to calibrate the PTZ control parameters to the field-of-view of the overview camera. After a rough manual calibration, for each position SURF feature matching is used to correct the local parallax error and to fine align the FoV of the PTZ and the desired canvas cutout from the overview camera.

Our server processes the 20 camera streams in low-resolution through the network, allowing image based decisions that will select the cameras to record (described in section 5.6.4) and control the PTZ (section 5.6.5).

5.6.4 Camera selection

One of the camera crew's tasks is deciding when to record, reducing large amounts of useless video material. This requires an algorithm that is able to anticipate the whereabouts of the residents, and their movements between rooms.

To make the recording decision automatically, we need to understand what a camera crew uses as criteria for recording cinematographic scenes. In a reality TV show that follows the residents, the presence of a person is the primary trigger. In most cases, a script is used to determine when an actor is on set. In reality TV this is not possible. We have to use the entering or exiting of a resident as cue to start recording, which are shots that are also usable in editing. Movie professionals therefore prefer to have a pre-roll: shortly before a subject enters a room, the camera should be switched on.

Figure 5.12 shows a possible setup, consiting of three rooms and four partially overlapping cameras. On figure 5.12a two zones, blue and red, are doors connecting the three rooms. The red zone is visible on C_1, C_2, and C_3. Every room also contains a virtual "bucket" that holds the room's interest level (IL). The door in red, will raise IL of the buckets A and B. When a person is present in the red zone of a camera, the connected buckets will fill with water. Then, when a threshold is reached, we start recording the cameras connected to the

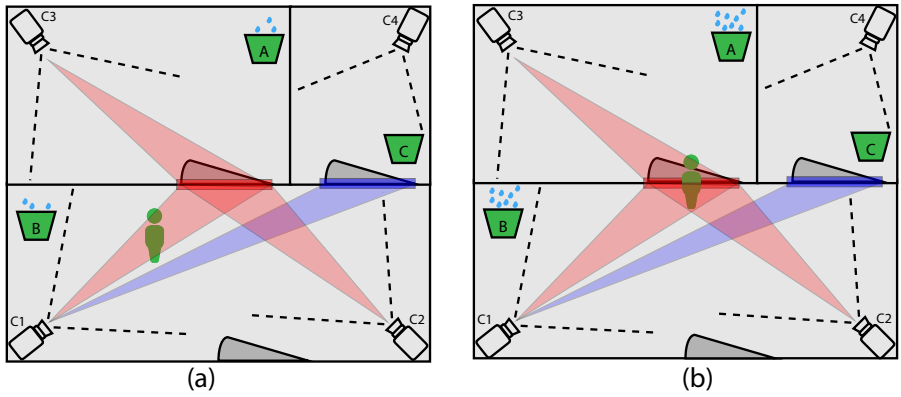


Figure 5.12: Top view with cameras, buckets and zones adding to the buckets.

bucket. This enables cameras that are in a different room to trigger a room before entering.

The cameras project the 3D environment on a 2D image, this means a person can be inside a zone, yet far away from a door opening, leading to false positives. Yet, as shown in figure 5.12b, multiple cameras will raise the IL faster, confirming the 3D location from multiple perspectives, providing more speed and reliability when triggering the recording. This, combined with different weights per zone, limits false positives and provides fine-tuning capabilities. The buckets also contain a "hole", that will lower the IL over time, greatly reducing the chances of accumulated low activity to trigger a false positive.

We use dedicated static cameras as well as the PTZ overview cameras to accurately monitor rooms inside a home. Since both of these cameras are in fixed positions, all movement in the video is caused by changes in the scene, meaning background subtraction (BGS) techniques work well.

In all of these cameras' frames we define zones (eg. a door opening) that fill up accompanying buckets. We determine the fill-up rate of a bucket by multiplying the amount of foreground activity in a zone with preset weights that we assigned to each zone. This ensures that eg. a cat walking through a door has less influence than a person.

5.6.5 Autonomous PTZ steering

The role of the PTZ camera in our setup is to replace a traditional camera man without the need for any human intervention. By adapting the three degrees of freedom of a PTZ to all kinds of scenarios that take place in front of the camera, we are able to make shots that are a lot more interesting than those captured by a static camera. To implement such an "intelligent camera man" it is essential to have a good understanding of 1. what is currently happening in front of the camera and 2. given the knowledge of the current situation how to capture it in a cinematographically pleasing way. Note that, contrary to other automated PTZ systems like [65] our system's main functionality is to make steady shots. For example, when someone is sitting on a coach watching TV, we do not continuously adjust the camera to keep the, sometimes slightly moving, person at a fixed position in the frame (often determined by the rule of thirds). Cinematographically speaking, such a "quivering shot" is not wanted. Instead, we wait for the person to become stationary, calculate what would be a good shot and then when we determine that the time is right, control the PTZ to make the shot.

Understanding the scene

In our application, the objective of our virtual camera man is to make shots of people. An obvious first step in creating such a system is to find out where people are located in the scene. To do this we compared different person detectors, like DPM [38,42] or ACF [32] combined with background subtraction and some scene constraints to speed up the algorithm. For accuracy (especially when dealing with occlusion) and because speed is not our main concern when creating static shots (real-time performance is not necessary) we choose to run a DPM upper-body detector. In section 6.5 we compare these detectors in more detail on our own dataset.

Apart from a person's location and size, which we get from the person detector, we also need to know the orientation and the position of the head inside the detection. For this, we use the approach proposed by [66]. This technique uses the Viola and Jones [155] face detector with three separate models, one trained on the frontal views, and the other on the left and right profile views of a face dataset. The results are then combined to estimate the gaze of the detected person. Because this method does not rely on small-grained facial features, it is proven to also work on small faces with limited resolution.

Apart from this, we also managed to speed up the detector by combining it with background subtraction. When we detect movement in the foreground

image we cut out bounding boxes around active areas. These areas are stored in a pool on which the detector is evaluated. The resulting bounding boxes of the detector (with some added margin) are then added to this pool to be evaluated again (together with the resulting bounding boxes of background subtraction) in the next image. Evaluating places where people were found in the previous image prevents people who are stationary, ie. invisible to the background subtraction algorithm, from disappearing.

Making cinematographically pleasing shots

The detector provides us with the necessary information to produce cinematographic shots: the location, size, position and general gaze direction for each person, detected in the overview camera. We give priority to keeping the maximum number of people in one shot, over making medium shots with the rule of thirds. This means that the distance between the outer detections will determine the width and the height by the 16:9 ratio of the shots. To provide enough space between the outer detections and the border, we add 15% to the width. Vertically, we place the estimated eye position of the highest detection to comply with the rule of thirds.

When there is only one detection found the height and aspect ratio of the detection is used as reference to determine the size of the frame. The position of the eyes is again used to determine the position of the frame on the y-axis. While the general gaze will now determine the position on the x-axis. When a person is looking to the right, we leave space to the right, when he is looking to the left, we provide space to the left. When there is no general gaze available, we choose to place the detection in the center. This is not optimal from a cinematographic perspective, but results in more usable shots than choosing one side as default. Some examples of the resulting shots are illustrated in figure 5.16.

Switching between different shots

As mentioned before, in our application, a continuously moving "quivering" camera is not desirable: we want to make steady shots. To make sure our camera makes steady shots and to prevent it from constantly alternating between different shots, we adjust the camera only if following conditions are met:

- The proposed shot is considerably different (determined by position and size) from the current.

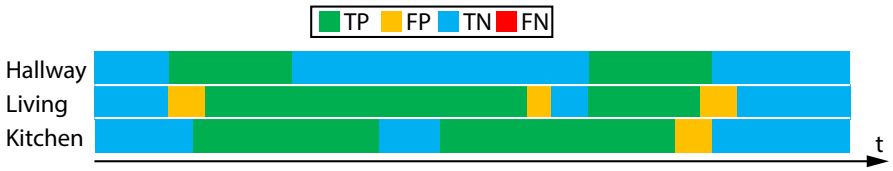


Figure 5.13: Camera selection results of three buckets during a 10 minute scenario.

- The new shot is steady, meaning people in it are only moving slightly or not at all.
- The old shot was kept a bit longer (2s in our case), to allow subjects to leave the scene, and has a duration of at least 6s.

If all these conditions are met, we map the proposed canvas from overview camera coordinates to PTZ control parameters.

5.6.6 Experimental results

In this section, we will report on quantitative results of the three main components of the proposed system (the camera selection, person detector and the cinematographical canvas selection method). However, as the proposed system is currently being used in a grand scale reality TV video production (8 families filmed 24/7 over 12 months), the very evaluation of the system will be in the success of that TV show.

Camera Selection

To test the camera selection algorithm we played a scenario inside the test environment, which included movements between different rooms. Figure 5.13 shows the analysis of three buckets being triggered, triggering recordings, during the 10 minute scenario, which is sampled every 10s and analyzed by comparing the actual output with the expected recording, as given by a real camera man. Since it's important to capture everything (and not lose any action) we set the sensitivity quite high, explaining the presence of multiple false positives (FPs) (triggering while no activity) and also the lack of false negatives (FNs) (activity but no or to late triggering of recordings). These results show that the buckets trigger the connected cameras on time, while not missing any vital video information. The accuracy of the tests by room are: hallway 100%, living

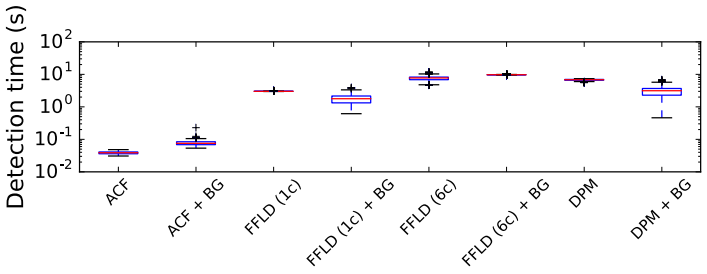


Figure 5.14: Comparison of the time complexity of different detection algorithms on our test footage.

room 87% and kitchen 95%. In this experiment, we saved 43.33% of disk space opposed to recording the full 10 minutes and recorded only 8% data overhead that contains no usable video data.

Person detector

In figure 5.14 we compare the time complexity of the different upper body detection algorithms on a test dataset that was recorded in a real setting (inside a typical home). We compare the algorithm with background subtraction enabled ("+ BG" in figure 5.14). While using background subtraction has a much faster average execution time, there are cases where the entire image needs to be searched causing a big performance hit, for instance when a person walks in front of the camera.

Figure 5.15 compares the performance of these detectors on the same dataset. We annotated a total of 105 frames with a total video duration of about two minutes. Some samples of this dataset can also be seen in figure 5.16. The DPM upper body detectors (`ffld_6c` and `dpm`) [38, 42] have the best performance in terms of accuracy, at the cost of being more computationally intensive, compared to the rigid ACF [32] detector which is not as accurate but much faster. When a faster reaction time is needed ACF would be the best options, when accuracy is prioritized one of the DPM detectors would be better. Limiting the search space by using background subtraction seems to have little impact on detector performance. Adding more people to the scene will only impact detection speed.

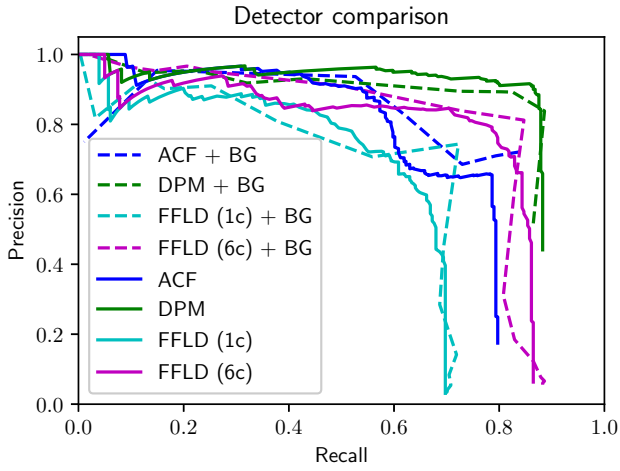


Figure 5.15: Detection performance on overview footage from a typical home.

PTZ canvas

Figure 5.16 shows the overview image with the detections and the resulting red frame that will be used to control the PTZ. As can be seen, the resulting canvases are designed to be cinematographically pleasing. To ensure this, we repeated the same 10 minute scenario 10 times, and continuously let the resulting canvases be evaluated by a group of trained camera men. Some video material of these tests can be seen at http://goo.gl/WIS0P4, containing video footage that is selected to be recorded (the multicam video) and multiple videos with detections and proposed canvases.

5.6.7 Conclusion

In this section, we compared ACF to other techniques for use in a system to automatically film indoor scenarios with multiple static and PTZ cameras, while complying to cinematographic rules. The first task of an autonomous camera crew is choosing whether to record or not. The test results show that our proposed bucket system is able to record all useful video streams with little overhead, decreasing the needed recording storage and search space for the human editor. The next task was using a PTZ that is able to take medium shots of people detected in an overview camera. Results show that the PTZ is

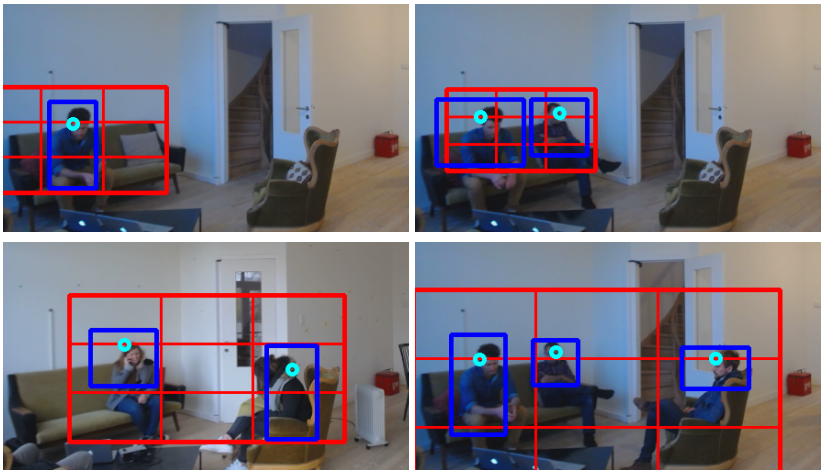


Figure 5.16: Proposed canvases (red) based on people detections (blue) controlling the PTZ. Video illustration on http://goo.gl/WIS0P4.

taking multiple cinematographically pleasing static shots of single and multiple persons based on the overview camera info. Remark that the system we have developed is currently being deployed in a real reality TV video production, providing us with more data to be validated.

5.7 GPU implementation

We can divide our GPU implementation of the ACF detector into two different steps: feature calculation and model evaluation. In this section we will explain both of them. In preliminary tests we saw that for the CPU version the feature calculation step took the longest (77% on the Jetson TX2 and 70% on a desktop system).

5.7.1 Feature calculation

Feature calculation on the GPU is quite straight-forward. It uses mainly primitive image processing operations that are already implemented in GPU libraries. We use the NVIDIA Performance Primitives (NPP) library, to do LUV colour conversion, smoothing, and to calculate the gradient and gradient magnitude. An advantage of calculating features on the GPU is that features

can remain in GPU memory, there is no need to do data transfers from host to GPU[1].

Histogram binning is done in a separate kernel we created ourselves. For each $n \times n$ patch in the gradient images we launch a separate thread. Each thread iterates over all pixels in its $n \times n$ patch and then divides the gradient magnitude at that position over two neighbouring bins. The result is stored in a separate histogram that is kept in private memory. When all bins in the patch are calculated we write the histogram bins one after the other to its corresponding channel in global memory. Keeping a buffer in private memory before we write to global memory ensures coalesced global memory access. Listing 5.1 shows the complete pipeline for feature calculation.

Listing 5.1: Feature calculation pipeline. Percentages indicate the amount of time spent during a step.

```
1. (15%) Copy input image to GPU
2. (15%) Convert rgb image to LUV
3. (7%)  Calculate gradient from
           L(uminocity)
4. (37%) Histogram binning
5. (26%) Downscale LUV /
           gradient magnitude
```

5.7.2 Model evaluation

While the feature calculation in the previous section was quite straightforward for a GPU, the model evaluation step is far from it. The way in which feature evaluation is done means that if we naively port the algorithm to the GPU i.e. by assigning each thread a separate window, a lot of branch divergence will happen. Windows that are done early (which is the majority) will idle while waiting for others (in the same warp) to complete[2]. Model evaluation is also mainly memory bound. The only real computation that needs to happen inside the kernel is the comparison of a feature value to its corresponding threshold, and calculating the next node to evaluate in the decision tree. The needed feature values are also sparsely populated throughout the memory making

[1] On the TX2 platform this is not a problem as memory is shared between host and GPU.

[2] In chapter 2 we explained how on NVIDIA's new architectures different threads in a warp do not have to execute in locked step anymore. This is not a solution the problem of branch divergence in this case. While different threads can diverge within a warp threads are still executed in units of warps. All threads within a warp still have to finish at the same time. Because of the uneven workload, threads with a small workload still have to wait on threads with a bigger workload within a warp.

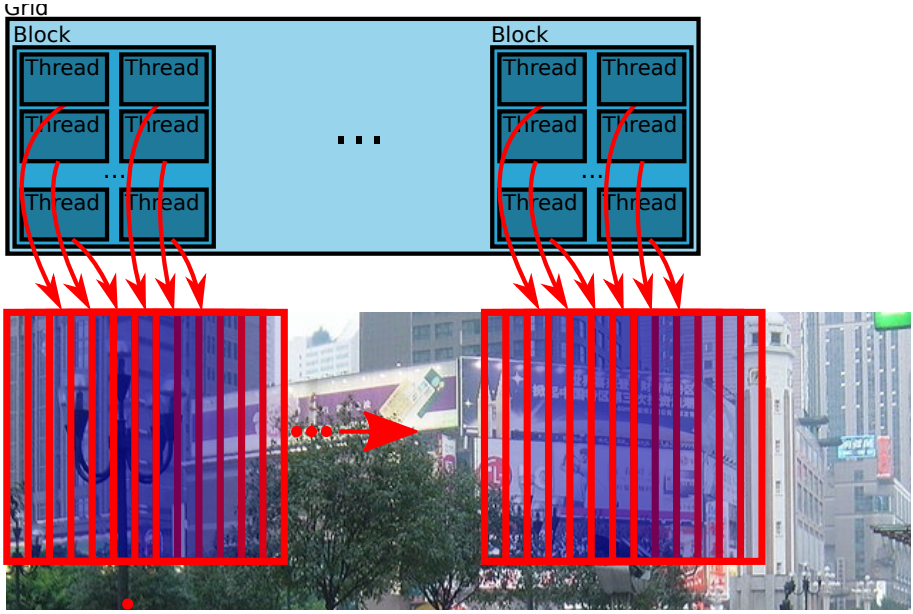


Figure 5.17: Instead of evaluating each window sequentially each window is assigned to a separate CUDA thread.

misaligned memory accesses a common occurrence. All of this means that it is quite challenging to get big speedups in the evaluation stage. In this section we will explain the different approaches we took to overcome these problems. In section 5.8 these approaches are evaluated in terms of runtime speed.

Naïve approach

As a first step, we made a naive implementation for comparison. As previously mentioned a naïve approach of porting the evaluation to GPU is by simply assigning each thread to a single window. Instead of doing each window one after another, we evaluate windows in parallel, see figure 5.17. As we will show in section 5.8, this approach on its own does not yield good results, because data is accessed sparsely throughout memory, and a lot of branch divergence occurs.

Course-fine detector

In a first attempt, we tried to gain speed by reducing the memory footprint during detection. Based on the approach of [107] who managed to speed up a part-based detector 10-fold, we divide the evaluation pipeline in multiple stages. The image is first evaluated using a coarse model which uses a higher shrinking factor. This results in a coarser feature map that is also much smaller. Using a smaller model means that it is much easier to keep features in cache longer which should yield higher performance, solving the memory sparsity problem somewhat. Detections that are not ruled out by the coarse detector are then given to a fine detector which is trained to affirm the coarse detector's verdict. While this approach would give speedups in theory, when testing the accuracy we could not reach the same levels as the original ACF implementation, because using a coarse detector lowers recall too much. In figure 5.18 we compare different configurations of the coarse-fine approach to our baseline CPU implementation. Since accuracy is an important property of a pedestrian detection algorithm, we decided not to pursuit this approach any further.

