Real-time Video Scene Analysis with Heterogeneous Processors

Calum Grahame Blair M.Eng.

A thesis submitted to
The Universities of
Glasgow,
Edinburgh,
Heriot-Watt,
and Strathclyde

for the degree of
Doctor of Engineering in System Level Integration

May 2014

Abstract

Field-Programmable Gate Arrays (FPGAs) and General Purpose Graphics Processing Units (GPUs) allow acceleration and real-time processing of computationally intensive computer vision algorithms. The decision to use either architecture in any application is determined by task-specific priorities such as processing latency, power consumption and algorithm accuracy. This choice is normally made at design time on a heuristic or fixed algorithmic basis; here we propose an alternative method for automatic runtime selection.

In this thesis, we describe our PC-based system architecture containing both platforms; this provides greater flexibility and allows dynamic selection of processing platforms to suit changing scene priorities. Using the Histograms of Oriented Gradients (HOG) algorithm for pedestrian detection, we comprehensively explore algorithm implementation on FPGA, GPU and a combination of both, and show that the effect of data transfer time on overall processing performance is significant. We also characterise performance of each implementation and quantify tradeoffs between power, time and accuracy when moving processing between architectures, then specify the optimal architecture to use when prioritising each of these.

We apply this new knowledge to a real-time surveillance application representative of anomaly detection problems: detecting parked vehicles in videos. Using motion detection and car and pedestrian HOG detectors implemented across multiple architectures to generate detections, we use trajectory clustering and a Bayesian contextual motion algorithm to generate an overall scene anomaly level. This is in turn used to select the architectures to run the compute-intensive detectors for the next frame on, with higher anomalies selecting faster, higher-power implementations. Comparing dynamic context-driven prioritisation of system performance against a fixed mapping of algorithms to architectures shows that our dynamic mapping

method is 10% more accurate at detecting events than the power-optimised version, at the cost of 12W higher power consumption.

Acknowledgements

I would like to acknowledge the consistent and enthusiastic help and constructive advice given to me by my supervisor, Neil Robertson, throughout the course of this doctorate.

I would also like to thank Siân Williams for all her procedural advice, before, during and after the winding-up of the ISLI.

I'm also grateful for the work done by Scott Robson during his internship at Thales. Acknowledgements are also given to the funders of this research, EPSRC and Thales Optronics.

Thanks are due also to my friends especially Chris, Kenny and Johnathan, for dragging me out to the pub whenever this degree started to get too overwhelming. Doubly so for those – including Marek – willing to accompany me as I dragged them up and down various Munros.

My thanks also go to Rebecca for her continued understanding, patience and enthusiasm.

Above all, I would like to thank my family, Mum, Dad, Mhairi and Catriona, for all the support and encouragement they have given me throughout this period, and particularly for their frequent offers to appear — especially with the dog — in my video datasets.

Contents

List of Publications

- *Characterising Pedestrian Detection on a Heterogeneous Platform*, C. Blair, N. M. Robertson, and D. Hume, in Workshop on Smart Cameras for Robotic Applications (SCaBot '12), IROS 2012.

- *Characterising a Heterogeneous System for Person Detection in Video using Histograms of Oriented Gradients: Power vs. Speed vs. Accuracy*, C. Blair, N. M. Robertson, and D. Hume, IEEE Journal of Emerging and Selected Topics in Circuits and Systems, **V3(2)** pp. 236–247, 2013.

- *Event-Driven Dynamic Platform Selection for Power-Aware Real-Time Anomaly Detection in Video*, C. G. Blair & N. M. Robertson, International Conference on Computer Vision Theory and Applications (VISAPP) 2014.