Stage parallel

Another option to parallelize the model evaluation of the ACF algorithm is to look for parallelism somewhere else. Instead of evaluating each window in parallel we can also evaluate decision trees in parallel. This comes at the cost of sometimes having to evaluate more trees than necessary. Groups of trees are evaluated at the same time, so if the results of the first trees in the group show that evaluation can be stopped, there is no way to stop the other trees in the loop as they were launched at the same time. Figure 5.19 gives an overview of this approach. Listing 5.2 shows the complete algorithm in pseudo code.

Hybrid window / stage parallel

The window parallel and stage parallel (section 5.7.2) approaches can also be combined into one. Digging deeper into the performance of the naive approach, we can assume that at the start of the evaluation pipeline most windows still have to be evaluated (large opportunity for data parallelism), while it is only at the later stages (when most windows can be pruned) that the naïve approach loses its advantage. It is at this transition that the "stage parallel-approach" starts gaining potential since the probability of pruning decreases with the number of decision trees that are evaluated per window.

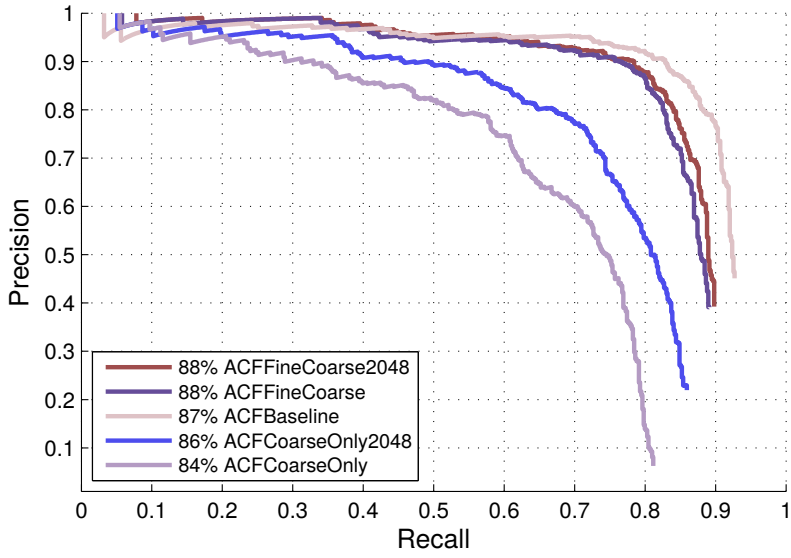


Figure 5.18: Comparison of the normal ACF to only the coarse detector and the coarse fine detector. Detectors with the 2048 suffix have a coarse detector trained up to a maximum of 2048 weak classifiers, the others only use 128 weak classifiers for the weak detector. "ACFBaseline" is our baseline ACF CPU implementation.

Listing 5.2: Overview of the stage parallel algorithm

```
for (i = thread_id; i < num_trees; i += block_size)
{
    added_score = walkTree()
    total_score += blockReduceSum(added_score)
    __syncthreads()
    if (thread_id == 0 and score <= threshold)
        break
}
```

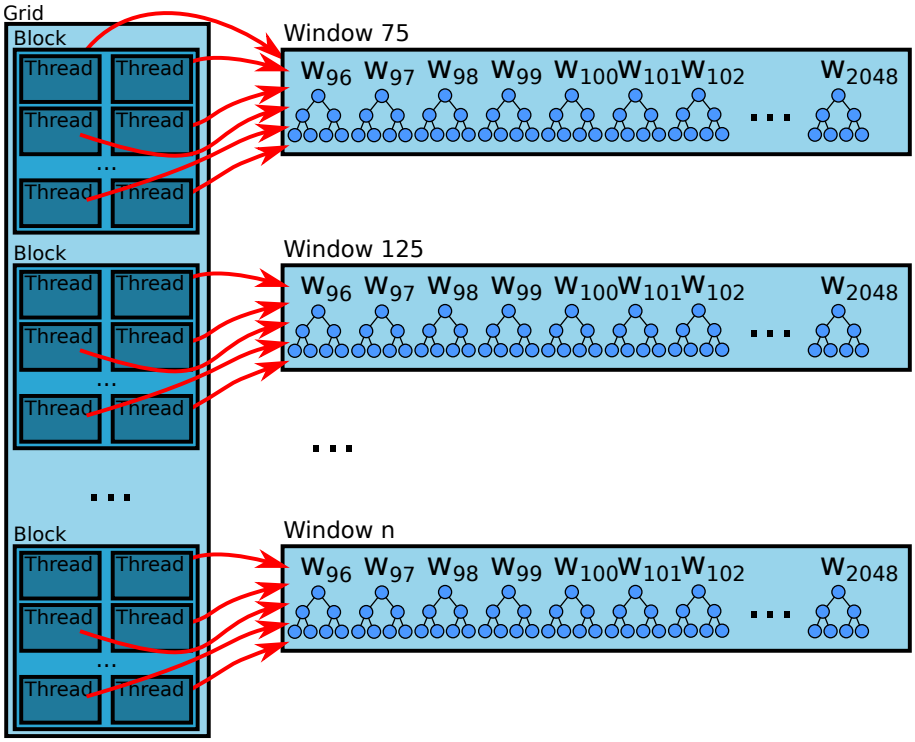


Figure 5.19: The stage parallel approach: each thread evaluates a separate decision tree.

As mentioned above we combine these two approaches by first evaluating n windows in parallel after which a separate kernel is launched to handle the remaining windows. To launch these kernels we use CUDA Dynamic Parallelism, a way to launch kernels from within other kernels [69]. Each thread that is assigned to a window that is not eliminated after N iterations will launch a separate thread block which executes the remaining decision trees in the window.

If dynamic parallelism is not available on the platform, we group the indices of windows that are still "alive" together into an array using a combine operation (from the thrust library [17]). Each thread in the stage parallel phase then executes a single item in the resulting array.

5.8 GPU implementation speed results

In this section we compare the different implementation methods in terms of performance. We evaluate our implementation on a desktop workstation (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, NVIDIA GTX 1080) and the NVIDIA TX2 embedded platform, hereafter called DES and TX2 respectively.

Tables 5.2,5.3 and 5.4 show a comparison of the average running time to process one image of a 1920x1080 video stream (TownCenter dataset [18]) of all our implementations. Feature calculation is something that scales quite well on GPU. This is clearly visible in table 5.2. We get an order of magnitude or more speedup compared to CPU on both devices. Also interesting is, that the performance of the Tegra TX2 comes close to that of the GTX 1080. This can be explained by the fact that processing a 1080p image really isn't that much work. The potential of such a powerful GPU is thus not fully exploited. The TX2 on the other hand has fewer CUDA cores and is utilized more fully. Also, on the TX2 there is no memory transfer cost (memory can be shared), compared to the GTX 1080 where the image has to be copied from host to GPU memory. Model evaluation on the other hand is quite difficult to port to the GPU. Using the window parallel or the stage parallel approach (section 5.7.2) on their own does not yield any speedups. Using the window parallel approach on its own means that a lot of threads are idling while their neigbours are still doing work. Using the stage parallel approach on its own means that too many threads need to be launched. A lot of them will do unnecessary work as they would be able to stop sooner had the weak classifiers be evaluated sequentially. Using a combination of both approaches however does yield a, albeit small, speedup on the GTX 1080 GPU. Although the speed-up of the evaluation part is limited, we were able to get a large speed-up in the most computationally expensive part of the algorithm. For the algorithm as a whole, we obtained a speed-up of 2.6x on the TX2 board, and even 3.8x on a desktop system compared to an already heavily optimized CPU implementation.

5.9 Detector comparison

Apart from comparing our implementation to a baseline ACF implementation we also find it important to compare it to the best performing object detectors at the time. In this section we evaluate how well our GPU implementation of ACF, and the ACF detector in general stacks up against the state-of-the-art YOLOv2 detector.

	TX2	
	Processing time (ms)	Speedup
Baseline (CPU)	223	1 ×
GPU	8.7	25.6×
	DES	
	Processing time (ms)	Speedup
Baseline (CPU)	74	1 ×
GPU	6.7	11 ×

Table 5.2: Comparison of different approaches for feature calculation.

	TX2	
	Processing time (ms)	Speedup
Baseline (CPU)	63	1 ×
Window par.	165	0.38 ×
Stage par.	933	0.068 ×
Hybrid	75	0.84 ×
	DES	
	Processing time (ms)	Speedup
Baseline (CPU)	31	1 ×
Window par.	76	0.4 ×
Stage par.	2228	0.014 ×
Hybrid	20	1.6 ×

Table 5.3: Comparison of different approaches for model evaluation.

	TX2	
	Processing time (ms)	Speedup
Baseline (CPU)	290	1 ×
Window par.	202	1.44 ×
Stage par.	1009	0.29 ×
Hybrid	112	2.6 ×
	DES	
	Processing time (ms)	Speedup
Baseline (CPU)	106	1 ×
Window par.	84	0.8 ×
Stage par.	2286	0.046 ×
Hybrid	28	3.8 ×

Table 5.4: Comparison of total processing times, with the exception of "Baseline (CPU)" feature calculation is done on GPU.

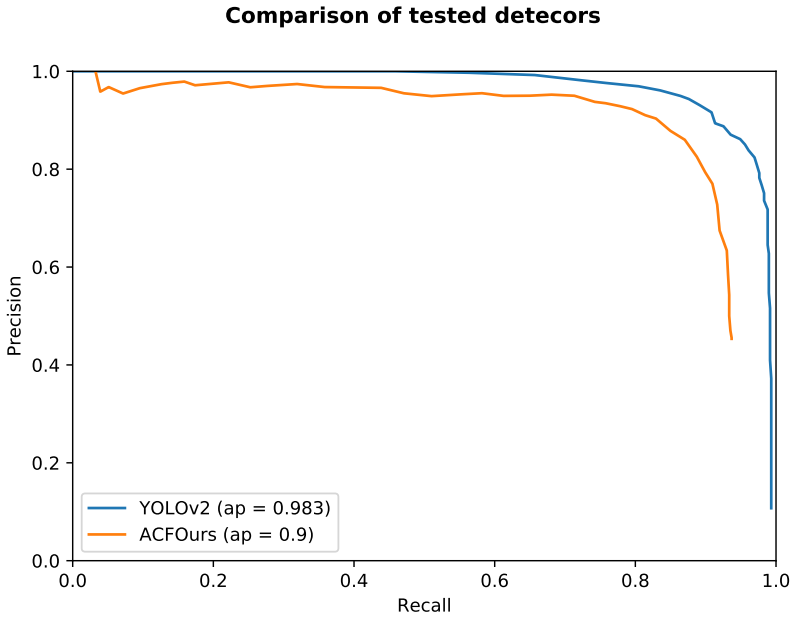


Figure 5.20: Comparison between tested detectors, "ACFOurs" represents our ACF implementations, "YOLOv2" is the standard YOLOv2 object detector.

In terms of accuracy the YOLOv2 detector appears to perform better than the ACF detector trained on the Inria dataset [28]. Figure 6.11 shows a comparison between both detectors evaluated on the Inria dataset. ACF has the lowest average precision. YOLOv2, the standard YOLOv2 multi object detector trained on COCO [85] does better than ACF with an average precision of 98% compared to 90%, both evaluated on the Inria test set.

While accuracy is an important property of an object detector, on an embedded platform running speed is a criterion that is at least as important. In table 5.5 we compare the tested detectors in terms of running speed. Because the main purpose of our ACF implementation was to use it in conjunction with some sort of scene constraints, which only require evaluation of one scale, we only have an implementation to evaluate one scale. The results in the table are estimations based on the speedups we got in section 5.8. As before the speed of the different detectors is tested on the 1920x1080 TownCenter dataset. In terms of speed the GPU ACF port still appears to perform better on TX2 compared to YOLOv2 (281 ms vs 343 ms). On the desktop GPU however, this is not the case. YOLOv2 is more than twice as fast as the ACF GPU port. If there is no

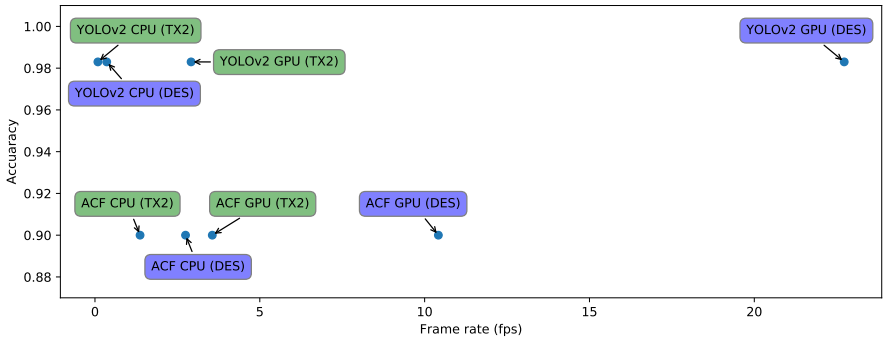


Figure 5.21: Comparison of the tested detectors in both speed and accuracy.

	DES (ms)	TX2 (ms)
ACF (CPU)	364	731
ACF (GPU)	96	281
YOLOv2 (CPU)	2805	11289
YOLOv2 (GPU)	44	343

Table 5.5: Comparison of total processing times of the tested detectors on both CPU and GPU.

GPU available on the system the clear winner is ACF. ACF is still capable of running quite fast on CPU alone. YOLOv2 is much slower making it unusable on CPU for time sensitive applications.

Figure 5.21 visualises the accuracy / speed trade-off on two different hardware platforms.

5.10 Application 3: Embedded Real-Time Object Detection for a UAV Warning System

In this section we highlight the last of our applications using ACF. This time using our GPU port.

This work was previously published in the proceedings of the 2017 ICCV workshop UAVision [137]. It was carried out together with Nils Tijtgat working at the Gent University at the time. My part in this work mainly consist of the evaluation of (GPU) ACF against the YOLO detector.

5.10.1 Introduction

Unmanned Aerial Vehicles (UAVs) have recently evolved from nice to have gadgets to full-fledged industrial grade workhorses in many domains. Precision agriculture [24, 133], industrial inspection [91] and package delivery service of the future [62] are only a few examples of the many domains where UAVs can prove their worth. UAV sales have as such immensely increased these past few years, and reports indicate this growth will continue well into the second decade of this century [48]. This fast growing market has sparked investments and advancements in UAV technology, and economy of scale has brought the cost down. Recent consumer devices display an impressive set of specifications and features at an affordable price. Industrial UAVs are meanwhile equipped with high-resolution sensor arrays to perform a very diverse set of applications, and are rugged enough for all-weather outdoor operation.

Parallel to UAV technology advancing, small form factor computing boards for embedded applications have become integrated powerhouses that enable edge computing on mobile devices. Running convolutional neural networks on embedded systems has become a reality.

Combining the technological novelties from both the UAV and embedded computing domain allows for some very interesting new approaches, which we will apply in this section from an emergency operation management perspective. UAVs can provide invaluable visual information during incidents and help rescue workers and coordinators better tackle the situation at hand. Aerial imagery can indicate possible entry-points during a fire, help locate missing persons during a search & rescue operation, locate possible dangerous containers on-site, etc. This is one of the issues we will address in this section: how can we perform accurate real-time object detection on-board a UAV, and integrate that into an emergency situation warning system? We specifically choose UAV on-board object detection, as a robust wireless link during an emergency is not always guaranteed, or the link lacks the bandwidth or availability to reliably support a video stream.

As features and application domain evolve at high pace, the major current UAV technology Achilles' heel remains the limited flight time. Even though lithium-ion battery technology is rapidly advancing, future technological improvements promise evolution rather than revolution when it comes to maximum energy capacity. Peripheral hardware weight and power consumption on-board our UAV should therefore always be carefully considered and kept as low as possible, limiting the available performance for our object detection algorithm.

In addition to that, the topic of this research will also serve as an interesting use case in trying to find out to which extent our ACF GPU port is capable

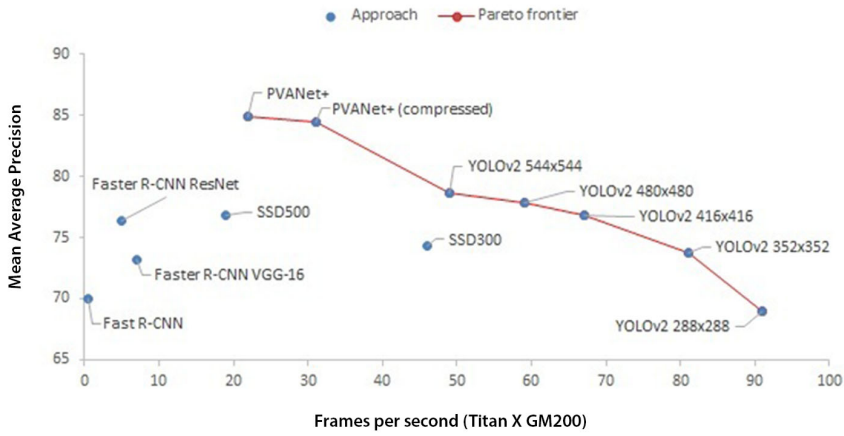


Figure 5.22: Comparison of different object detection algorithms tested on Pascal VOC2007 data (trained on VOC2007 and VOC2012). Image courteously provided by AlexeyAB from his darknet GitHub repository (`https://github.com/AlexeyAB/darknet`).

of running on board of a UAV. We also compare it to some of the latest deep learning based techniques for object detection.

Problem type

In this work we investigate which is the best detector for (dangerous) objects and people detection in a decision support system for firefighters. The system needs to be able to detect objects on board of a UAV without processing somewhere else. This puts a clear constraint on the hardware we can use. In this work we use the NVIDIA TX2 platform as it is lightweight and low-power enough to be put on a drone. Given the hardware platform, we *explore* the best detection algorithm and also try our *optimized* GPU version to find the best trade-off between speed and accuracy. Figure 5.23 depicts a hardware constrained algorithm in our triangle framework.

5.10.2 Approach

The context in which we report the findings of this application, is that of a UAV platform capable of arriving at an emergency scene on very short notice to

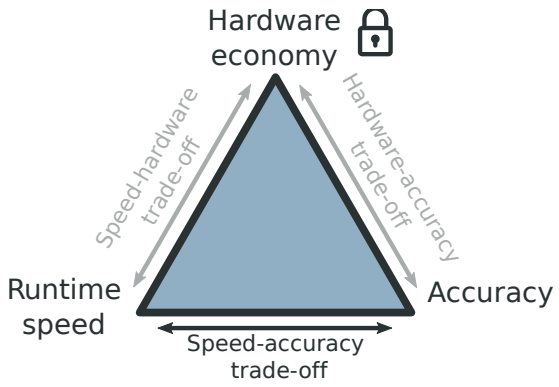


Figure 5.23: Because the systems needs to work on a drone, where power consumption and speed are important, the hardware constraint is fixed. We explore the trade-off between speed and accuracy further in this work.

provide increased situational awareness for emergency response coordination. The UAV could be stationed on top of the fire station or a regional launch site and depart as soon as a distress call rolls in. Using a combination of a 'straight as the crow flies' path and predefined paths at specific heights to avoid known obstacles in the area, the UAV can ignore traffic and autonomously arrive on scene faster than emergency responders can. Once on-site, it helps decision makers assess the situation more efficiently and coordinate the operation with greater ease than would normally be the case. The UAV should have a high degree of autonomy and accept high level commands to perform in-depth inspections of the area (including e.g. capturing additional imagery of heavily impacted areas, monitoring operational progress). An on-board high resolution optical camera offers decision makers a thorough view on the situation. To assist operators even more during stressful situations (where information overload can occur at any given time), an automated decision support system notifies of ongoing relevant events. As an example: if a rescue worker approaches a dangerous situation (high pressure gas pipe, burning gas tank) a warning is generated to alert those involved of the potentially imminent danger. One vital module of this decision support system is the object detection algorithm that looks for predefined items and passes this detection on to the decision engine. In this engine, the location of the detected object is estimated so that reasoning can take place in correlation with different objects present (distance of a fire truck to a burning barrel for instance). This application focuses on the on-board object detection algorithm and integration with risk assessment. People, emergency responders, vehicles, dangerous goods containers, gas tanks, are some examples of the predefined objects we want our model to detect.

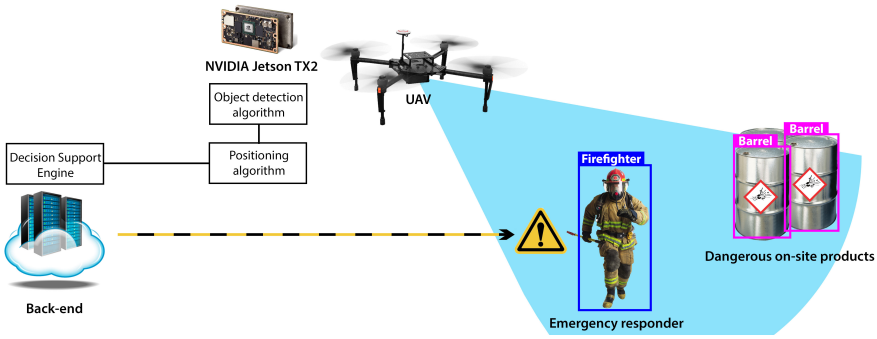


Figure 5.24: System architecture of a UAV-based decision support system for emergency response

5.10.3 Related work

In this section we discuss some of the related work in embedded systems and object positioning. The related work on object detection can be found in the beginning of this chapter, and in chapter 6.

Embedded platform

When it comes to choosing a UAV on-board embedded system for image detection, In [67] a survey of different platforms with their respective computing power and influence on system battery life is presented. The paper mentions the Jetson TK1 (predecessor of our Jetson TX2) and rates its CPU performance below average. This makes perfect sense: the Jetson is a powerful GPU embedded platform and the benchmarking algorithm used is CPU-based. The main algorithm we will be running is GPU based. The paper hence does not directly show us what GPU platform to choose, but the ideas and effects on battery lifetime of different embedded platforms remain valid and relevant.

Object positioning algorithm

Once our embedded system has detected objects of interest, we determine their position to feed this information to the decision support system. Preferably we want a solution that doesn't require a complex on-board setup (LiDAR or other expensive and/or heavy imaging system) to minimize cost and maximize flight time. In [138] Tijtgat et al. present a solution that matches this requirement and

provides a positional accuracy of <1.5m. The work presented in [79] proposes a vision-based UAV approach and landing scenario. This could serve as inspiration, with the limiting factor that the UAV would have to hover over the detected object's location to determine its position.

5.10.4 Architecture

The current architecture is the result of a set of design decisions upon which we will elaborate in the following subsections. As an example, Figure 5.24 illustrates what the architecture looks like for our use case. An autonomous UAV captures video data that the on-board hardware processes. In the example, a 'Firefighter' and two 'Barrel' instances are detected and their position is calculated. This information is relayed to the decision support system, that generates an alert if the firefighter gets too close to the dangerous products.

Object detection algorithm

The object detection algorithm we choose will have a major influence on the performance and battery life of our application. Many recent implementations are available, each having their merit. We compare YOLOv2 [114], TinyYOLO and our GPU ACF detector because of their excellent balance between frame rate and detection accuracy. Both YOLO algorithms run in the same Darknet framework, which allows us to easily run two different configurations and compare results. For ACF we use our own C++ implementation and its GPU port.

Embedded GPU processing platform

Since YOLOv2 heavily relies on the CUDA toolkit, we need to find a powerful GPU platform that is both constrained in weight, dimensions and power consumption. Modern desktop GPUs require big amounts of power to run under full load (NVIDIA GTX 1080Ti has a TDP of 250W) but deliver high amounts of floating-point operations per second (TFLOPS) in exchange (11.3TFLOPS in the example of the GTX 1080Ti). While the raw power of this type of GPU is perfect during the offline and unconstrained training phase, it is clear that we need a completely different approach to meet our requirements (minimal power consumption, limited onboard processing power, minimal weight) for the inference phase on-board the UAV.

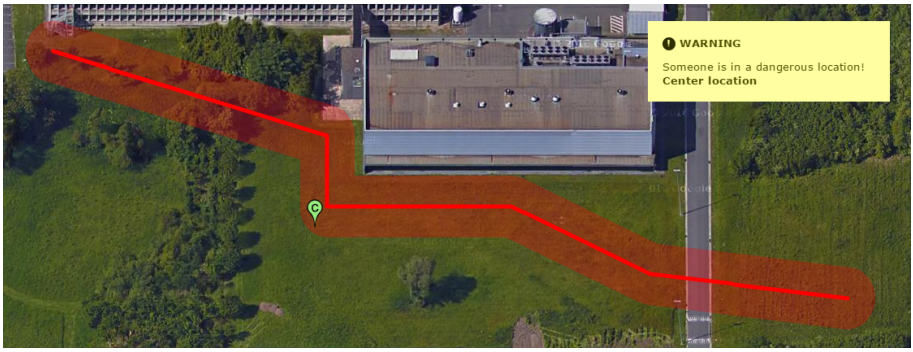


Figure 5.25: Example decision support interface

To determine if weight or power consumption should be the key parameter to consider when choosing our embedded platform, let's have a look at equation 5.2 [1]. This equation describes the power P (in Watts) required to hover an aircraft of mass m (in kg) with a propeller of radius r (in meters). K depends on the air density Q_{air} as defined in equation 5.3 and g is the gravitational acceleration with a conventional standard value of exactly 9.80665 m/s^2. At 20°C and a pressure of 1atm, K, an air density dependent coefficient, has a value of 0.363562254.

$$P = K \cdot \frac{(m \cdot g)^{3/2}}{r} \tag{5.2}$$

$$K = \sqrt{\frac{1}{2\pi \cdot Q_{air}}} \tag{5.3}$$

As an example, a commercially readily-available DJI Phantom 4 weighs in at 1380g total takeoff weight and has a propeller radius of 12cm. According to equation 5.2, it requires just over 150W of power divided over its four propellers to hover. This theoretical result does not take into account the power required to operate the flight controller and other on-board electronics and sensors. It's clear that the $m^{3/2}$ factor in 5.2 progressively penalizes every bit of added weight. Adding 100g of weight (+7.2%) to previous example increases the power required by 16.7W (11.1%). The weight of most relevant embedded computing platforms ranges from 40g (Raspberry Pi 3) to about 180g (Intel NUC Board NUC5I3MYBE).

Let's have a look at the power consumption of embedded computing platforms to determine if this influences the total UAV power consumption more than adding extra weight or not. Typical computing platforms intended for embedded applications tend to not use more than 15W under load (Intel NUC Board NUC5I3MYBE for instance), many even settle with a lot less (RaspBerry Pi 3 under load uses about 6.7W). Taking into account the previous thoughts and the fact that we need as much CUDA power as we can get in an embedded package, the choice for the NVIDIA Jetson TX2 was quickly made. At 85 grams (without expansion board or cooling fan) it presents 256 CUDA cores (1.5TFLOPS) and runs at either 7.5W (peak efficiency) or 15W (peak performance). It is twice as efficient as its predecessor (Jetson TX1) and features specifications that are currently unmatched in developer boards. The influence of the power consumption of the embedded board seems to be comparable to its extra weight.

Decision support system

The automated decision support system's role is to decrease the information load decision makers and actors on scene face during emergencies. In stressful and volatile situations details are easily overlooked, or there simply isn't enough time to assess the full context of the incident. The decision support system can provide helpful insights based on information gathered by the UAV or other data sources. An example of how this could look like can be found in Figure 5.25. The bold red line represents an underground gas pipe, the location of which can be acquired from an online API, or an offline cached database. A slightly transparent red area represents a security perimeter around the gas pipe that should be kept clear during incidents and the green marker represents an emergency responder, located by the detection setup running on our autonomous UAV. The event-based decision support engine reasons that a person is inside a restricted area and generates a warning to notify whoever is coordinating the operation of this incident. Many other incidents can be defined and processed in the same fashion: the UAV could autonomously search for wounded persons in the area, track the location of rescue workers and rescue vehicles on site etc. An evaluation of the decision support system is out of the scope of this section. In what follows we focus on achieving a real-time object detection on-board the UAV.

5.10.5 Evaluation

As the viewpoint of a low-altitude UAV distinctly differs from a traditional ground perspective, it makes sense to evaluate the different YOLO configurations using appropriate image and video material. The UAV123 data set [101] provides

exactly that. The dataset consists of partially annotated video sequences in 16:9 format (1280x720) captured from a UAV. However only one person is annotated, to calculate precision we filter out sequences where multiple people are visible. The following subsections discuss the various parameters we have analysed.

Input resolution

The input resolution of the Darknet neural network YOLOv2 (which will be extensively explained in chapter 6) defaults to 416x416. Every image subject to training or inference is resized to this raster. Increasing these dimensions will increase the time required for inference, but also results in a greater detection accuracy. This allows us to define either a low frame rate, high accuracy system, a high frame rate, lower accuracy system or anything in between, depending on the application. As every image or video frame needs to be resized to the defined input resolution, the dimensions of the input frame relative to the raster will have a major impact on the attainable frame rate. Figure 5.26 illustrates the increasing detection accuracy with increasing input resolution. From the figure, it follows that for YOLOv2 the accuracy goes up only marginally if the width is maintained, but the height increased to form a square input raster. This can be explained by the fact that the source images are all in 16:9 format. Resizing a rectangular image to a square raster introduces irrelevant information (YOLOv2 does not crop while resizing) that doesn't help detection accuracy. This deduction is not valid for TinyYOLO, a stripped down version of the YOLOv2 detector that runs faster then the original. A rectangular input resolution of 416x256 or 608x320 gravely harms the mAP.

Increasing the detection accuracy by means of a larger input raster comes at a hefty price as Figure 5.27 illustrates. Comparing the lowest accuracy (83.67% @ 416x256) to the best value (97.67% @ 1216x1216), we see the YOLOv2 frame rate drop from 6fps to 0.497fps, a 12-fold decrease. The Jetson TX2's limited memory prevented us from running TinyYOLO at a 1216x1216 resolution, but looking at the other results we see a comparable decrease from 11.80fps (416x256) to 1.67fps (1280x736).
Combining results from both previous plots yields an interesting conclusion: the frame rate difference between an input resolution of 608x320 and 416x416 is smaller than between 608x320 and 608x608, while the detection accuracy of 608x320 over 416x416 is greatly increased and there is almost no difference between 608x320 and 608x608. As such, when configuring a YOLOv2 detector it is advised to set the aspect ratio of the input raster to that of the images or video stream the network will process, in order to obtain an optimal frame rate and detection accuracy. In the case for TinyYOLO, a rectangular input raster should be chosen.



Figure 5.26: Input resolution compared to accuracy

Video input dimensions

To evaluate the exact effects the input video dimensions have on the frame rate, we chose a NASA 4K video[3] (3480x2160, 24 frames per second, 20.71Mbps bit rate, Public Domain Mark 1.0) named 'Red Lettuce', as it displays multiple categories we can detect using the default COCO and VOC trained models for YOLOv2 and TinyYOLO. We have subsequently downscaled this video to other 16:9 formats using ffmpeg and let both YOLOv2 and TinyYOLO process this batch. The input resolution was configured at 416x416 for every run. Figure 5.28 illustrates the results of this evaluation.
A first obvious conclusion we can draw, is that the frame rate goes down as the input video resolution goes up. This effect is not the same for YOLOv2 and TinyYOLO however: TinyYOLO is affected by this in a greater way. The TinyYOLO COCO frame rate drops from 14.87fps (256x144) to 4.91fps (3840x2160) a 66.98% decrease, while YOLOv2 COCO goes from 6.97fps (256x144) to 4.25fps,

[3] https://archive.org/details/NASA-Ultra-High-Definition

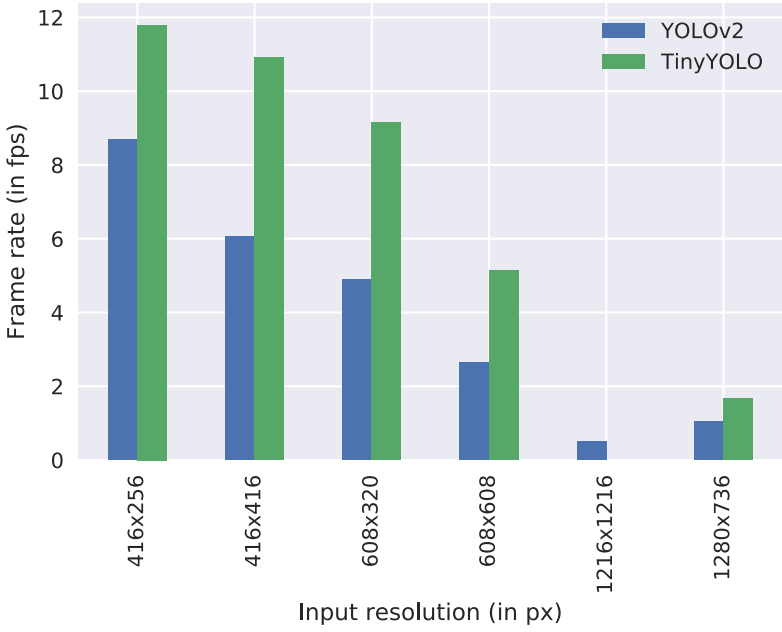


Figure 5.27: Influence of input resolution on detection frame rate

representing a 39.02% decrease. A second observation shows that the inference time greatly depends on the model size. Comparing the VOC (20 categories), COCO (80 categories) and YOLO9000 (9142 categories) models for YOLOv2, we see that as the amount of detectable categories increases, more time is required to perform the detection. As the amount of detectable categories in our use case will be well below 100, this will not pose a major influence.

ACF compared to Yolo

To see how YOLO compares to traditional detection methods, we compare the different configurations of YOLO and Tiny YOLO in this section to the aggregate channel features (ACF) detector proposed by Dollár et al. [32]. We used our own implementation of ACF (based in part on [31]) which was ported from Matlab to C++ for better performance. We compared it to our GPU implementation of the ACF detector which can currently only evaluate features at a single layer in the feature pyramid, but can still give us an idea about

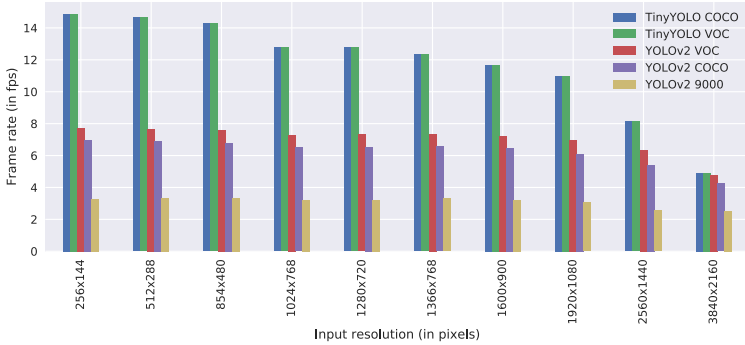


Figure 5.28: Influence of video dimensions on detection frame rate

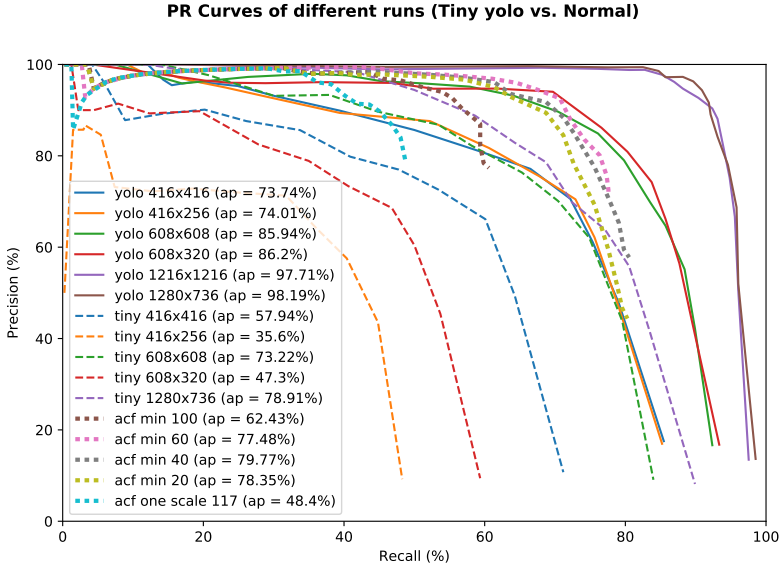


Figure 5.29: Comparison of the ACF detector to YOLO

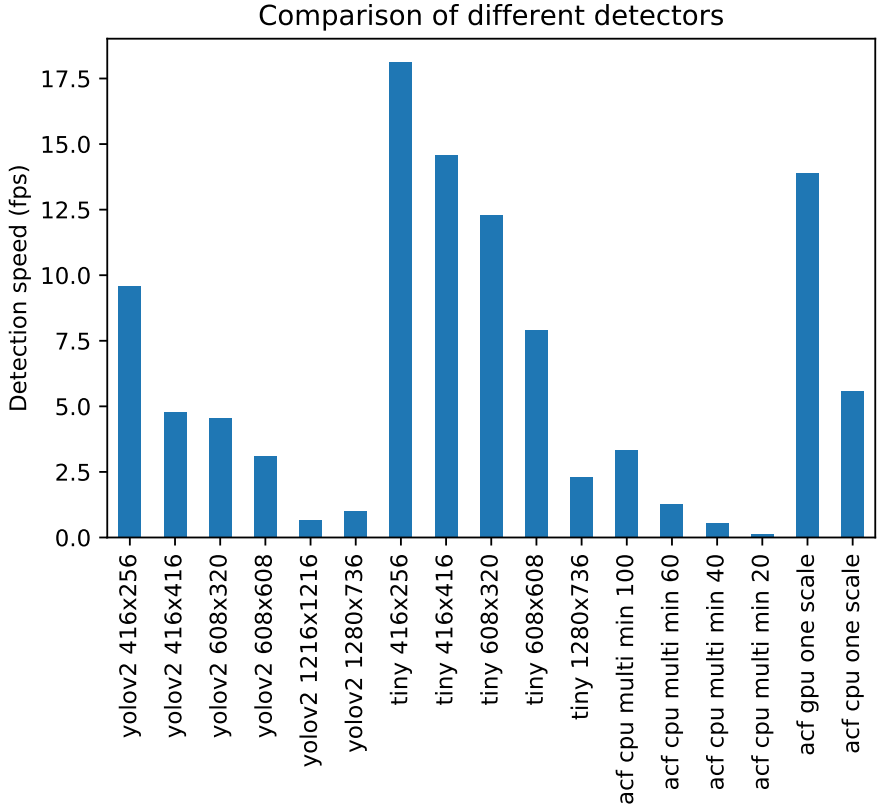


Figure 5.30: Comparison of different detectors in terms of speed

how well ACF would scale to GPU. To try to still make a fair comparison we assumed that the speed ups for one scale, scaled linearly to others.

The results in terms of detection accuracy can be seen in Figure 5.29. ACF (evaluated with a minimum detection scale of 20 pixels) shows similar performance to TinyYOLO evaluated at a resolution of 1280x736 pixels and YOLOv2 using an input resolution that lies between 416x256 and 608x320 pixels.

Figure 5.30 compares the speed of different detectors. The CPU multi scale ACF detector with minimum detection height of 100px is about as fast as YOLOv2 with a resolution of 608x608. Looking at the accuracy of both detectors, it follows that YOLOv2 outperforms ACF with an average accuracy of 85.9% vs. 62.4% suggesting that at least compared to the CPU version of ACF, YOLOv2

performs better at the same frame rate. Also note that the YOLO 608x320 variant attains about the same accuracy and is faster. Lowering the minimum height to 40px gives us the best accuracy for the ACF detector (ap = 79.7%), the closest corresponding YOLO detectors are YOLOv2 608x320 and YOLOv2 416x256 with an average precision of 86.2% and 74.0% respectively. The ACF detector reaches a mean frame rate of 0.54 fps while the YOLO detectors reach a frame rate of 4.53 and 9.57 fps respectively.