List of Tables

List of Figures

List of Abbreviations

AP	Activity Path.
API	Application Programming Interface.
ASIC	Application-Specific Integrated Circuit.
ASR	Addressable Shift Register.
BAR	Base Address Register.
CLB	Combinatorial Logic Block.
COTS	Commercial Off-the-Shelf.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DET	Detection Error Tradeoff.
DMA	Direct Memory Access.
DSE	Design Space Exploration.
FIFO	First-In First-Out buffer.
FPGA	Field Programmable Gate Array.
FPPI	False Positives per Image.
FPPW	False Positives per Window.
FPS	Frames per second.
GB/S	Gigabytes per second.
GPGPU	General-Purpose Graphics Processing Unit.
GPU	Graphics Processing Unit.

GT/S Gigatransfers per second.

HOG Histogram of Oriented Gradients.

i-LIDS Imagery Library for Intelligent Detection Systems.
ISTAR Intelligence, Surveillance, Target Acquisition, and Reconnaissance.

MAC/S Multiply-Accumulate Operations per second.
MB/S Megabytes per second.
MOG Mixture of Gaussians.
MPS Maximum Payload Size.

NMS Non-Maximal Suppression.
NPP Nvidia Performance Primitives.

PCIE PCI Express.
PE Processing Element.
POI Point of Interest.

QVGA Quarter VGA, 320×240 resolution.

RBF Radial Basis Function.
ROC Receiver Operating Characteristic.
RTL Register Transfer Level.

SBC Single-Board Computer.
SIMD Single Instruction Multiple Data.
SIMT Single Instruction Multiple Thread.
SM Streaming Multiprocessor.
SP Stream Processor.
SSE Streaming SIMD Extensions.
SVM Support Vector Machine.

TLP Transaction Layer Packet.

Declaration of Originality

Except where I have explicitly acknowledged the contributions of others, all work contained in this thesis is my own. It has not been submitted for any other degree.

1. Introduction

Computer vision, or the science of extracting meaning from images, is a large and growing field within the domains of electronic engineering and computer science. As humans, vision is our primary sense and many of our everyday tasks depend heavily on an ability to see our surroundings. Teaching or programming machines to perceive the world as we do opens up a myriad of possibilities: routine, repetitive tasks can be automated, dangerous situations made safer, and many more options for entertainment become feasible. Autonomous vehicles equipped with cameras allow us to explore areas of our world and universe which would be extremely hostile to humans. Grand aims such as these cover much of the motivation for research in this field.

From an engineering perspective, many tasks within computer vision are difficult problems. The human brain has specialised hardware built for processing information from images, with a design time of millions of years. It is capable of forming images, extracting shapes, recognising objects, inferring meaning and intent to observed motion, and using this information to interact with the world around it — fast enough that we can catch a flying ball or step out of the way of a speeding car. A machine built or programmed to perform tasks which require interpretation of visual data must operate *accurately* enough to be effective and complete its task fast enough that the data it extracts is timely enough to be usable. In many cases, this is in real time; we must process images at the same speed or faster than they are received, and we accept some known time delay or *latency* between starting and finishing processing of a single image.

And what of the underlying processing hardware that we rely on to do this work? The state of the art in electronics has continued to advance rapidly; using computers built within the last few years we can now make reasonable progress towards creating implementations of complex signal processing algorithms which can run in real

time. These same advances have allowed devices containing sensors and processors to shrink to where they become handheld or even smaller. Their ubiquity and low cost, along with their size, further expand the potential benefits of mobile computing systems, and offer even more applications for embedded or autonomous vision systems. However, the *power consumption* of any machine must be considered, and this is the limiting factor affecting processing devices at all scales, from handhelds to supercomputers. These three characteristics — power consumption, latency and accuracy — are ones which we will return to repeatedly in this thesis.