Previous results did not run on a GPU, which is a really powerful component on the Jetson TX2 when compared with the ARM CPU. To get equivalent results as YOLOv2, a GPU ACF implementation would need a speedup of at least 10× compared to CPU. Comparing the one scale implementation of ACF suggests that with the current implementation we can get a speedup of 2.5×. The biggest bottleneck here is the traversal of decision trees, which is a memory bound problem and thus hard to speed up on GPU. An additional speed increase in ACF could consist of the addition of scene constraints by using the height and orientation of the drone, and assuming a flat surface underneath, as was done in [31]. We can vastly narrow down the amount of scales that need to be searched and thus speed up the detector even more. Figure 5.29 shows the accuracy of the ACF detector at one scale, where we set the scale to 117 pixels (the average height of a person in the UAV123 dataset). The poor performance shows that it is indeed necessary to detect on multiple scales when using the ACF detection. Alternatively, with the knowledge of the earlier drone measurements we can greatly increase this accuracy by detecting on the correct scale. Of the currently evaluated detectors, the YOLOv2 detector seems to bring the best accuracy combined with the highest frame rate. Figure 5.32 shows a visual comparison of a selection of the evaluated detectors.

5.10.6 Conclusion

We have presented and evaluated our approach for a real-time object detection system on-board an autonomous UAV. Multiple configurations are available, depending on the frame rate and detection accuracy the final application requires. The presented evaluations can help configure future applications and demonstrate what is currently feasible using off-the-shelf hardware. The modern, neural net based YOLOv2 algorithm attains higher frame rates and detection accuracy results than ACF. Figure 5.31 presents a concluding view of how the different algorithms perform under varying conditions on an NVIDIA Jetson TX2, comparable to how Figure 5.22 did for several algorithms running on an NVIDIA Titan X (GM200).

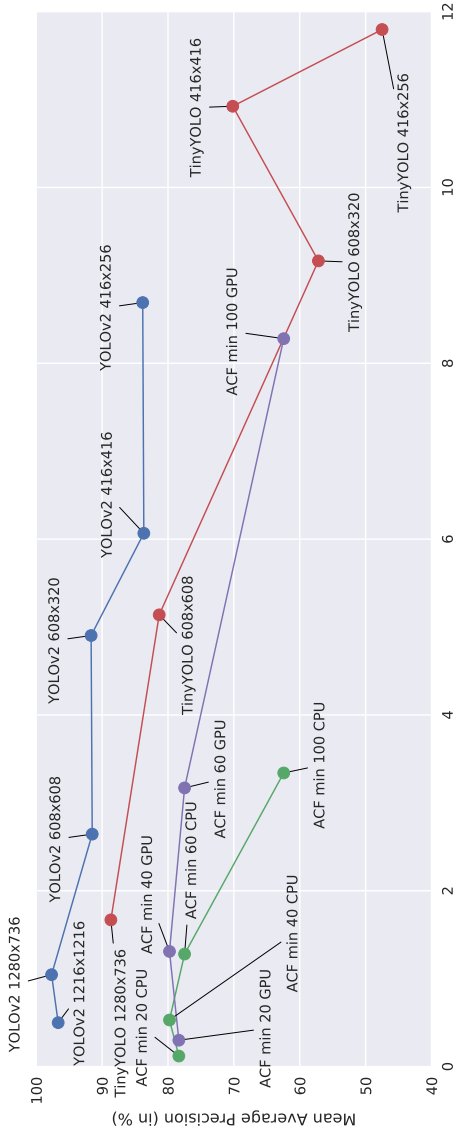


Figure 5.31: Mean Average Precision vs. frame rate for the discussed object detection algorithms running on an NVIDIA Jetson TX2

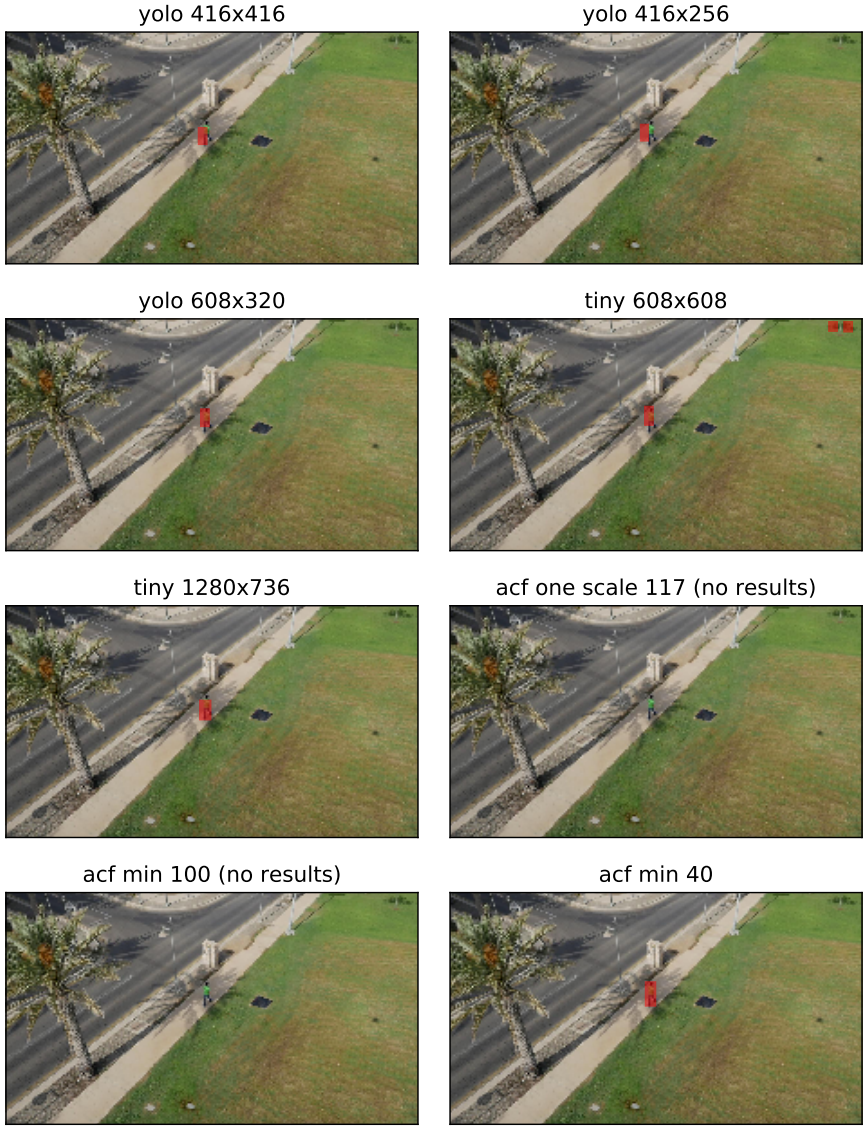


Figure 5.32: Visual comparison a selection of detector on a single frame of the UAV123 dataset.

5.11 Conclusion

In this chapter we described how a pre-deep learning detector which uses handcrafted features such as ACF, can be sped up by utilizing the GPU. We evaluated our GPU implementation on two platforms, the embedded Jetson TX2 GPU platform, and a desktop equipped with a NVIDIA GTX 1080. While the ACF detector does not lend itself easily to big speedups by parallelizing (especially the evaluation step) we still managed to get some significant speed increases on both tested platforms. Compared to the newer deep learning detector YOLOv2 we still manage to get the fastest detections using the GPU ACF method on a TX2. YOLOv2 is however more accurate, as could be expected taking into account the fast (deep learning based) evolution object detection techniques have seen in recent years. It is likely that in the future deep learning methods will overtake traditional methods in the field of real time embedded systems as they have with much of the rest of the field of object detection. As for right now, we would say that traditional methods still have their place on embedded platforms.

The tested deep learning methods do also require the presence of a powerful GPU. If no GPU is present, as is the case with many low-power embedded platforms (the TX2 is an exception in this case), traditional methods still win by a wide margin in terms of speed.

Chapter 6

Fast Simultaneous People Detection and Re-identification in a Single Shot Network

In this chapter, we tackle the last case in this PhD about a combined detection and re-identification pipeline. A traditional re-identification pipeline consists of a detection and re-identification step, i.e. a person detector is run on an input image to get a cutout which is then sent to a separate re-identification system. In this work we combine detection and re-identification into one single pass neural network. based on the YOLO architecture. We propose an architecture that can do re-identification simultaneously with detection and classification. The effect of our modification has only a negligible impact on detection accuracy, and adds the calculation of re-identification vectors at virtually no cost.

This work was published in the proceedings of 2018 AVSS conference [145].

6.1 Introduction

Re-identification (re-id), the task of matching person identities between different appearances of the same person is a well studied problem. Current state of the art techniques use deep convolutional neural networks (CNNs) trained using a

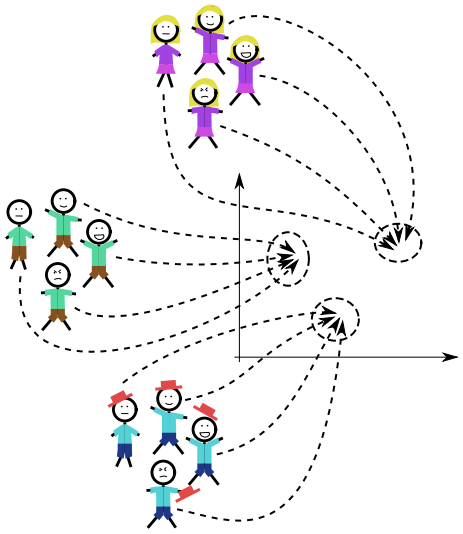


Figure 6.1: Illustration of the mapping from image to embedding space.

loss function like triplet loss [58, 128], contrastive loss [51] and on-line instance matching (OIM) loss [162]. The principle generally remains the same: the neural network transforms an input image into an (embedding) vector which contains distinctive features of the subject in the image (people in our case). This vector is then embedded in a space where vectors of same identities (e.g. coming from pictures of the same person) are close to each other, and vectors of different identities far from each other. Figure 6.1 illustrates this mapping. A distance measure (often cosine or Euclidean distance) in this vector space can then be used to measure how similar two objects are to each other.

Re-id can be useful in many applications: In the field of surveillance, it can be used to search for a person in a huge quantity of video sequences coming from different cameras. Other applications include recognizing people in photos which is useful for all kinds of purposes (e.g. social media, photo albums). Apart from that, re-id can also be used to make person tracking more robust. Matching identities in consecutive frames in a video sequence (with an optional interval) can reaffirm tracks created by a tracker. It makes it possible to overcome total occlusions and allows the tracker to be used in applications where non-overlapping multi camera handover is required. By treating the tracking problem as a clustering problem it can also be used as a tracker by itself.

The mapping from image to id-vector is generally done from a cropped image

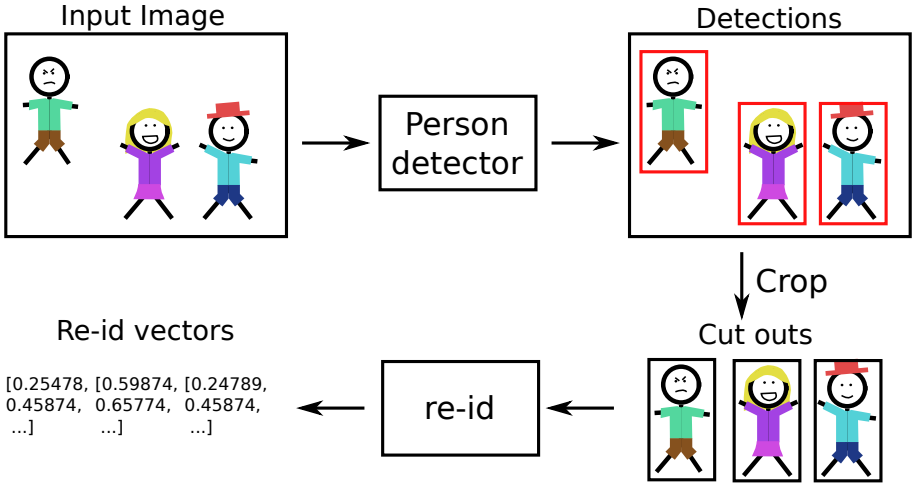


Figure 6.2: Classic re-identification pipeline, a detector is first run on the input image after which the detections are cut out and sent to the re-identification network.

containing one person. A two stage pipeline is used: first a person detector is used to locate all people in the image, after which cutouts of the image containing each a single person are sent to a re-id network which transforms them into an embedding space. To detect people in a video frame an object detection algorithm is used which can be based on hand crafted features (like ACF [33] or DPM [40]) or use a Convolutional Neural Network (CNN) pipeline (like YOLO [114], SSD [87], FPN [84] or Faster R-CNN [117]). Figure 6.2 gives an overview of the classic pipeline.

Indeed, to get state-of-the-art results a computationally intensive two-step CNN pipeline is mainly used. For off-line processing pipelines that require highly accurate results this is often acceptable. However, for systems that need to work in real-time on the other hand (e.g. a tracker) this often rules out re-id altogether. This combined process of detection and re-identification is often referred to as person search. Figure 6.3 demonstrates the difference between re-identification and this combined person search pipeline.

This raises the question: Is it not possible to combine this pipeline into one fast neural network that can fulfil both objectives at the same time? I.e. we envision a neural network that at the same time outputs detections and also mappings of these detections into an embedding space. As illustrated in figure 6.4. So far, combining these steps has only been attempted in a primitive way by Xiao *et al.* [161] with the more computationally intensive Faster R-CNN [117]. Whereas

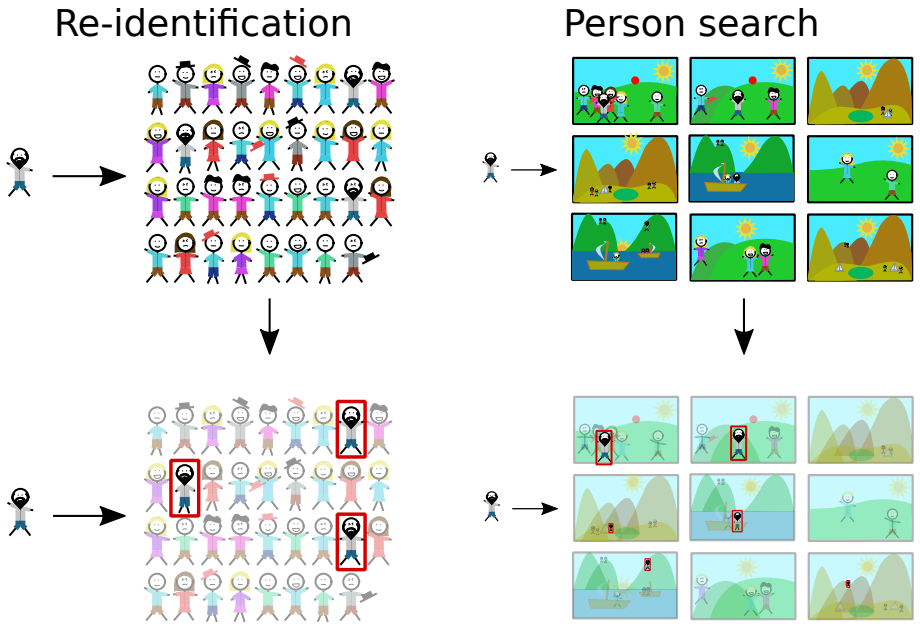


Figure 6.3: Comparison between re-identification and person search. A more detailed comparison can be found in the paper [162] of Xiao et al.

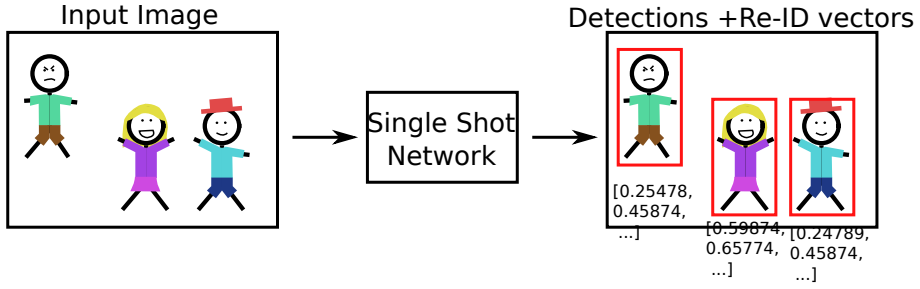


Figure 6.4: Proposed pipeline.



Figure 6.5: We create a neural network that can do both detection and re-identification at the same time. Left shows the Euclidean distance of a tracked person compared to all other output cells, right shows detection output and centre uses both detection and re-identification to create a tracker.

their focus lies more on accuracy, we focus more on speed.

In this chapter we investigate how this can be done using a single one pass fully convolutional network. In this chapter we introduce YOLO-REID, a single shot convolutional neural network based on YOLOv2 that is able to do both person detection and re-identification at the same time introducing only a negligible increase in computational cost and small penalty in detection accuracy. We make the choice of the YOLOv2 architecture because of its compactness (only 23 convolution layers), which gives it the ability to meet our real-time constraints. We of course do not expect the same accuracy as other re-id approaches like [59] which use the deeper Inception or ResNet networks. If better accuracy is desired, the proposed approach can however similarly be used in deeper networks.

To the best of our knowledge we are the first to attempt a combination of a single-shot-detector and re-identification. This approach has unique properties: it would be able to do detection and re-id at the same time without much extra cost. By running a detector you essentially get re-id for free. It would be possible to output an embedding vector of every person in the image without adding any extra cost independent of the number of people in frame. We demonstrate our approach by applying it to a tracker. The output is visually shown in figure 6.5.

The main benefits of our integration of re-identification in a single shot detector

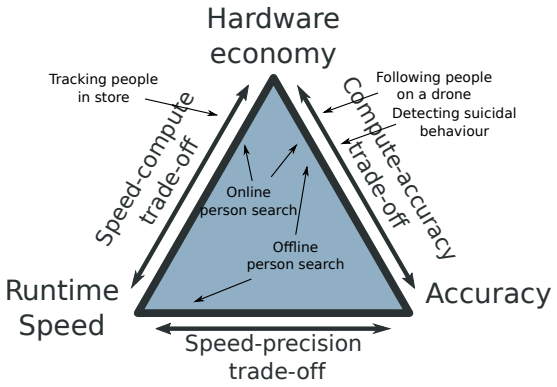


Figure 6.6: Visualization of different application, and where they lie on the triangle.

is primarily the speed at which a person search pipeline can run. While we do not directly have a particular application in mind, with the current results described in this chapter, many possible applications in both person search and tracking are possible.

Figure 6.6 gives an overview of possible applications together with their trade-offs in the framework. These are:

Offline person search The task of finding a query person in a big gallery of people is something that can normally be done offline. However, in recent times, more and more data from, for instance, security cameras becomes available. Processing this stored data can often take a huge amount of time even on powerful hardware. In the case of huge amounts of data we might say that either hardware or speed becomes a constraint, and accuracy becomes less important. In this case a trade-off has to be made between either speed and accuracy, or hardware and accuracy. Because of its focus on efficiency, this work lies more on the speed and hardware side of the spectrum, and is more removed from accuracy. Using deeper networks such as Resnet [136] in our proposed pipeline can however move it closer to the accuracy point.

Online person search An example of online person search is that of trying to find a suspicious person in live video feeds. Say for instance that authorities want to locate a fleeing bank robber. Given a picture of the robber, we want to know in which surveillance cameras the robber is present at the moment. In this case, video streams coming from all surveillance cameras have to be processed in real-time. Because of this,

we can say that the speed constraint is hard in this case. This leaves a trade-off between compute power and accuracy, by using more hardware we can run on higher resolutions, or use a deeper network for the same speed. Alternatively, if the lack of accuracy results in too many false alarms and it is tolerable to wait a bit longer (no more real-time constraint) we can use a more accurate version (accuracy constraint) of the system at the cost of speed or computing power.

Tracking Another possible application of this work is in tracking. By using the descriptive features that come out of the network we can make existing tracking approaches more reliable. Tracking on its own knows many applications with many constraints. Some of these include:

Following people on a drone Hard real-time application that is hardware constrained.