The thesis itself describes the research undertaken for the Engineering Doctorate in System Level Integration. The work is in the technical field of characterization and deployment of heterogeneous architectures for acceleration of image processing algorithms, with a focus on real-time performance. This was carried out in combination with the Visionlab, part of the Institute for Sensors, Signals and Systems at Heriot-Watt University[1], and Thales Optronics Ltd[2]. It was sponsored jointly by the Engineering and Physical Sciences Research Council (EPSRC) and Thales Optronics. It was managed by the Institute for System Level Integration, a joint venture between the schools of engineering in the four Universities of Glasgow, Edinburgh, Heriot-Watt and Strathclyde. Operating between 1999 and 2012, it ran courses for postgraduate taught and research students, along with commercial electronics design consultancy services. Its website was shut down following its closure in 2012, but an archived copy is available[3].

This chapter is laid out as follows: in Section 1.1 we give an overall statement of the problem studied and our motivation for conducting research in this area. As the EngD involves carrying out commercially relevant research, Section 1.2 places this work in a commercial context and gives the business motivation behind it. We then concentrate on the specific aims of this thesis in Section 1.3. This is followed in Section 1.4 by our research outputs and knowledge transfer outputs to industry. Finally, Section 1.5 states the contributions made by this work and Section 1.6 gives a roadmap for the rest of this thesis.

[1] http://visionlab.eps.hw.ac.uk
[2] http://www.thalesgroup.com/
[3] http://web.archive.org/web/20130527020950/http://www.isli.ac.uk/

Figure 1.1: Land defence vehicles such as the British Army's Mastiff now include cameras for local situational awareness.

1.1 Academic Motivation and Problem Statement

We start by considering the problem of situational awareness. Locally, this involves monitoring of one's own environment. In a military situation, simply looking at a scene to identify threats has its own problems; visual range is limited, and merely being in an unsafe area to monitor it involves some level of risk to the observers. Visual and infrared sensors allow situational awareness of both local and remote environments with reduced risk; the current generation of land defence vehicles for the British Army now include multiple cameras for this reason (see Figure 1.1).

However, the deterioration of performance of human operators over time when performing *vigilance* tasks such as monitoring radar or closed-circuit TV screens, or standing sentry duty, is well-known [1]. It was first established by Mackworth in 1948; he showed that human capability to detect events decreased dramatically after only half an hour on watch, with this degradation continuing over longer watches [2]. Donald argues that CCTV surveillance falls under the taxonomy of vigilance work and should be treated the same way [3]. In both military and civilian domains, there is thus a clear benefit to deploying machines which can perform automated situational awareness tasks.

1.1.1 A Motivating Scenario

We now consider the situations in which such a machine could be deployed. The vehicle in Figure 1.1 is likely to perform two main types of tasks: (i) situational awareness while moving and on patrol, and (ii) surveillance while stationary. In each

case, some image processing of visual or infrared sensor data must be done. When the vehicle is moving, fast detections and a high framerate may be required so that actions may be taken quickly, in response to changes in the vehicle's environment which may pose a threat. The engine will be running, so plenty of electrical power will be available for image processing. In the second scenario, we assume the vehicle is performing surveillance and is stationary with the engine turned off. Any processing done in this state should not drain the battery to the point where (i) the engine can no longer start or possibly (ii) where continued surveillance operations become impossible. The operating priorities of such a system will change so that power conservation becomes more important than fast processing.

Expanding on this, if we consider a scenario where the degree of computational operations increases with the number of objects or amount of clutter in an image then the weighting given to power consumption, latency and accuracy of object classification may change dynamically. This would require the system to either change the way it processes data (starting or stopping processing entirely) or moving processing to different platforms more suited to the current priorities.

In an ideal world, we would have a processing platform and an algorithm which is *the most accurate, the fastest and the least power-hungry when compared to all possible alternatives*. However, as we explain in detail later in this thesis, any combination of processor and algorithm involves a compromise and no such consistently optimal solution exists. Any solution is a tradeoff between power, time, accuracy, and various other less critical factors. It is this *problem of adapting our system performance and behaviour to best fit the changing circumstances of the operating environment* that we wish to study here.