Tracking the path of people in a store Accuracy constrained. Having accurate tracks is important here.

Detecting suicidal behaviour in a train station It is observed that people that are about to commit suicide in a train station usually walk a pretty abnormal pattern before they go through with the unfortunate action. Using a data driven approach it is possible to detect this behaviour based on the pattern they have walked [142]. To respond on time the system needs to notify authorities in a short time span.

The remainder of the chapter is structured as follows: Section 6.2 will explain some of the related work. Section 6.3 will go into further detail on the architecture of our network and how we train it. In section 6.4 we demonstrate our work by using it as the basis of a tracker, section 6.5 compares our approach to both itself and others in terms of tracking, detection and re-identification accuracy. Our conclusions can be found in section 6.6.

6.2 Related Work

One of the motivations of this work is to use re-identification to create a reliable tracker. We will first briefly discuss the related work about tracking. Next, we will explain related work about re-identification and detection.

Multi-object tracking can be divided into two approaches. On the one hand there is tracking by detection which only looks at the output of an object detector, and tries to associate detection boxes to create accurate tracks [19, 105]. The

recent SORT [19] uses a simple Kalman tracker combined with the Hungarian algorithm and an association metric based on detection overlap to create an accurate multi-person tracker. Other approaches [20] use only a simple IoU measure that is able to achieve state-of-the-art results thanks to the recent advances in object detection accuracy. Off-line approaches have also been proposed [12,63,68,98,108]. Because they have all detection data available they can solve the track assignment problem as a global optimization problem.

Approaches that make use of appearance features [12,160,164] generally perform better than tracking-by-detection on its own at the cost of more computational complexity.

The rise of deep learning in the field of re-id [36,53,55,58,74,128,132,165,166,170] lies at the foundation of this chapter. From these works triplet loss [58,128] in combination with hard mining [58,75] still seems to be a top performer in terms of learning strategy.

Deep learning has also had a big impact on object detectors. Hand crafted approaches like ACF [33] and DPM [40] have for while now been having a hard time to keep up with deep learning pipelines (as was shown in chapter 5. Region based approaches like R-FCN [27] and Faster R-CNN [117] in combination with Feature Pyramid Networks [84] currently get the best results in terms of accuracy, while Single-Shot networks like SSD [87] and YOLO [114,116] offer an optimal trade-off in terms of accuracy and speed.

Recent work [161,162] has also tried to combine detection and tracking into one network for the task of identifying a single person or multiple people in a larger video set. They modify the Faster R-CNN [117] object detector. The region proposal part of Faster R-CNN is kept as is, and they concatenate the region proposal net of Faster R-CNN with a re-id network, creating in essence a detector that can be used directly for re-identification. Compared to our approach by using Faster-R-CNN they still use a two stage pipeline, albeit a pipeline that was previously used for detection and classification. The disadvantage of using Faster R-CNN however is that it is not as fast as a single shot network. Computational cost also increases when more people are in the scene.

Training multiple tasks at the same time can be a difficult problem. By adjusting different scaling factors in the loss function different tasks can train at different speeds. Ideally however we would like both tasks to train at the same time. The recent GradNorm [26] automatically adjusts scaling factors to learn both tasks at roughly the same speed. Another method is [72] which uses an approach that weights multiple loss functions by considering the homoscedastic uncertainty of each task. However, GradNorm [26] seems to perform slightly better.

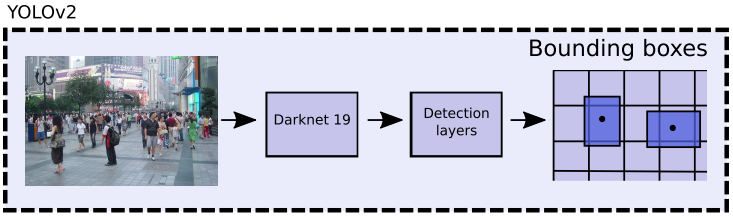


Figure 6.7: Simplified YOLOv2 network architecture

6.3 YOLO-REID

The idea of our person detection and re-identification network *YOLO-REID* is to create one network, based on YOLOv2 [114], that is able to accurately detect people in an image (same accuracy as standard YOLOv2) and calculate re-identification vectors (embeddings) on these people at the same time by evaluating the network only once.

6.3.1 Network architecture

Figure 6.7 gives a (simplified) overview of the YOLOv2 network architecture. An input image is first passed to the Darknet19 fully convolutional network, which outputs a set of feature maps which are used as input to an again fully convolutional network that outputs class probabilities and detections in the final layer. The output layer of YOLOv2 is structured as a grid of 13x13 cells (in the case of the default input size of 416x416) each containing k anchor points representing different bounding box sizes. Every anchor point contains a bounding box description (x and y-offset, a scale in x and y direction w.r.t. the anchor and the probability that this bounding box contains an object) and also a classification output. In our modified architecture we propose to also output an embedding for each cell in the output grid. For more information about the YOLOv2 architecture we refer to [114].

One of the goals in our proposed architecture is to be able to share Darknet19 weights for detection and re-identification, so we only need to pass through the network once. To do this we tried two architectures: "split end" (figure 6.8) and "mixed end" (figure 6.9). During training, we update all weights starting from the pre-trained Darknet19 weights trained on ImageNet. In both approaches the classification outputs from the original YOLO are removed, and instead the network outputs a 128-value embedding at each cell in the output grid.

"Split end" uses two separate layers (having the same size as the YOLO detection

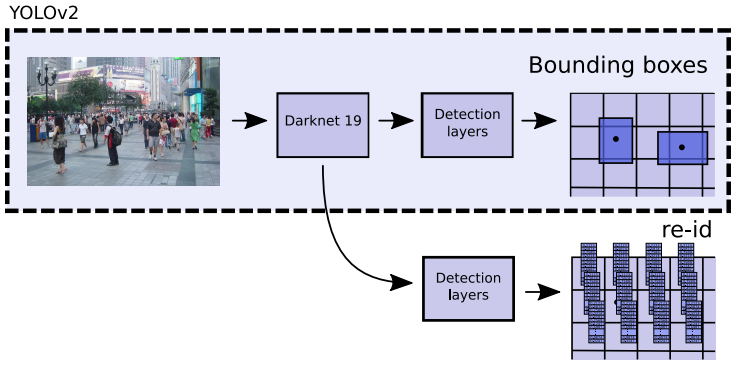


Figure 6.8: Overview of the "split end"-architecture: detection and re-identification are preformed separately.

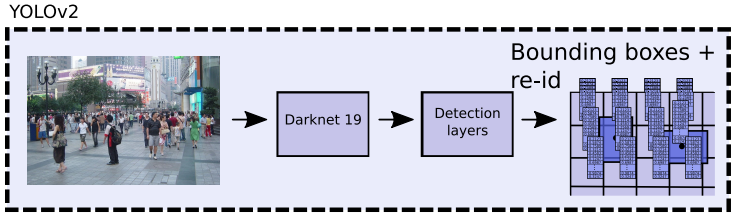


Figure 6.9: Overview of the "mixed end"-architecture: detection and re-identification layers are merged.

layers) that are kept separate from the detection pipeline, with the idea that the network can use these to learn a better embedding without impacting detection. "Mixed end" does away with this separate pipeline and fits everything in the YOLO architecture, creating a network that is smaller than the original YOLO architecture with 80 classes (see section 6.5.3). Training for both architectures remains the same. Our total loss function is the following:

$$L = L_{\text{region}} + \alpha L_{\text{reid}}, \tag{6.1}$$

where α is the weight of the re-id loss, L_{region} is the standard YOLOv2 region loss with the softmax classification loss removed, and L_{reid} is the re-identification (triplet) loss (see section 6.3.2).

The scale factor α is determined empirically in part, but also varies depending on the input resolution of the images used to calculate region loss during training.

6.3.2 Triplet loss

We use triplet loss [58,128] with batch hard mining as described by Hermans et al. [58], implemented in the open-reid framework[1]:

$$L_{\text{tri}}(\theta) = \sum_{\substack{a,p,n \\ y_a=y_p\neq y_n}} \max(D_{a,n} - D_{a,p} + \text{m}, 0) \tag{6.2}$$

Where $D_{x,y}$ is the Euclidean distance between two embeddings, a is an anchor embedding, p is a positive embedding (same person as a), and n is a negative embedding (different person than a). This loss function essentially tries to maximize the distance between a and n, and tries to minimize the distance between a and p within some margin m. If $D_{p,a} < m$ the triplet is not moved closer to the anchor. y_a, y_p and y_n are respectively the anchor, positive and negative ground truth identities.

As proposed by Hermans et al. the loss function using batch hard mining is given as follows:

$$L_{\text{Re-ID}}(\theta, X) = \overbrace{\sum_{i=1}^{P}\sum_{a=1}^{K}}^{\text{all anchors}} \max(m \overbrace{+ \max_{p=1\ldots K} D(f_\theta(x_a^i), f_\theta(x_p^i))}^{\text{hardest positive}} \underbrace{- \min_{\substack{j=1\ldots P \\ n=1\ldots K \\ j\neq k}} D(f_\theta(x_a^i), f_\theta(x_n^j))}_{\text{hardest negative}}, 0) \tag{6.3}$$

where f_θ is the neural net parametrized by θ that maps from image space to embedding space. For every group of samples (batch) of size P ($= 6$) and for every sample K within that batch, the hardest positive sample (the one furthest away from the anchor) and the hardest negative sample (the one closest to the anchor) is chosen to calculate loss. For every identity two instances are sampled from the dataset.

6.3.3 Multi target training

During training we calculate both losses separately. Region loss is trained in the same manner as standard YOLOv2 complete with data augmentation.

[1] https://cysu.github.io/open-reid/

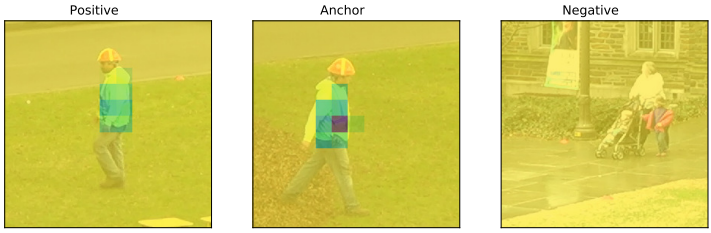


Figure 6.10: Example of the output of the network. The distance between anchor point and embedding is visualized, brighter means a large distance, darker means a small distance.

Our implementation is based on two free software PyTorch implementations: Lightnet[2] and pytorch-yolo2[3].

We use the CUHK-SYS dataset [161] as it seems to lend itself perfectly for this purpose. Instead of cropped patches it contains annotated persons (both id and bounding box) in uncropped images in a variety of scenes.

As explained in section 6.3.2 we use batch hard mining to train re-id. For every mini-batch a random set of ids and corresponding images is sampled from which the hardest positive and hardest negative is selected to calculate triplet loss. To know which images are the hardest, we make a forward pass of all images through the network. Since images within this mini-batch are selected randomly, we also use them to calculate region loss. For every image (not just the one selected to calculate triplet loss) we have bounding box annotations that we use to calculate region-loss in addition to triplet-loss.

We use triplet loss as explained in section 6.3.2 with that difference that we only learn the embedding-vector at the grid cell in the centre of the target person, the cell where an anchor point would also generate a detection. Figure 6.10 illustrates the desired outcome, cells at the centre of the positive patch (left) have a low Euclidean distance at the anchor point in the centre of the person (darker) and other samples (right and elsewhere in the images) have a high distance (lighter).

We use the recent GradNorm [26] to automatically adjust scaling factors to learn both task at roughly the same speed.

[2] https://gitlab.com/EAVISE/lightnet
[3] https://github.com/marvis/pytorch-yolo2

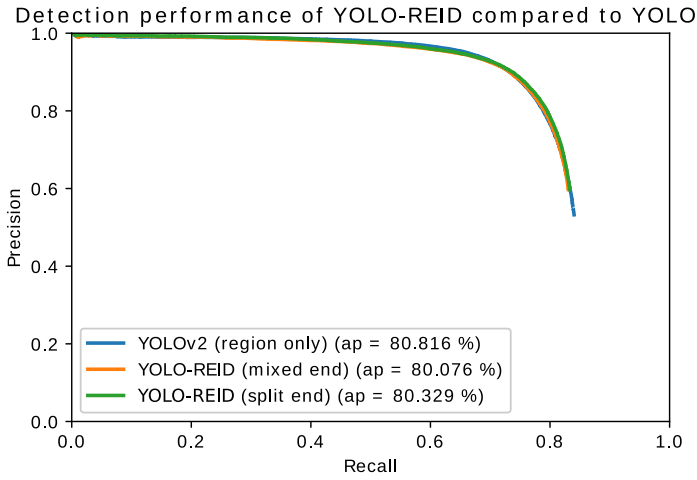


Figure 6.11: Comparison of our YOLO-REID network to standard YOLOv2.

6.4 Person Tracking

We demonstrate the usefulness of our approach by making a person tracker. Indeed, taking into account that the re-id vector computed by our 23-layer network will not be equally powerful as those computed by a full ResNet, we will test its usefulness in the reasonably less demanding application of tracking people within one camera view. We will now briefly discuss the functionality of this tracker. We base our tracker on the SORT tracking-by-detection tracker [19], which uses a simple Kalman tracker combined with the Hungarian algorithm and a simple IoU metric for track assignment. Similarly to [160], we extend this tracker using our learned re-id vector as an association metric. The tracker uses both detections and appearance embedding vectors coming from our network. In contrast to [160], in our approach the addition of an appearance feature comes at a negligible cost in terms of computational complexity. The computational complexity is also not influenced by the amount of people in the image. Appearance features are calculated by the network for every cell. A full evaluation of the tracking algorithm can be found in section 6.5.4.

	mAP (%)	Top 1 (%)
YOLO-REID (mixed end)	20.38	33.37
YOLO-REID (split end)	21.58	35.67
YOLOv2 + histogram	4.44	9.66
Resnet + YOLO	42.24	58.91
Xiao et al. [4]	55.7	62.7

Table 6.1: The re-id performance of our two architectures compared to colour histogram matching, which has a similar computational cost.

6.5 Results

In this section we evaluate the performance of our network. We will first evaluate the detection performance, compared to our standard YOLOv2, trained and evaluated on the CUHK-SYS train/test-set. Next, we evaluate the network using the criteria of person search. We calculate average precision and top-1 accuracy on the test set of CUHK-SYS yielding a combined figure for detection and re-id. We also give an overview of the computational complexity of the different architectures. We evaluate the tracker using the MOT16 tracking challenge metrics.

6.5.1 Detection

Figure 6.11 compares the standard YOLO people detector to our YOLO-REID network with embedding vectors built in. Both detectors where trained on the same CUHK-SYS dataset [161], "Mixed end" uses shared weights up to the last output layer of the network (figure 6.9), "split end" uses a different approach where the two last output layers are kept separate (figure 6.8). We can see that detection accuracy does not seem to suffer much by replacing the multi class classification with a re-id vector. Compared to "mixed end", "split end" seems to perform better by a small margin. This can be expected considering the extra layers in "split end".

If we add a deeper head to the network (Resnet50 [57] in this case) we an see that the accuracy increases even further.

6.5.2 Person search

Person search is the combination of detection and re-id merged into one task. Given some query images together with the annotated query person, it is the

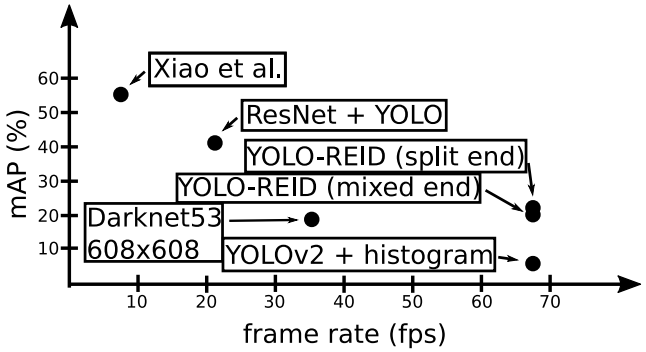


Figure 6.12: Comparison in speed and accuracy.

	Parameters	MFLOPs
YOLO-REID (mixed end)	50 704 761	29 370.03
YOLO-REID (split end)	62 505 337	33 357.24
YOLOv2 (80 class)	50 983 561	29 464.17

Table 6.2: Computational cost of different networks.

task of the algorithm to find this person in the gallery. It has to detect the person and also correctly identify them as the query person.

Table 6.1 shows the "person search" accuracy in terms of both mean average precision (mAP) and top-1 score. We evaluate on the CUHK-SYS test set with a gallery size of 50. As a baseline we also compare to colour histogram matching (based on an implementation of a particle tracker[5]) with YOLOv2 region only detections, which is often used in tracking as another low cost method. It is clear that our method performs a lot better. We can also see that adding extra layers in "split end" clearly has increased accuracy over "mixed end". Xiao et al. [161] still seems to be more accurate than our method. However, if we also compare them in terms of speed (figure 6.12) our method makes a great trade-off in terms of speed and accuracy. This makes it usable in resource constrained applications like embedded platforms and large scale video processing.

6.5.3 Computational complexity

We mentioned frequently that our network does not come with much extra cost compared to standard YOLO. Table 6.2 shows the computational complexity in terms of network parameters and floating-point operations (FLOPs) per

[5] https://bitbucket.org/kschluff/particle_tracker

image of both architectures compared to standard YOLOv2. The complexity of "mixed end" is lower than the standard YOLOv2 because classification output is removed. Adding extra layers comes at the cost of 12% extra computation and 19% extra parameters.

6.5.4 Person tracking

We evaluate our tracker using re-id appearance features against a standard one class YOLO tracker trained ourselves without the use of appearance features (tracking by detection, which only uses a Kalman tracker combined with Hungarian algorithm and IoU) using the MOT16 evaluation framework. The result is shown in table 6.3. Both trackers are run at the same working point on the PR curve. Overall we get great improvements in tracking accuracy using our method. We get significantly more mostly tracked (MT) and partly tracked (PT) trajectories suggesting that our re-id vector is successful in reconstructing tracks. Overall MOTA and IDF scores are also increased. More information about these measurements can be found in [97].

6.6 Conclusion

We presented a single shot neural network architecture based on YOLOv2 that is able to perform both detection and re-identification, while only introducing a negligible increase in complexity compared to the normal YOLOv2. Our experiments indicate that our architecture is able to yield a usable re-id vector, with minimal loss in accuracy ($\pm 0.5\%$ average precision), compared to other low cost approaches. We evaluated different architectures and found that adding extra layers in the more complex "split end" architecture benefits detection and re-id accuracy only a small amount. Adding a deeper front end, Resnet50 increases the accuracy significantly.

We demonstrated the effectiveness of our approach in the application of tracking. Using our approach tracking accuracy increased greatly compared to only using detection for tracking.

By changing the network backend i.e. exchanging darknet19 for Resnet50 we can *explore* the speed accuracy trade-off in many applications. Using a deeper head generally gives us better accuracy at the cost of more inference time.

	IDF1	IDP	IDR	Rcll	Prcn	FAR	GT	MT	PT	ML	FP	FN	IDs	FM	MOTA	MOTP	MOTAL
YOLOv2 Tracking-by-Detection	27.0	**59.1**	17.5	27.8	**93.6**	**0.40**	517	13	187	317	**2102**	79759	774	**1951**	25.2	**77.5**	25.9
YOLO + histogram	31.8	57.3	22.0	34.2	89.1	0.87	517	23	220	274	4630	72646	1135	2336	29.0	75.8	30.0
YOLO-REID (split end)	**33.9**	56.7	24.2	**36.1**	84.5	1.37	517	**26**	227	264	7306	**70580**	882	2327	28.7	75.5	29.5
YOLO-REID (mixed end)	33.2	**59.1**	23.1	35.2	89.8	0.83	517	**26**	212	279	4393	71569	**723**	2001	**30.5**	76.5	**31.2**

Table 6.3: Comparison of tracking performance against our own trained one class YOLO without re-id and with re-id enabled.