So far we have used the example of a military patrol vehicle, but this problem is also one faced by autonomous vehicles or remotely operated sensors — indeed, any device which must conserve battery power while doing some kind of signal processing. This would encompass civilian applications such as disaster recovery or driver assistance, as well as the military example we use throughout this thesis.

Figure 1.2: An example scene: normal pedestrian and vehicle behaviour is to some extent dictated by the structure of the scene, and these patterns can be learned via prolonged observation. However, unexpected behaviour (cars driving onto pavement or running red lights, or a person running across the road) is still possible.

1.1.2 Specifying Surveillance Subtasks

Given that we wish to automate some existing surveillance task – under power and complexity constraints – we now consider what this might involve. We choose to focus on the detection of pedestrians and vehicles, for several reasons:

- Humans (and, to a lesser extent, vehicles controlled by humans) are arguably the most important objects in any scene. They will often have a routine or pattern of life affected by their surroundings, but be capable of easily deviating from this. Consider the scene in Figure 1.2; the position of the road and pavement influences pedestrian and vehicle location, and features such as traffic lights and double yellow "No Parking" lines influence their behaviour – but not to the extent that illegal parking or jaywalking is impossible.

- There are clear advantages to deploying this technology in military and civilian applications, and a tangible benefit to doing this in real time. The car manufacturer Volvo is already including pedestrian detection systems for driver assistance which rely on video and radar in their latest generation of cars [4]. However, doing this on a mobile phone-sized device and without relying on active sensing is still a challenge.

- The algorithms necessary to perform pedestrian detection generalise well to other object detection tasks; *e.g.* an existing pedestrian detector can produce state-of-the-art results when applied to a road sign classification problem [5].

1.1.3 Wider Motivations

The UK Ministry of Defence's research division, the Defence Science and Technology Laboratory, has identified around thirty technical challenges in the area of signal processing [6], and, together with the Engineering and Physical Sciences Research Council, has provided £8m in funding for research which will directly address these, under the umbrella of the Universities Defence Research Collaboration[4]. While these were formulated well after this project was started, the themes of this thesis are nevertheless applicable to the open problems faced by the wider defence and security research community today. Several UDRC challenges touch on the area of anomaly detection in video ("Video Pattern Recognition" and "Statistical Anomaly Detection in an Under-Sampled State Space"), while another specifically addresses the implementation of algorithms on mobile or handheld devices ("Reducing Size, Weight and Power Requirements through Efficient Processing").

In the civilian domain, the UN World Health Organisation's 2013 Road Safety Report notes that half of all road deaths are from vulnerable traffic users (pedestrians, cyclists, and motorcyclists) and calls for improved infrastructure and more consideration of the needs of these vulnerable users. Starting in 2014, the European New Car Assessment Programme will include test results of Autonomous Emergency Braking systems for cars. These detect pedestrians or other vehicles ahead of the car, then brake automatically if the driver is inattentive [7]. Finally, in 2013 the first instance of an unmanned aerial vehicle being used to locate and allow the rescue of an injured motorist was recorded [8], demonstrating the applications of this technology for disaster recovery scenarios in the future.

To summarise our motivations at this point: within the field of computer vision, the problem of pedestrian and vehicle detection has a wide variety of applications, many of which involve anomaly detection and surveillance scenarios. Many of these scenarios require real-time solutions operating under low power constraints. We comprehensively survey progress towards these solutions in Chapter 2, but we note

[4] http://www.mod-udrc.org/technical-challenges

here that this is an open field with advances required in all three metrics of accuracy, speed and power.

1.2 Commercial and Industrial Motivation

There are several commercial factors which have influenced this work. We start by briefly considering the field of high-performance computing, then narrow our focus to look at the factors affecting Thales Optronics.