Chapter 7

Conclusion

The title of this thesis is "Real world applications of Artificial Intelligence on constrained hardware". Throughout this thesis we looked at many applications of AI from a very wide range of its subfields. We looked how they can be implemented on hardware that is constrained in some way. During this thesis we identified three categories of constraints that come back again and again in many of the different applications. We laid them out into a framework that identified which constraints were "hard", meaning that they cannot be changed, which constraints were changeable, and what the impact on other constraints is if we were to change them. We call the interaction between these changeable constraints "trade-offs".

While studying the many applications in this thesis, we found that generally speaking, there are two ways to play with trade-offs. The first option is to *explore* a trade-off. By changing one constraint (e.g.: increasing the amount of computing power), the other constraint becomes less strict (e.g.: the algorithm runs faster). Exploring a trade-off can be done without changing the algorithm. It's simply a parameter that can be set in the application. This can be a software parameter that influences for instance the accuracy, or a hardware parameter that represents the amount of computing power (e.g.: the number of available cores on a system). Setting these parameters correctly is often essential to delivering a usable application.

Apart from the exploration of trade-offs, a big part of this thesis was also about optimizing trade-offs. Compared to exploration, when we optimized a trade-off, we actually tried to increase the performance of one constraint, at no cost of another. Generally, this is not possible by only changing parameters in the application. We have to change some aspects of the system more cleverly,

so that we can utilize its current constraints more effectively. In many cases this comes down to making adaptions to the algorithm, or even exchanging it outright for one that fits better in the current constraints. This process of changing the algorithm is called *optimisation*.

In the introduction, we already touched on when we can do this, and how it can be done in the best way. Optimizing an algorithm to run on parallel hardware (GPUs) for instance, can be quite hard. Depending on the algorithm it might also be impossible to exploit even more parallelism in the algorithm because it simply isn't there. Options other than parallel hardware do however also exist. By for instance using application specific hardware (like FPGAs or ASICs), one might get even more performance out of an application that is limited by hardware. Compared to a GPU which uses a somewhat fixed processor layout that is today still mainly designed for graphics (although this is changing, by the ever expanding deep learning market, and the inception of neural network generated graphics [159]), using application specific hardware it is possible to design a platform that is perfectly tailored for an application (has only the necessary hardware), without any unnecessary hardware. This allows for huge performance improvements in terms of speed and power consumption [50].

Human effort, and human development time is also a factor that should not be overlooked. While it is often the case that, by studying a problem in-depth, one is able to optimise it a great deal, this also takes time, which can be a constraint on its own. To which level one should optimize an application of course greatly depends on the application itself. When the constraints of an application are quite loose, efficiency might not be the first concern. However, in the case that some constraints are tight, it might be worth it to spend some extra time on optimisation.

Figure 7.1 shows the decision process of choosing an appropriate algorithm for a given design constraint (i.e.: one of the corners of our triangle framework). We begin by taking a look at an algorithm that is likely to meet the specifications. If the constraint in the specification is met we can make a trade-off between the two other constraints. We pick whichever is the most beneficial for our application. If the algorithm does not already meet our specified constraint we can try to modify other constraints such that our constraint is met. If the algorithm does not allow for a trade-off (e.g.: the algorithm can't be made more accurate by making it run slower or using more hardware), we pick the algorithm that looks most suited now, and repeat the process. If no algorithm remains, the problem is not solvable with current technology. Extra fundamental research might be able to solve the problem, or we have to change the specification somehow (e.g.: asking for more data, using a server based architecture...). If we do find that we can meet the constraint by reducing another property, we go back to making a trade-off between that property and the remaining property. The selected

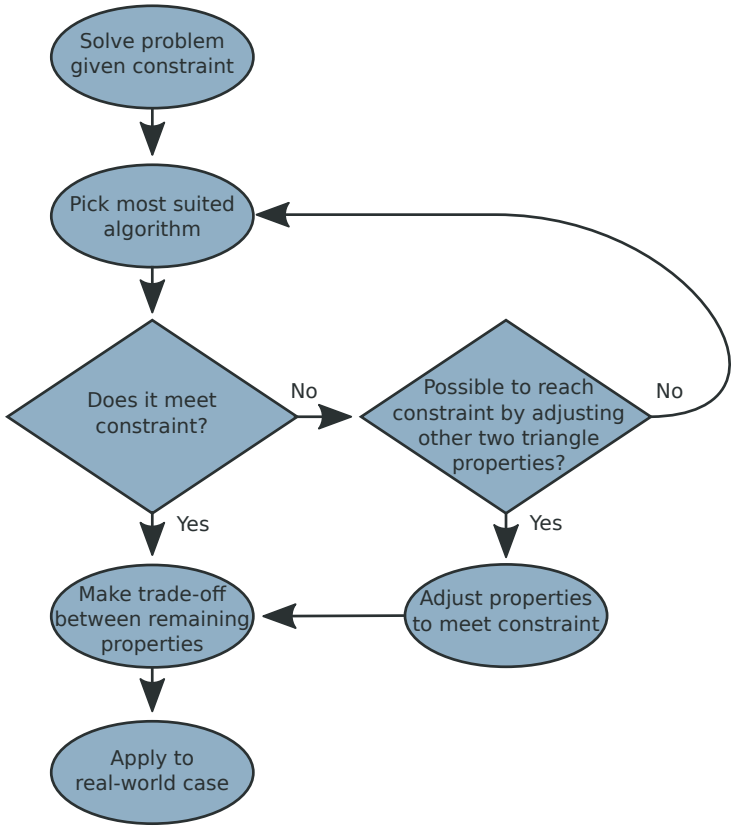


Figure 7.1: Decision process to choose an algorithm that meets constraints.

trade-off and constraint can then be used in our real-world application.

While in-depth knowledge about the application can definitely be a big asset in making it run optimally under some constraints, We do not think that it is always an absolute necessity. In-depth knowledge enables you to exploit certain properties about the application that cannot be done without simply changing some parameters. In applications where we tried to optimise the algorithm (which is all of them up to some level in this thesis) we found that domain knowledge is absolutely a must have asset in getting the most performance out of an algorithm. The case in which this became the most obvious during this PhD was the GPU implementation of ACF in chapter 5. Only by understanding completely how the algorithm worked were we able to make adjustments that resulted in (often minor) speed increases. However, this is true when we are

trying to optimise the algorithm ourselves. With the rise in interest in deep learning an AI, with big cooperation like Google, Facebook and NVIDIA backing big, often Open Source projects, doing the optimisation ourselves is often not necessary. The code released in existing frameworks is often already pre tuned to run very efficiently. The use of deep learning frameworks mean that a lot of the popular primitives to do computation are already available under the hood, by tuning the parameters or doing things differently we are often able to get increases in speed. Lastly, hardware manufacturers like NVIDIA provide tools (TensorRT in the case of NVIDIA) that are able to automatically tune the code of a deep learning pipeline for their hardware, without having to have to much in depth knowledge (as long as the operations are supported inside the framework).

While optimising code can certainly be important, for small, lower budget, the importance of picking the right algorithm from literature, and tuning its parameters to fit your needs can often be as much if not more important then spending the effort in optimising the application.

We hope that the conclusions from the applications in this thesis can, serve as a reference for developing new applications, and that by looking at the problems we encountered, the reader finds inspiration when developing other related applications.

In the remainder of this concluding chapter we will look back from each pillar of our triangle framework, that was used throughout this thesis, towards the applications. Section 7.1 is about the runtime speed computing power trade-off, section 7.2 about the compute accuracy trade-off, and section 7.3 about the speed accuracy trade-off.

7.1 Runtime compute trade-off

Throughout this thesis we saw many instances where the accuracy constraint was hard, and we had to make a trade-off between runtime speed and computing-power (figure 7.2). In the introduction 1.2.1 we talked about the implication of such a trade-off, an already identified some practical trade-offs where such a trade-off is applicable. In this thesis we came across many applications with this type of trade-off.

In section 5.5, we studied the best way to identify solar panels from areal photography. As this was an exploratory study about the best way to detect solar panels, we were not given an accuracy target in advance. We did however desire a system with an accuracy that was as high as possible. Because of that,

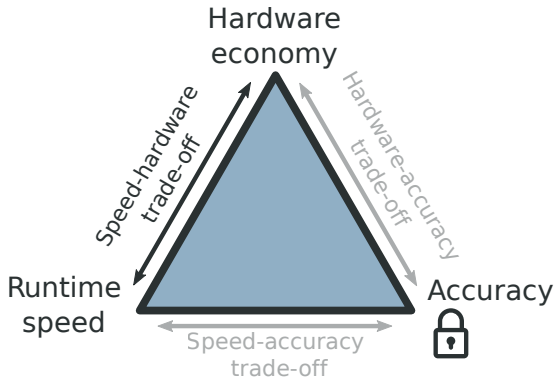


Figure 7.2: Visualization of the runtime-compute trade-off, we cannot change anything about the accuracy.

the solution we ended up choosing was also the one with the highest accuracy, which happened to also perform quite well in terms of hardware needs and runtime speed.

Section 5.6 was about automatically steering a PTZ camera to film people in their living room. We made a system that was able to autonomously steer a PTZ camera to make static shots of people. The production house that came to us with this problem valued the quality of the shot highly. Being a static shot the speed at which the algorithm ran was not as important. Most filming locations also had room for enough server hardware, also making it of lesser importance.

In chapter 6 about person search and tracking, we identified two possible use cases that had a runtime vs. compute trade-off:

1. In a system that does online person search, where we ring an alarm every time a target person is seen in one of the video cameras, it might become annoying (which can lead to people ignoring alarms) to get too many false alarms. To solve this we might want to reach a certain target accuracy.

2. When tracking people in a store to analyse their shopping behaviour making fewer mistakes is far more important compared to speed or hardware needed. This type of computation is typically done offline so slower algorithms are generally tolerable, and apart from cost there are no real implications on the hardware that is used.

Overall we can identify two types of applications that have this kind of trade-off: applications where offline processing is done, and applications where accuracy is highly important. What we learned from both of these applications is the following:

In the former, offline processing, we are usually not limited in time and often also not in speed, leaving accuracy not always as a constraint but certainly something that is nice to optimize for. In the cases we did that had offline processing we found that usually the most accurate algorithm was the one that fit best. If the amount of speed or hardware is not a problem, it might not be necessary to take the effort to optimize even further. However, in many cases, in real-world systems they can become important. Sometimes the waiting times become too long (hours) or a company likes to get more performance out of their investment (a GPU cluster might for instance be cheaper than the same amount of processing power on CPU). This then comes at the cost of either development time or extra hardware. This is a trade-off companies have to make for each application.

Often applications also just require the best accuracy available. The field might still be very immature, and accurate fast and low power applications might not exist. Controlling the PTZ camera was such an application. When we developed it, person detection was still very hard (compared to today) and only the best detectors yielded good results. We had some leeway in speed and hardware, so we used it to get the highest quality results. In other applications this is often also the case, when there is some leeway in speed and/or accuracy it is often good to use it to get the best quality result.

7.2 Hardware accuracy trade-off

In the hardware accuracy trade-off (shown in figure 7.3) we made a trade-off between the amount of computing power a system has (eg. from the use of expensive GPUs, multiple systems to embedded hardware) and the accuracy at which the results of the algorithm are computed. Runtime speed in these systems was a hard constraint.

In the introduction we also talked about the distinction between hard, firm and soft real-time systems in section 1.2.2. To recall we used the following definitions:

Hard real-time For mission critical systems, if the task is not carried out by a certain deadline the complete task is ruined, which can even have safety

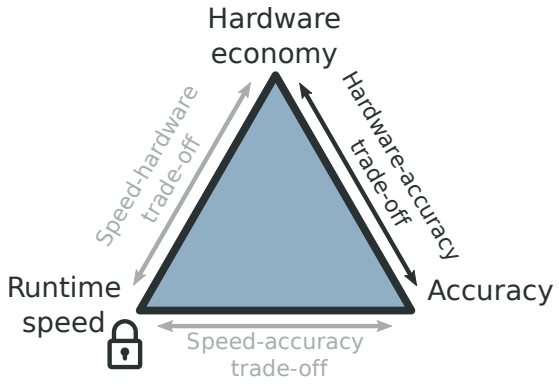


Figure 7.3: Visualization of the runtime-compute trade-off, we cannot change anything about the speed.

critical consequences. E.g.: ABS systems in a car, controlling the position of a 3D printer.

Firm real-time The system will not operate like it should if deadlines are not met, but the system can continue operating. As long as not too many deadlines are missed it's only an inconvenience for the user. E.g.: Decoding a video to display in real-time.

Soft real-time We are okay as long as the system returns results within a reasonable amount of time. E.g.: Taking a picture from a remote security camera.

In this thesis the following applications fitted in the hardware accuracy trade-off in our triangle framework:

While the applications of our work in chapter 3 titled "GPU implementation of a forward sampling algorithm in CP-logic" are numerous, we gave the example of an automated director to apply it to the triangle framework. The idea here was that our decision system automatically determines which camera to show the viewer in a multi camera setup. During live video broadcasting making fast decisions becomes crucial. This is an example of a *firm* real-time application, not making the deadline is okay, as long as it doesn't happen too often. In the work we did here, we managed to speed up inference of CP-logic theories substantially.

Chapter 4 titled "Ultra-low-latency automatic endoscopic image orientation stabilisation", is perhaps one of the best examples of a speed constrained system.

One of the key specifications of the system was that it had to work in real-time, with a sub frame latency < 16ms. Taking this constraint into account, we tried a few different implementations using different hardware setups. By changing the amount of compute power in the hardware we could clearly see a trade-off in tracking drift (accuracy). This is an example of a *hard* real-time application. It is crucial for the success of surgery that video is streamed at low latency.

Chapter 6 again has some possible applications with a speed constraint:

1. Following people from a drone. When the visual input of a camera on a drone is used to steer the drone, minimizing latency is important. Depending on the application, this can be *hard* real-time, e.g. missing a person may result in the drone possibly crashing.

2. Detecting suicidal behaviour on a railway platform based on a tracking trajectory. Being able to sound the alarm early in order to prevent a possible death is crucial here. As this application just has to create an alarm in a reasonable amount of time, this application is *soft* real-time.

3. Offline person search. When we have huge amounts of data, speed becomes more important in order to get results back in a reasonable time. Again, *soft* real-time as the response time in not crucial to the successful operation of the system.

4. Online person search. For instance during a bank robbery it is important to know the location of a robber fast to make sure that authorities can react quickly. Responding in a reasonable amount of time is sufficient here, so this is *soft* real-time.

In summary, we took a look at many types of applications that have a hardware constraint. Looking back we found that the following holds in all of them: For hard real-time systems it is essential that deadlines are met. In the case in chapter 4 we did this by maintaining as much as possible of the existing pipeline so that video was still broadcast at ultra-low-latency. We decoupled our algorithm from the pipeline. In the other not hard real-time applications we generally choose an algorithm that was fast to start with, but might not be the most accurate one. We found that in some cases optimizing was the best way to get more accuracy on the same hardware and in others adding more hardware was the best solution to reaching desired accuracy. There's no general rule here that can be done for each case. We recommend weighing the benefits of each solution keeping in mind the particularities of the application.

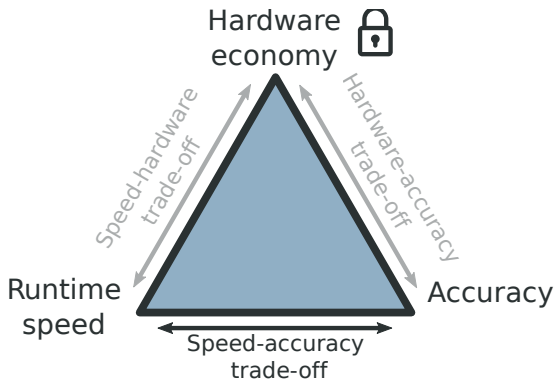


Figure 7.4: Visualization of the runtime hardware trade-off, we cannot change anything about the hardware economy.

7.3 Speed accuracy trade-off

We had to make a trade-off between speed and accuracy (shown in figure 7.4) in applications where the hardware platform was fixed. Depending on the application we moved more towards speed or accuracy.

In this thesis we tackled the following applications:

In section 5.10 we explored the possibilities of doing person detection on board of UAV. As UAVs are limited in payload and power consumption, the hardware platform is also limited.

One of the possible applications in chapter 6 about person search is offline person retrieval. In many cases the hardware environment where such an application has to run is already pre-existing, and we want to make a system that runs on top of that hardware. In the work we did, we set the speed accuracy trade-off more towards speed. We created an algorithm that can do person search really fast.

While in both of these applications, hardware is fixed, the reasons that it's fixed and the type of applications still vary widely. In the drone applications hardware is fixed due to physical properties of the drone, and in the person search application simply because it's not desirable to add more hardware. By playing with the network backend and the input image resolution of the network we can play with the trade-off between speed and accuracy. In both off these applications we managed to find a better trade-off by optimizing and changing the algorithm that was used.

What we learned when working on this type of application is that a good first step is to start with an algorithm that is already quite fast. If the results in speed or accuracy are still not sufficient, optimizing the algorithm further for the platform is a good idea. In some cases, for instance when using GPUs, this optimisation can be done across different platforms. However, if we want to get the most performance out of the application, we do need to make platform specific optimisations (e.g.: using intrinsic functions, modifying the application to use the available low level memory). Before undertaking this effort it is also a good idea to estimate how well you expect the application to run on the target platform to determine if it is worth the optimisation cost.

7.4 Future work

AI is a very active domain. In this thesis we studied some applications of this domain. As the field is so large we acknowledge that the topics studied in this thesis represent only a fraction of the domain. We anticipate that AI as a very broad term that encompasses many subfields will only continue to keep growing and find uses in many different and new applications. Outlining what the future holds for all of these applications is outside the scope of this thesis. In this future work section we will limit ourselves to describing future work for the applications that were covered in the different chapters.

7.4.1 Using the GPU for Probabilistic Logic Programming

In this work we provided the first forward sampling algorithm for PLL theories using GPU accelerators.

As was already mentioned in the conclusion of chapter 3: the fact that our method is not query-based opens up some interesting possibilities. Our method could be applied to instances where query-based methods have problems with an infinite number of explanations. Currently our implementation's ground-and-solve method cannot cope with infinite grounding, but there does not appear to be a conceptual difficulty in switching to a lazy grounding method.

The second future work mentioned in chapter 3 is that our implementation might also be easier to extend to new language features, because it more closely follows the execution model semantics of CP-logic. This may in particular be the case for language features that extend the expressive power of the heads of CP-logic rules. For instance, while the head of a CP-logic rule is currently a choice between a fixed number of alternatives, we could allow the set of possible

alternatives and/or their probabilities to be dynamically determined, based on the interpretation $I(s)$ in the state s where it is executed. A similar feature can be found in the P-log language [14], which has some similarities to CP-logic, but does not follow Sato's distribution semantics. (A comparison between CP-logic and P-log can be found in [154].) A second example is that, instead of allowing only a discrete choice in the head of rules, we could also allow a value to be selected from a continuous distribution. Finally, recent work [153] has investigated the possibility of extending CP-logic with negated atoms in the head. Integrating such features into our forward sampling algorithm seems much easier than extending one of the proof-based methods.

Our method allows for very fast sampling, and can be useful in many applications. In this thesis we already talked about the automatic video director. But others certainly also exist. Take for instance the firefighter surveillance application using drones. What we do right now is simply sound an alarm when we see a dangerous object or someone enters a dangerous area. Using PLL, this system can be made way more complex, to for instance calculate the true risk a firefighter is taking.

Apart from that, we also think that further optimisation is certainly possible. Since the time that this work was published, much has changed in the world of GPU programming. New GPUs with more memory have become available and GPUs have also become available on embedded platforms. During our tests there were some instances where we ran into memory limitations. Newer GPUs will certainly be able to cope with bigger memory requirements. Apart from that we also think it is possible to prevent memory size issues by using more efficient ways of handling memory (for instance by not putting all weights into GPU memory).

7.4.2 Image processing at ultra low latency

In this application we were quite limited by the hardware we had to work with, and the latency constraints we were given. Without changing this it is quite hard to use more computationally intensive algorithms. By introducing a compute server however it was possible to use a more accurate algorithm. Keeping track of previous frames and using descriptive keypoints like SIFT, SURF or Oriented FAST and Rotated BRIEF (ORB) should be able to reduce long term drift even more. Other applications on this hardware architecture are also possible. For instance, using SLAM techniques it might be possible to create a map of the region where surgery is performed to aid the orientation of the surgeon even more.

7.4.3 Object detection

Object detection has come a long way since we started this PhD. Back then the best approaches used hand-crafted features together with some kind of classifier evaluating each window using a sliding window approach. With the rise of deep learning, object detection has become more and more reliable. Things that seemed impossible just a few years ago are now easy. Much of the datasets used back then are now essentially solved. While object detection is still a very active area of research, a lot of the research you see today is shifting towards more challenging domains. Segmentation were a mask of a person is detected is one of them, reliably tracking is another, re-identification yet another, visual question answering yet another and so on. Computer vision and object detection is a very active area of research, with a lot of subfields.

In terms of optimizing these techniques efficiently for embedded platforms many advances have also been made since this research was carried out. Embedded GPUs are becoming even more popular, with NVIDIA coming out with more and more powerful embedded GPU platforms, and purpose built hardware like Intel's Movidius stick becoming more and more popular. A lot of these platforms are now developed with deep learning for the purpose of object detection in mind from the start. Specialized frameworks like NVIDIA's TensorRT allows you to get the most out of them. Doing more research on how to efficiently deploy state-of-the-art detectors to these platforms remains an interesting area of research. The neural network itself can be made more efficient too. Things like network pruning and bottleneck layers can make neural networks faster by reducing the amount of computation that needs to be carried out, and thus also reducing the amount of memory needed. Knowledge transfer, letting a teacher network transfer its knowledge to a student network, allow a more compact network to learn from a big network, without having to retrain on all training data. More and more hardware is also starting to support lower precision floating-point (and 8-bit integer) operations. By reducing the size of number representation the hardware is essentially able to carry out more operations at once. Making it twice or four times as fast as a single 32-bit floating-point operation.

7.4.4 Re-identification

In our work in chapter 6 we laid out a method to do re-identification and detection in one single shot network. In this work, our approach was mainly focused on having a fast pipeline that does both detection and re-identification. The re-identification part essentially came for free together with the detector. Because of the focus on speed we chose the YOLO detector as a baseline to

create our approach. The YOLO detector and its darknet19 backend are however not the architectures that deliver the best results in terms of accuracy. At the end of the chapter we also experimented with deeper network architectures like ResNet50 and darknet53. While the results of this seem promising, we believe that even more research in this area can lead to even more accurate results.

Our novel approach in re-identification also leads the way to other promising re-identification schemes. One of the applications we are currently working on in our research group is the re-identification of children in pictures taken during the school-day to make photo albums. One of the challenges we encountered here was that children tend to move around a lot, and appear in different poses inside of pictures. Traditional re-identifcation pipelines however assume that persons are upright in the image (the way they typically also occur in a dataset). This poses a problem in working with these kinds of images that a variation of our architecture might be able to solve. Instead of identifying a cut out patch, our approach asks the question: Who is the person under this point (anchor) in the image. It is assumed then that the network has some idea of where the person is underneath the point and what the pose of this person is. Indeed, compared to the traditional approach our solution should be able to handle irregularly shaped objects (children in this case) without many problems.

Another approach which might lead to an increased accuracy is eliminating the bounding box detection altogether. In some applications, knowing the exact location of people in the image is not strictly necessary. For instance in a person retrieval application in which information about past frames is stored in the database we don't really need the exact location of people. Instead, we can work only with the representation of people in the form of a re-identifcation vector. This vector can be output by our network with possibly a "personness-score" which expresses how likely it is that the anchor contains a person, without a need for bounding boxes. In our database, we can then save all vectors corresponding to a person in that image. Note that while we do not have an exact bounding box in this case, the position of a person can still be determined by looking at the location of the matching vector inside the resulting output grid.

Conversely to this, recent research has also determined that multi-task training can lead to a higher accuracy for all tasks compared to training for only one task. The recent Mask R-CNN detector is an example of this. Instead of just working as a detector, it also outputs a detection mask [56]. In this future we might see more and more networks that combine multiple tasks into a single backend network.

7.5 Vision on the future of artificial intelligence in real-world applications

Apart from thinking about the future from the point of view of the applications, we can also think about what is to come more generally and in the more distant future. Which applications will become possible with technology that is now still in it's infancy.

One interesting area is the fusion of two separate topics of this thesis. Up to this point, the applications we're working on the computer vision side of the research group is still mainly concerned about interpreting what a camera sees. An object detector is now able to highlight very accurately where people are positioned in a frame, or a re-identification network is able to accurately recognise a person it has seen before. However going further than that, *reasoning* about those things, making the computer understand what it is seeing is still very much in it's infancy. Human thought is structured around understanding physical objects that are present in the environment and reasoning about them given rules it has learned about the world. Whether it be answering questions about the world [11] (e.g.: which person is standing left to the tree in this picture), the physical interaction between objects [52, 163] (i.e.: an object in the sky with nothing underneath it will probably fall down) or even human behavior [13] (e.g.: a person approaching a zebra crossing is likely to cross the street). An example of this is presented in [78], where the authors created a character recognition engine that is able to outperform traditional methods by representing them as simple programs that best explain examples under a Bayesian criterion. Motivated by how well a human is able to apply concepts to examples (such as how a character is drawn versus just looking at pixel level) often given only one example, they create a learning algorithm that is essentially able to do the same. Instead of looking at the character itself the system uses a representation of how a character was drawn.

Another problem we often encounter in the research we focus on in our group is a lack of training data. Due to the specificity in some of the cases that we solve training data often comes at a premium. The field of self-supervised learning seems to have interesting solution to this. Instead of training on ground truth of one type of data, multi modal data is used to train a model. The training data is split in different modi, and it is the goal of the network to predict one from the other. E.g.: trying to predict the sound of a video using only visual data. Recent work has proven that this approach can reach promising results [157], for video colourisation and tracking. Trying this on real-world applications would certainly be interesting.

Another struggle with current techniques in deep learning is that of architecture

and parameter tuning. Part of why big players like Google and Facebook are able to dominate the field is because they have a lot of resources and are able to try many different architectures. The field of AutoML, searching for the best architectures and parameters to get the best result promises to do this automatically [10]. Similar to what deep learning did to hand crafted feature detectors. AutoML can automatically learn the best architecture without having to do any engineering on the architecture side.

While this thesis is largely about the technical side of AI another discussion that is becoming hot at the moment is that of the societal impact of deep learning. Self learning systems are going to find more and more uses in the world. As the field progresses we will have to question the ethical side of the systems we make more and more. Who is responsible for the actions a robot takes in the real-world? As is talked about in [15], by using training data that already has biases, are we building in biases in our models? Re-identification systems have a lot of issues about privacy, and the use of them in public surveillance have a lot of issues of privacy [23]. In the field of deep learning explainability is also a big topic. Deep neural networks are, compared to the techniques we used in chapter 3, mostly seen as a black box. It is often hard, even for the designers to explain what is going on. While this is on a technical level not a reason that prevents us to use deep networks in real-world environments, from a social perspective this is a very important issue that is currently goes mostly unaddressed. Deep neural networks are used more and more to make safety critical decisions, think of most medical applications, self driving cars, autonomous drones, security systems... In these applications we (as a society) often demand to know why the algorithm took a certain decision (especially if they go wrong). Making current techniques explainable or searching new techniques that are explainable will certainly be relevant in the future.

Bibliography

[1] How much power is needed to hover ? https://web.archive.org/web/20180808002907/http://www.starlino.com/power2thrust.html, 2015. Accessed: 27 February 2019.

[2] Embedded systems development kits, modules & sdks. https://web.archive.org/web/20180918090448/https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/, 2018. Accessed: 01 October 2018.

[3] i.mx6 arm development board. https://web.archive.org/web/20180920042924/https://boundarydevices.com/product/sabre-lite-imx6-sbc/, 2018. Accessed: 01 October 2018.

[4] Jetson tk1 embedded development kit. https://web.archive.org/web/20180901000104/https://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html, 2018. Accessed: 01 October 2018.

[5] Nvidia jetson agx xavier developer kit. https://web.archive.org/web/20180908153253/https://developer.nvidia.com/embedded/buy/jetson-xavier-devkit, 2018. Accessed: 01 October 2018.

[6] Vivante gc2000. https://web.archive.org/web/20181130121846/http://www.vivantecorp.com/index.php/en/technology/3d.html, 2018. Accessed: 19 February 2019.

[7] Abbas-Turki, L. A., Vialle, S., Lapeyre, B., and Mercier, P. High dimensional pricing of exotic european contracts on a gpu cluster, and comparison to a cpu cluster. In *Parallel & Distributed Processing* (2009), IEEE, pp. 1–8.

[8] Aerts, B., Goedeme, T., and Vennekens, J. A probabilistic logic programming approach to automatic video montage. vol. 285, Kaminka, GA, IOS Press; Amsterdam, pp. 234–242.

[9] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (1967), ACM, pp. 483–485.

[10] Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and De Freitas, N. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems* (2016), pp. 3981–3989.

[11] Antol, S., Agrawal, A., Lu, J., Mitchell, M., Batra, D., Lawrence Zitnick, C., and Parikh, D. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision* (2015), pp. 2425–2433.

[12] Bae, S.-H., and Yoon, K.-J. Robust online multi-object tracking based on tracklet confidence and online discriminative appearance learning. In *CVPR* (2014), pp. 1218–1225.

[13] Baker, C. L., Jara-Ettinger, J., Saxe, R., and Tenenbaum, J. B. Rational quantitative attribution of beliefs, desires and percepts in human mentalizing. *Nature Human Behaviour 1*, 4 (2017), 0064.

[14] Baral, C., Gelfond, M., and Rushton, N. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming 9*, 1 (2009), 57–144.

[15] Barocas, S., and Selbst, A. D. Big data's disparate impact. *Calif. L. Rev. 104* (2016), 671.

[16] Bay, H., Tuytelaars, T., and Van Gool, L. Surf: Speeded up robust features. In *European conference on computer vision* (2006), Springer, pp. 404–417.

[17] Bell, N., and Hoberock, J. Thrust: A productivity-oriented library for cuda. *GPU computing gems Jade edition 2* (2011), 359–371.

[18] Benfold, B., and Reid, I. Stable multi-target tracking in real-time surveillance video. In *CVPR* (June 2011), pp. 3457–3464.

[19] Bewley, A., Ge, Z., Ott, L., Ramos, F., and Upcroft, B. Simple online and realtime tracking. In *2016 IEEE International Conference on Image Processing (ICIP)* (2016), pp. 3464–3468.

[20] Bochinski, E., Eiselein, V., and Sikora, T. High-speed tracking-by-detection without using image information. In *Advanced Video and Signal Based Surveillance (AVSS)* (2017), IEEE, pp. 1–6.

[21] Bouguet, J.-Y. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel Corporation 5*, 1-10 (2001), 4.

[22] Breedveld, P. Observation, manipulation, and eye-hand coordination problems in minimally invasive surgery. In *European Conference on Human Decision Making and Manual Control (ed): Proceedings of the 16th European Conference on Human Decision Making and Manual Control, Kassel* (1997), pp. 219–231.

[23] Brey, P. Ethical aspects of facial recognition systems in public places. *Journal of information, communication and ethics in society 2*, 2 (2004), 97–109.

[24] Candiago, S., Remondino, F., De Giglio, M., Dubbini, M., and Gattelli, M. Evaluating multispectral images and vegetation indices for precision farming applications from uav images. *Remote Sensing 7*, 4 (2015), 4026–4047.

[25] Chen, J., and Carr, P. Autonomous camera systems: A survey. In *AAAI-14 Workshop on Intelligent Cinematography and Editing* (2014).

[26] Chen, Z., Badrinarayanan, V., Lee, C.-Y., and Rabinovich, A. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. *arXiv preprint arXiv:1711.02257* (2017).

[27] Dai, J., Li, Y., He, K., and Sun, J. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems* (2016), pp. 379–387.

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

[29] Daniyal, F., and Cavallaro, A. Multi-camera scheduling for video production. In *Visual Media Production (CVMP), 2011 Conference for* (2011), IEEE, pp. 11–20.

[30] De Raedt, L., Kimmig, A., and Toivonen, H. Problog: A probabilistic prolog and its application in link discovery. *International Joint Conference on Artificial Intelligence (IJCAI)* (2007).

[31] De Smedt, F., Hulens, D., and Goedemé, T. On-board real-time tracking of pedestrians on a uav. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* (2015), pp. 1–8.

[32] Dollár, P., Appel, R., Belongie, S., and Perona, P. Fast feature pyramids for object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence 36*, 8 (2014), 1532–1545.

[33] Dollár, P., Appel, R., et al. Fast feature pyramids for object detection. *TPAMI 36*, 8 (2014), 1532–1545.

[34] Dollár, P., Tu, Z., et al. Integral channel features. In *BMVC* (2009).

[35] Dollár, P., Tu, Z., Perona, P., and Belongie, S. Integral channel features.

[36] Dosovitskiy, A., Springenberg, J. T., Riedmiller, M., and Brox, T. Discriminative unsupervised feature learning with convolutional neural networks. In *Advances in Neural Information Processing Systems* (2014), pp. 766–774.

[37] Doubek, P., Geys, I., Svoboda, T., and Van Gool, L. Cinematographic rules applied to a camera network. In *Omnivis 2004* (2004), Prague, pp. 17–29.

[38] Dubout, C., and Fleuret, F. Exact acceleration of linear object detectors. In *European Conference on Computer Vision* (2012), Springer, pp. 301–311.

[39] Felzenszwalb, P., McAllester, D., and Ramanan, D. A discriminatively trained, multiscale, deformable part model. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on* (2008), IEEE, pp. 1–8.

[40] Felzenszwalb, P., McAllester, D., and Ramanan, D. A discriminatively trained, multiscale, deformable part model. In *CVPR* (2008), IEEE, pp. 1–8.

[41] Felzenszwalb, P. F., Girshick, R. B., et al. Cascade object detection with deformable part models. In *CVPR* (2010), pp. 2241–2248.

[42] Felzenszwalb, P. F., Girshick, R. B., and Mc-Allester, D. Cascade object detection with deformable part models. In *CVPR* (2010), IEEE, pp. 2241–2248.

[43] Fierens, D., Broeck, G. V. d., Thon, I., Gutmann, B., and De Raedt, L. Inference in probabilistic logic programs using weighted cnf's. *UAI: Uncertainty in Artificial Intelligence* (2011).

[44] Fischler, M. A., and Bolles, R. C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM 24*, 6 (1981), 381–395.

[45] Freund, Y., Schapire, R., et al. A short introduction to boosting. *JSAI Journal 14*, 771-780 (1999), 1612.

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

[47] Friedman, J., Hastie, T., Tibshirani, R., et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics 28*, 2 (2000), 337–407.

[48] Glaser, A. Dji is running away with the drone market. https://www.recode.net/2017/4/14/14690576/drone-market-share-growth-charts-dji-forecast, 2017. [Online; accessed 14-04-2017].

[49] Gleason, J., Nefian, A., et al. Vehicle detection from aerial imagery. In *ICRA* (2011), IEEE, pp. 2065–2070.

[50] Guo, K., Zeng, S., Yu, J., Wang, Y., and Yang, H. A survey of fpga based neural network accelerator. *arXiv preprint arXiv:1712.08934* (2017).

[51] Hadsell, R., Chopra, S., and LeCun, Y. Dimensionality reduction by learning an invariant mapping. In *CVPR* (2006), vol. 2, IEEE, pp. 1735–1742.

[52] Hamrick, J. B., Battaglia, P. W., Griffiths, T. L., and Tenenbaum, J. B. Inferring mass in complex scenes by mental simulation. *Cognition 157* (2016), 61–76.

[53] Han, X., Leung, T., Jia, Y., Sukthankar, R., and Berg, A. C. Matchnet: Unifying feature and metric learning for patch-based matching. In *CVPR* (2015), pp. 3279–3286.

[54] Harris, C., and Stephens, M. A combined corner and edge detector. In *Alvey vision conference* (1988), vol. 15, Citeseer, pp. 10–5244.

[55] Harwood, B., Carneiro, G., Reid, I., Drummond, T., et al. Smart mining for deep metric learning. *arXiv preprint arXiv:1704.01285* (2017).

[56] He, K., Gkioxari, G., Dollár, P., and Girshick, R. Mask r-cnn. In *Computer Vision (ICCV), 2017 IEEE International Conference on* (2017), IEEE, pp. 2980–2988.

[57] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[58] Hermans, A., Beyer, L., and Leibe, B. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737* (2017).

[59] Hermans, A., Beyer, L., and Leibe, B. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737* (2017).

[60] Hinz, S., and Baumgartner, A. Automatic extraction of urban road networks from multi-view aerial imagery. *Journal of Photogrammetry and Remote Sensing 58*, 1 (2003), 83–98.

[61] Höller, K., Penne, J., Schneider, A., Jahn, J., Boronat, J. G., Wittenberg, T., Feussner, H., and Hornegger, J. Endoscopic orientation correction. In *International Conference on Medical Image Computing and Computer-Assisted Intervention* (2009), Springer, pp. 459–466.

[62] Hong, I., Kuby, M., and Murray, A. A deviation flow refueling location model for continuous space: A commercial drone delivery system for urban areas. In *Advances in Geocomputation*. Springer, 2017, pp. 125–132.

[63] Huang, C., Wu, B., and Nevatia, R. Robust object tracking by hierarchical association of detection responses. In *ECCV* (2008), Springer, pp. 788–801.

[64] Hulens, D. Towards an autonomous flying cameraman.

[65] Hulens, D., Goedemé, T., and Rumes, T. Autonomous lecture recording with a ptz camera while complying with cinematographic rules. In *CRV* (2014).

[66] Hulens, D., Van Beeck, K., and Goedemé, T. Fast and accurate face orientation measurement in low resolution images on embedded hardware. In *VISIGRAPP* (2016), vol. 4, pp. 538–544.

[67] Hulens, D., Verbeke, J., and Goedemé, T. Choosing the best embedded processing platform for on-board uav image processing. In *VISIGRAPP* (2015), Springer, pp. 455–472.

[68] Izadinia, H., Saleemi, I., Li, W., and Shah, M. 2t: Multiple people multiple parts tracker. In *ECCV* (2012), Springer, pp. 100–114.

[69] Jones, S. Introduction to dynamic parallelism. In *GPU Technology Conference Presentation S* (2012), vol. 338, p. 2012.

[70] Karp, R. M., and Luby, M. Monte-carlo algorithms for the planar multiterminal network reliability problem. *Journal of Complexity 1*, 1 (1985), 45–64.

[71] Karp, R. M., Luby, M., and Madras, N. Monte-carlo approximation algorithms for enumeration problems. *Journal of algorithms 10*, 3 (1989), 429–448.

[72] Kendall, A., Gal, Y., and Cipolla, R. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. *arXiv preprint arXiv:1705.07115 3* (2017).

[73] Kimmig, A., Demoen, B., De Raedt, L., Costa, V. S., and Rocha, R. On the implementation of the probabilistic logic programming language problog. *Theory and Practice of Logic Programming 11*, 2-3 (2011), 235–262.

[74] Kumar, B., Carneiro, G., Reid, I., et al. Learning local image descriptors with deep siamese and triplet convolutional networks by minimising global loss functions. In *CVPR* (2016), pp. 5385–5394.

[75] Kumar, V. B., Harwood, B., Carneiro, G., Reid, I., and Drummond, T. Smart mining for deep metric learning. *arXiv preprint arXiv:1704.01285* (2017).

[76] Kumcu, A., Elprama, S., Vermeulen, L., Duysburgh, P., Platisa, L., Van Nieuwenhove, Y., Van De Winkel, N., Jacobs, A., Van Looy, J., and Philips, W. Effect of video latency on performance and subjective experience in laparoscopic surgery. In *15th Medical Image Perception Society conference (MIPS XV-2013)* (2013), Medical Image Perception Society (MIPS), pp. 59–59.