Within the last decade, computing applications have no longer been able to improve performance by continually increasing the clock speed of the processors they run on. The "power wall" acts to limit the upper clock speed available, and development efforts have instead focused on increasing the number of cores in a processor; the "Concurrency Revolution" [9]. This allows improved performance of concurrent and massively parallel applications. Taken to its logical conclusion, this has allowed, firstly, the development of processors with thousands of cores on them, all capable of reasonable floating-point performance [10]; secondly, division of labour inside a computer system or network. A *multicore* processor optimised for fast execution of one or two threads may control overall program flow, but the *embarrassingly parallel* calculations which make up the majority of "big data" scientific data computation and signal processing operations can be farmed out to throughput-optimised *massively multicore accelerators*. Such an approach is known as *heterogeneous computing*. The validity of this approach is borne out by the Top 500 list of most powerful supercomputers; as of November 2013, 53 computers on the list were using some form of accelerator, including the first and second most powerful (using Intel Xeon Phi and Nvidia Graphics Processing Unit (GPU) accelerators respectively) [11].

As we will discuss in Chapter 2, the choice of processing platform to use for a specific application has significant implications for performance. Thales is an engineering firm which designs and manufactures opto-electronic systems for applications throughout the defence sector, including naval, airborne and land defence. Changing customer requirements in recent years have lead to an increase in the processing capability included in the systems they develop. This is part of a move from current image *enhancement* (such as performing non-uniformity correction on the output from an infrared camera) to near-term future image *processing* capability,

such as detection and tracking of potential targets. The Ministry of Defence has formalised the requirements for interoperability of such systems for land defence applications [12], meaning that cameras from one vendor can in theory be paired with signal processing equipment from another, and processing equipment can be easily upgraded when required.

Thales are thus concerned with the deployment of image-processing algorithms in embedded systems, and are aware that such technology operating with real-time performance has a wide variety of current and future applications, limited in many cases by the size, weight and power of any developed solution. As these are designed for military operations, various other economic factors must be considered. Small, irregular production runs are the norm. Rather than a company defining its own product release roadmap to a regular schedule as in the telecommunications industry, development and release of new products is customer-driven in response to contracts or tenders. Products must also be supported by the manufacturer for much longer than commercial devices; requirements to be able to provide support and replacement parts for twenty years are not unusual. Military devices must also operate in more extreme temperature ranges than commercial products. Taken together, all these constraints preclude the use of Application-Specific Integrated Circuits (ASICs), many Commercial Off-the-Shelf (COTS) parts, and the ability to make use of economies of scale. In the last decade or so, Field Programmable Gate Arrays (FPGAs) have been used to perform most image and signal processing tasks in embedded systems. FPGA boards are available in form factors designed for defence applications, such as OpenVPX cards. Unlike ASICs, FPGAs can be reprogrammed at some point in their operational lifetime to add new features, without replacing the entire unit.

However, the long development times and limited potential for component reuse between different designs (a high *Non-Recurring Engineering* cost) have meant that FPGA development has been regarded as time-consuming, complicated and expensive. The recent growth of GPU computing has offered firms like Thales an alternative to this. The faster development cycle of GPU programming and in some cases its lower cost must be weighed against a probable increase in power consumption when compared to FPGA. Another major concern is availability of parts in twenty years time, particularly for products where a new generation is launched around

every 18–24 months. The wide availability of highly optimised matrix mathematics libraries on GPU may further reduce development time.

GPUs have another advantage over ASICs in that they are quickly reprogrammable at runtime (new kernels can be launched in under $10\mu s$ [13]). Dynamically reconfigurable FPGAs also behave similarly. These approaches allow the same hardware to be used for different tasks within the same mission, reducing the size, weight and power of the equipment carried. (As an example, consider a system running different algorithms on the same processing platforms, using visual sensors in daylight and infrared at night, or automatically selecting different segmentation or detection algorithms in urban and rural environments). Again, the differing approaches of the GPU ("run a new task on a fixed architecture") and FPGA ("shut down parts of the chip and reprogram it") to these changing mission profiles should be contrasted.