[77] Labati, R. D., Genovese, A., Muñoz, E., Piuri, V., Scotti, F., and Sforza, G. Biometric recognition in automated border control: a survey. *ACM Computing Surveys (CSUR) 49*, 2 (2016), 24.

[78] Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science 350*, 6266 (2015), 1332–1338.

[79] Lange, S., Sunderhauf, N., and Protzel, P. A vision based onboard approach for landing and position control of an autonomous multirotor uav in gps-denied environments. In *Advanced Robotics, 2009. ICAR 2009. International Conference on* (2009), IEEE, pp. 1–6.

[80] Lee, A., Yau, C., Giles, M. B., Doucet, A., and Holmes, C. C. On the utility of graphics cards to perform massively parallel simulation of advanced monte carlo methods. *Journal of computational and graphical statistics 19*, 4 (2010), 769–789.

[81] Li, W.-C., and Tsai, D.-M. Wavelet-based defect detection in solar wafer images with inhomogeneous texture. *Pattern Recognition 45*, 2 (2012), 742–756.

[82] Liao, L., Fox, D., and Kautz, H. Blocation-based activity recognition using relational markov networks,[in proc. 19th int. In *JCAI, Edinburgh, Scotland* (2005), pp. 773–778.

[83] Liao, S., Zhu, X., et al. Learning multi-scale block local binary patterns for face recognition. In *Advances in Biometrics*. Springer, 2007, pp. 828–837.

[84] Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., and Belongie, S. Feature pyramid networks for object detection. In *CVPR* (2017), vol. 1, p. 4.

[85] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *European conference on computer vision* (2014), Springer, pp. 740–755.

[86] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. Ssd: Single shot multibox detector. In *European conference on computer vision* (2016), Springer, pp. 21–37.

[87] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. SSD: Single shot multibox detector. In *ECCV* (2016), Springer, pp. 21–37.

[88] Lowe, D. G. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on* (1999), vol. 2, Ieee, pp. 1150–1157.

[89] Lucas, B. D., Kanade, T., et al. An iterative image registration technique with an application to stereo vision.

[90] Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., and Buck, I. Gpgpu: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (2006), ACM, p. 208.

[91] Luque-Vega, L. F., Castillo-Toledo, B., Loukianov, A., and Gonzalez-Jimenez, L. E. Power line inspection via an unmanned aerial system based on the quadrotor helicopter. In *MELECON, 2014 17th IEEE* (2014), IEEE, pp. 393–397.

[92] Matas, J., Chum, O., et al. Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing 22*, 10 (2004), 761–767.

[93] Mayer, H. Automatic object extraction from aerial imagery—a survey focusing on buildings. *CVIU 74*, 2 (1999), 138–149.

[94] Meert, W., Struyf, J., and Blockeel, H. Learning ground cp-logic theories by leveraging bayesian network learning techniques. *Fundamenta Informaticae 89*, 1 (2008), 131–160.

[95] Meert, W., Struyf, J., and Blockeel, H. Cp-logic theory inference with contextual variable elimination and comparison to bdd based inference methods. In *International Conference on Inductive Logic Programming* (2009), Springer, pp. 96–109.

[96] Meredith, J. S., Alvarez, G., Maier, T. A., Schulthess, T. C., and Vetter, J. S. Accuracy and performance of graphics processors: A quantum monte carlo application case study. *Parallel Computing 35*, 3 (2009), 151–163.

[97] Milan, A., Leal-Taixé, L., Reid, I., Roth, S., and Schindler, K. Mot16: A benchmark for multi-object tracking. *arXiv preprint arXiv:1603.00831* (2016).

[98] Milan, A., Roth, S., and Schindler, K. Continuous energy minimization for multitarget tracking. *IEEE transactions on pattern analysis and machine intelligence 36*, 1 (2014), 58–72.

[99] Moll, M., Koninckx, T., Van Gool, L., and Koninckx, P. Unrotating images in laparoscopy with an application for 30 laparoscopes. In *4th European conference of the international federation for medical and biological engineering* (2009), Springer, pp. 966–969.

[100] Moore, G. E., et al. Cramming more components onto integrated circuits, 1965.

[101] Mueller, M., Smith, N., and Ghanem, B. A benchmark and simulator for uav tracking. In *European Conference on Computer Vision* (2016), Springer, pp. 445–461.

[102] Munshi, A. The opencl specification version: 1.2 document revision: 15, 2011.

[103] NVIDIA, Tesla. V100 GPU architecture. The world's most advanced data center GPU. Version WP-08608-001_v1. 1. *NVIDIA 2017.* (2017), 108.

[104] Obukhov, A. Haar classifiers for object detection with cuda. *GPU Computing Gems Emerald Edition* (2011), 517–544.

[105] Oh, S., Russell, S., and Sastry, S. Markov chain monte carlo data association for multi-target tracking. *IEEE Transactions on Automatic Control 54*, 3 (2009), 481–497.

[106] Pearl, J. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014.

[107] Pedersoli, M., Vedaldi, A., Gonzalez, J., and Roca, X. A coarse-to-fine approach for fast deformable object detection. *Pattern Recognition 48*, 5 (2015), 1844–1853.

[108] Pirsiavash, H., Ramanan, D., and Fowlkes, C. C. Globally-optimal greedy algorithms for tracking a variable number of objects. In *CVPR* (2011), IEEE, pp. 1201–1208.

[109] Poole, D. Abducing through negation as failure: Stable models within the independent choice logic. *The Journal of Logic Programming 44*, 1-3 (2000), 5–35.

[110] Puttemans, S., and Goedemé, T. Visual detection and species classification of orchid flowers. In *MVA* (2015), IEEE, pp. 505–509.

[111] Puttemans, S., Goedemé, T., et al. Automated walking aid detector based on indoor video recordings. In *EMBC* (2015), IEEE, pp. 5040–5045.

[112] Puttemans, S., Van Ranst, W., and Goedemé, T. Detection of photovoltaic installations in rgb aerial imaging: a comparative study.

[113] Rayman, R., Primak, S., Patel, R., Moallem, M., Morady, R., Tavakoli, M., Subotic, V., Galbraith, N., Van Wynsberghe, A., and Croome, K. Effects of latency on telesurgery: an experimental study. In *International Conference on Medical Image Computing and Computer-Assisted Intervention* (2005), Springer, pp. 57–64.

[114] Redmon, J., and Farhadi, A. Yolo9000: better, faster, stronger. *arXiv preprint arXiv:1612.08242* (2016).

[115] Redmon, J., and Farhadi, A. Yolo9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242* (2016).

[116] Redmon, J., and Farhadi, A. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[117] Ren, S., He, K., Girshick, R., and Sun, J. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems* (2015), pp. 91–99.

[118] Renkens, J., Van den Broeck, G., and Nijssen, S. k-optimal: A novel approximate inference algorithm for problog. *Machine learning 89*, 3 (2012), 215–231.

[119] Riguzzi, F. Slgad resolution for inference on logic programs with annotated disjunctions. *Fundamenta Informaticae 102*, 3-4 (2010), 429–466.

[120] Riguzzi, F. Mcintyre: A monte carlo system for probabilistic logic programming. *Fundamenta Informaticae 124*, 4 (2013), 521–541.

[121] Riguzzi, F., and Swift, T. The pita system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming 11*, 4-5 (2011), 433–449.

[122] Riguzzi, F., and Swift, T. Well–definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and practice of logic programming 13*, 2 (2013), 279–302.

[123] Sánchez-Friera, P., Piliougine, M., et al. Analysis of degradation mechanisms of crystalline silicon pv modules after 12 years of operation in southern europe. *Progress in photovoltaics: Research and Applications 19*, 6 (2011), 658–666.

[124] Sato, T. A statistical learning method for logic programs with distribution semantics. In *IN PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING (ICLP'95* (1995), Citeseer.

[125] Sato, T., and Kameya, Y. Prism: a language for symbolic-statistical modeling. In *IJCAI* (1997), vol. 97, pp. 1330–1339.

[126] Scharr, H. Optimal filters for extended optical flow. In *Complex Motion*. Springer, 2007, pp. 14–29.

[127] Schmid, C., Mohr, R., and Bauckhage, C. Evaluation of interest point detectors. *International Journal of computer vision 37*, 2 (2000), 151–172.

[128] Schroff, F., Kalenichenko, D., and Philbin, J. Facenet: A unified embedding for face recognition and clustering. In *CVPR* (2015), pp. 815–823.

[129] Shafer, G. *The art of causal conjecture*. MIT press, 1996.

[130] Shi, J., and Tomasi, C. Good features to track. Tech. rep., Cornell University, 1993.

[131] Shterionov, D. S., Kimmig, A., Mantadelis, T., and Janssens, G. Dnf sampling for problog inference. *International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS)* (2010).

[132] Simo-Serra, E., Trulls, E., Ferraz, L., Kokkinos, I., Fua, P., and Moreno-Noguer, F. Discriminative learning of deep convolutional feature point descriptors. In *ICCV* (2015), IEEE, pp. 118–126.

[133] Sona, G., Passoni, D., Pinto, L., Pagliari, D., Masseroni, D., Ortuani, B., and Facchi, A. Uav multispectral survey to map soil and crop for precision farming applications. *Internation Archives Of The Photogrammetry, Remote Sensing And Spatial Information Sciences 41* (2016), 1023–1029.

[134] Song, B., Kamal, A. T., Soto, C., Ding, C., Farrell, J. A., and Roy-Chowdhury, A. K. Tracking and activity recognition through consensus in distributed camera networks. *TIP 19*, 10 (2010), 2564–2579.

[135] Struyf, L., De Beugher, S., Van Uytsel, D., Kanters, F., and Goedemé, T. The battle of the giants: A case study of gpu vs fpga optimisation for real-time image processing. vol. 1, VISIGRAPP; Lisbon, pp. 112–119.

[136] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI* (2017), vol. 4, p. 12.

[137] Tijtgat, N., Van Ranst, W., Volckaert, B., Goedeme, T., and De Turck, F. Embedded real-time object detection for a uav warning system. vol. 2018-January, IEEE, pp. 2110–2118.

[138] Tijtgat, N., Volckaert, B., and De Turck, F. Real-time hazard symbol detection and localization using uav imagery. In *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)* (2017), IEEE, pp. 1469–1474.

[139] Tomasi, C., and Kanade, T. Detection and tracking of point features.

[140] Tsai, D.-M., Wu, S.-C., et al. Defect detection in solar modules using ica basis images. *IEEE Transactions on Industrial Informatics 9*, 1 (2013), 122–131.

[141] Van Beeck, K. The automatic blind spot camera: hard real-time detection of moving objects from a moving camera.

[142] Van Beeck, K., Van Engeland, K., Vennekens, J., and Goedeme, T. Abnormal behavior detection in lwir surveillance of railway platforms. IEEE.

[143] Van Gelder, A., Ross, K. A., and Schlipf, J. S. The well-founded semantics for general logic programs. *Journal of the ACM (JACM) 38*, 3 (1991), 619–649.

[144] Van Ranst, W., Callemein, T., and Goedemé, T. The autonomous hidden camera crew. IEEE Xplore, pp. 34–37.

[145] Van Ranst, W., De Smedt, F., Berte, J., and Goedemé, T. Fast simultaneous people detection and re-identification in a single shot network. IEEE.

[146] Van Ranst, W., De Smedt, F., and Goedeme, T. Gpu accelerated acf detector. vol. 5, Ranst, W, Scitepress; Setúbal, pp. 242–248.

[147] Van Ranst, W., Goedeme, T., and Vennekens, J. Automatic endoscopic image orientation stabilisation with ultra-low-latency. *IADIS International Journal on Computers Science and Information Systems 11*, 2 (2016), 119–131.

[148] Van Ranst, W., Goedemé, T., and Vennekens, J. Ultra-low-latency automatic endoscopic image orientation stabilisation. IADIS, pp. 172–178.

[149] Van Ranst, W., and Vennekens, J. An opencl implementation of a forward sampling algorithm for cp-logic. Angelopoulos, Nicos, Vienna, pp. 1–14.

[150] Van Ranst, W., and Vennekens, J. An opencl implementation of a forward sampling algorithm for cp-logic. *International Journal of Approximate Reasoning 67* (2015), 60–72.

[151] Van Ranst, W., and Vennekens, J. Ultra-low-latency endoscopic image stabilisation, 2015.

[152] Van Ranst, W., and Vennekens, J. Ultra-low-latency endoscopic image stabilisation. *Benelux Conference on Artificial Intelligence (BNAIC)* (2015).

[153] Vennekens, J. Negation in the head of cp-logic rules. *Workshop on Answer Set Programming and Other Computing Paradigms* (2013).

[154] Vennekens, J., Denecker, M., and Bruynooghe, M. Cp-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and practice of logic programming 9*, 3 (2009), 245–308.

[155] Viola, P., and Jones, M. Rapid object detection using a boosted cascade of simple features. In *CVPR* (2001), vol. 1, IEEE, pp. I–511.

[156] Viola, P., Jones, M. J., and Snow, D. Detecting pedestrians using patterns of motion and appearance. In *null* (2003), IEEE, p. 734.

[157] Vondrick, C., Shrivastava, A., Fathi, A., Guadarrama, S., and Murphy, K. Tracking emerges by colorizing videos. In *European Conference on Computer Vision* (2018), Springer, pp. 402–419.

[158] Waldrop, M. M. The chips are down for moore's law. *Nature News 530*, 7589 (2016), 144.

[159] Wang, T.-C., Liu, M.-Y., Zhu, J.-Y., Liu, G., Tao, A., Kautz, J., and Catanzaro, B. Video-to-video synthesis. *arXiv preprint arXiv:1808.06601* (2018).

[160] Wojke, N., Bewley, A., and Paulus, D. Simple online and realtime tracking with a deep association metric. In *IEEE International Conference on Image Processing (ICIP)* (2017), pp. 3645–3649.

[161] Xiao, T., Li, S., Wang, B., Lin, L., and Wang, X. End-to-end deep learning for person search. *arXiv preprint*.

[162] Xiao, T., Li, S., Wang, B., Lin, L., and Wang, X. Joint detection and identification feature learning for person search. In *CVPR 2017* (2017), IEEE, pp. 3376–3385.

[163] Ye, T., Wang, X., Davidson, J., and Gupta, A. Interpretable intuitive physics model. In *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 87–102.

[164] Yu, F., Li, W., Li, Q., Liu, Y., Shi, X., and Yan, J. Poi: multiple object tracking with high performance detection and appearance feature. In *ECCV* (2016), Springer, pp. 36–42.

[165] Yuan, Y., Yang, K., and Zhang, C. Hard-aware deeply cascaded embedding. *CoRR, abs/1611.05720 1* (2016).

[166] Zagoruyko, S., and Komodakis, N. Learning to compare image patches via convolutional neural networks. In *CVPR* (2015), IEEE, pp. 4353–4361.

[167] Zhang, S., Bauckhage, C., Klein, D. A., and Cremers, A. B. Exploring human vision driven features for pedestrian detection. *IEEE Transactions on Circuits and Systems for Video Technology 25*, 10 (2015), 1709–1720.

[168] Zhang, S., Benenson, R., Omran, M., Hosang, J., and Schiele, B. How far are we from solving pedestrian detection? In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 1259–1267.

[169] Zhang, S., Benenson, R., Schiele, B., et al. Filtered channel features for pedestrian detection. In *CVPR* (2015), vol. 1, p. 4.

[170] Zhang, X., Luo, H., Fan, X., Xiang, W., Sun, Y., Xiao, Q., Jiang, W., Zhang, C., and Sun, J. Alignedreid: Surpassing human-level performance in person re-identification. *arXiv preprint arXiv:1711.08184* (2017).

[171] Zhou, Y., Liepe, J., Sheng, X., Stumpf, M. P., and Barnes, C. Gpu accelerated biochemical network simulation. *Bioinformatics 27*, 6 (2011), 874–876.

About the candidate



Wiebe Van Ranst was born on the 9th of November 1990 in Bornem, Belgium. He got his bachelor's degree in Electronics-ICT (major ICT) in 2011. Two years after that in 2013 he got his master's degree on the same subject. Both Wiebe's bachelor's and master's thesis were about GPU processing which was still a very small field at the time. His bachelor's thesis was about setting up a benchmarking system for the GPU computation language OpenCL, and his master's thesis was about a 3D mesh simplification algorithm in the same OpenCL language. Starting a PhD was a logical continuation of this. Wiebe started his PhD in 2013, after finishing his master's thesis, under professor Joost Vennekens. His main interest then mainly focused on artificial intelligence in general and GPU computing. In 2016 Wiebe briefly worked on a start-up company called *obtronics*, focusing on computer vision, the technology of which was one year later taken over by the company RoboVision. During this experience Wiebe's interests shifted towards GPU processing for computer vision, and later also computer vision itself. His research was then also supervised by professor Toon Goedemé. This work towards a PhD is the result of his interest in a wide spectrum of topics.

Apart from that, in his spare time, Wiebe participates intensively in the sport known as ultra trail running, running what some people would consider, stupidly

long distances in mountainous environments. Apart from his training grounds in Belgium he does many international races. Wiebe finished his first hundred miler in the Swiss Alps in september of 2018.

List of publications

Conference publications

[1] W Van Ranst, F De Smedt, J Berte, and T Goedemé. Fast simultaneous people detection and re-identification in a single shot network. In *IEEE International Conference on Advanced Video and Signal-based Surveillance (AVSS)*, number 70, Dec 2018.

[2] W Van Ranst, F De Smedt, and T Goedeme. Gpu accelerated acf detector. In *VISIGRAPP 2018 - Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISAPP)*, volume 5, pages 242–248, Jan 2018.

[3] W Van Ranst, T Callemein, and T Goedemé. The autonomous hidden camera crew. In *Proceedings of the 15th IAPR International Conference on Machine Vision Applications (MVA)*, number 04-03, pages 34–37. Nagoya, Japan, IEEE Xplore, May 2017.

[4] W Van Ranst, T Callemein, and T Goedemé. Automatic camera selection and ptz canvas steering for autonomous filming of reality tv. In W Bares, V Gandhi, Q Galvane, and R Ronfard, editors, *Eurographics Workshop on Intelligent Cinematography and Editing*, number 1013/20171073. Lyon, France, The Eurographics Association, Apr 2017. * T. Callemein and W. Van Ranst contributed equally to this paper.

[5] N Tijtgat, W Van Ranst, B Volckaert, T Goedeme, and F De Turck. Embedded real-time object detection for a uav warning system. In *IEEE International Conference on Computer Vision workshops (ICCVW) - International Workshop on Computer Vision for UAVs (UAVision)*, IEEE International Conference on Computer Vision Workshops, pages 2110–2118. Venice, Italy, Jan 2017.

[6] S Puttemans, W Van Ranst, and T Goedemé. Detection of photovoltaic installations in rgb aerial imaging: a comparative study. In *GEOBIA 2016 proceedings*, number 34. Enschede, The Netherlands, Sep 2016.

[7] W Van Ranst, T Goedemé, and J Vennekens. Ultra-low-latency automatic endoscopic image orientation stabilisation. In *Proceedings of the International Conference on Computer Vision and Image Processing - Part of the Multi Conference on Computer Science and Information Systems*, pages 172–178, Jan 2016.

[8] W Van Ranst and J Vennekens. An opencl implementation of a forward sampling algorithm for cp-logic. In N Angelopoulos and A Kimmig, editors, *International Conference on Inductive Logic Programming - Workshop on Probabilistic Logic Programming*, pages 1–14. Vienna, Austria, 2014.

Journal articles

[1] W. Van Ranst, T. Goedeme, and J. Vennekens. Automatic endodoscopic image orientation stabilisation with ultra-low-latency. *IADIS International journal on computer science and information systems*, 11:119–131, Jan 2016.

[2] W. Van Ranst and J. Vennekens. An opencl implementation of a forward sampling algorithm for cp-logic. *International journal of approximate reasoning*, 67:60–72, Dec 2015.

Demos

[1] W. Van Ranst and J. Vennekens. Ultra-low-latency endoscopic image stabilisation, 2015.



FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
EAVISE
Jan De Nayerlaan 5, B-2860 Sint-Katelijne-Waver
B-3001 Leuven
first.last@kuleuven.be
http://eavise.be