A comparison of the performance of FPGA compared to GPU for image processing applications, then, is a pressing business requirement for Thales. This can be split into a commercial side — studying hardware costs, design time and expenditure, and how to manage longevity — and a technical one. The technical study would use one or more signal processing tasks to investigate the relative performance of FPGA and GPU in the three metrics of power, latency and accuracy, as these have direct and indirect effects on SWaP.

We concentrate on the technical question in this thesis. Previous comparisons have been reported in the literature, and are discussed in Chapter 2. These assume a direct choice between a single FPGA and GPU in a system. We wished to build on this earlier work by characterizing the performance of a joint system containing three processors: FPGA, GPU and Central Processing Unit (CPU). Such a system, if built today, would have little integration between the different accelerator types; the complexity of data transfer between devices has already been demonstrated [14]. However, commercial embedded devices containing both reconfigurable logic and manycore processors on the same platform are now becoming available (such as the Parallella[5]). In the near future, integration of these devices on the same die can be expected, and this approach could offer substantial performance and SWaP improvements.

[5] http://www.parallella.org/board/

1.3 Aims

To summarise our dual motivations from the previous section, we wish to investigate the performance of processing architectures capable of pedestrian and vehicle detection, within a surveillance context. Conceptually, we use a vehicle with some onboard processing capability as a target platform, while keeping in mind its power constraints.

Our commercial motivations involve ascertaining the best architecture to run such a system on, and also whether or not a system with multiple heterogeneous processors outperforms *e.g.* a single-GPU one. As we argue in the previous section, knowledge gained from studying this problem has implications for defence and civilian applications, and is both relevant and timely. We thus apply our academic and industrial motivations to a specific problem within the field of surveillance.

This work aims to answer two questions:

1. *"How does the performance of an algorithm when partitioned temporally across a heterogeneous array of processors compare to the performance of the same algorithm in a singly-accelerated system, when considering a real-world image processing problem?"*

2. *"What is the optimal mapping of a set of algorithms to a heterogeneous set of processors? Does this change over time, and does a system with this architecture offer any advantage in a real-world image processing task?"*

We answer these in detail in Chapters 5 and 6 respectively, while the remainder of this thesis places this in more context and provides details of the underlying hardware which these results depend on. Chapter 5 considers the effects on performance of partitioning parts of a single algorithm, while Chapter 6 addresses the same topic at task level.

Note that throughout this work we refer to "real-time" operation. This uses the "soft" definition of real-time computing, where results received after a deadline are less useful. In a "hard" real-time system, failure to generate results by a deadline would be catastrophic. We use the frame rate of 30 frames per second, and accept a small measure of latency during processing.

1.4 Knowledge Transfer

1.4.1 Research Outputs

- A workshop paper [15] was presented at the Workshop on Smart Cameras for Robotic Applications at the IEEE Conference on Intelligent Robots and Systems in 2012.

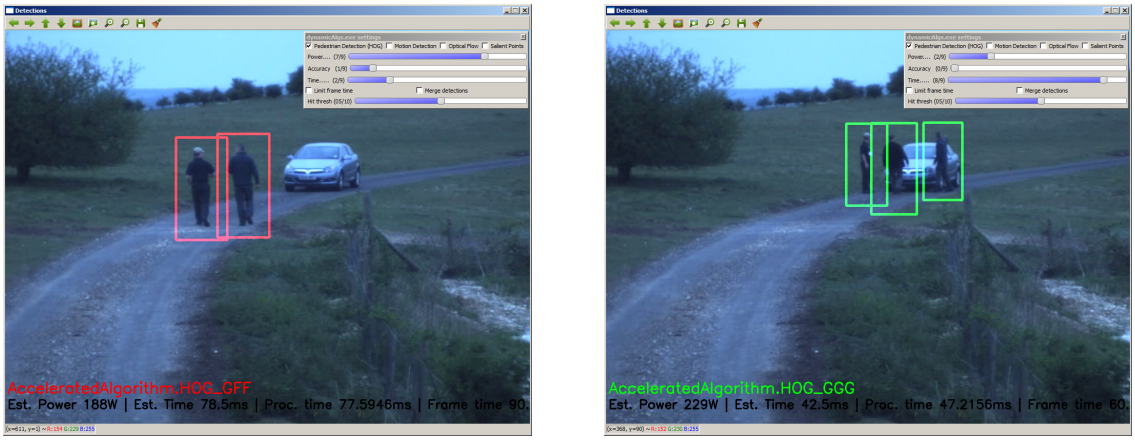
- This was followed by a longer journal paper "Characterising a Heterogeneous System for Person Detection in Video using Histograms of Oriented Gradients: Power vs. Speed vs. Accuracy" [16], based on the work carried out in Chapter 5. This was published in a special issue on Smart Cameras in the IEEE Journal of Selected and Emerging Topics in Circuits and Systems.

- An invited talk on the subject of "Power, Speed and Accuracy Tradeoffs: Characterising a Heterogeneous System for Person Detection in Video using HOG" was given at a BMVA Symposium on "Vision in an Increasingly Mobile World", in May 2013.

- A paper was presented at the International Conference on Computer Vision Theory and Applications, in January 2014. This was based on the work in Chapter 6, titled "Event-Driven Dynamic Platform Selection for Power-Aware Real-Time Anomaly Detection in Video" [17]. This was accepted for a full oral presentation.

1.4.2 Knowledge Transfer within Thales

Thales Research and Technology, the research division within the multinational Thales Group, hosts an annual research day called "Journee de Palaiseau". This allows PhD students seconded to various divisions and countries within Thales, who are working on a common theme defined as "Software and Critical Information Systems", to present updates to their work and explore opportunities for collaboration. Work from this thesis was presented at these days on two occasions. Based on this, the algorithms discussed within this thesis were considered for implementation on another hardware architecture platform developed within Thales. This involved undertaking training on the Ter@pix architecture and the steps required to evaluate its performance on an algorithm. This occurred both at a low level, involving



(a) Power consumption given priority (b) Processing time given priority

Figure 1.3: Screenshots of demonstration platform with user-driven selection of performance priorities. In (a), increasing the priority of "Time" causes algorithm processing to be moved from FPGA to GPU. This increases speed at the expense of power consumption, as shown at the bottom of (b).

operators (analogous to Compute Unified Device Architecture (CUDA) kernels or basic image processing steps), and a higher algorithmic level, involving operator performance, data processing capacity and host/device transfer characteristics. Ultimately, the Ter@pix platform was not used in this project, but this is discussed further in Chapter 3 and Chapter 7.

A demonstration of the dynamic architecture selection parts of this thesis (a user-driven version of the system described in Chapter 6) was also given at a Thales Research Day, in conjunction with another student's work. In this technology demonstrator, emphasis was given to changing power, time and accuracy priorities and their effect on dynamic selection of algorithm implementations within a system. Examples of this are shown in Figure 1.3. A main theme in other work shown at this exhibition was products to improve Intelligence, Surveillance, Target Acquisition, and Reconnaissance (ISTAR). These were demonstrated to various customers of Thales in the defence and security sectors, and conveyed Thales' capability for system development in the future. The demonstration we gave also fitted within this broad theme.

Throughout this project, several presentations were also given to engineers and managers within Thales to inform them about current research developments, and

to receive feedback on potential approaches for future work. Finally, priorities for future architecture and system-level research within Thales have been identified based on the conclusions from work documented in this thesis.

1.5 Contributions

The key contributions of this thesis are as follows:

- We give a comprehensive analysis of the performance of a complex signal processing algorithm when applied to a platform with multiple heterogeneous accelerators (FPGA and GPU). Taking into account processing time, power consumption and accuracy, we show the cost (in absolutes and in percentage change from best measurement for that characteristic) of trading one of these against the other. An example of this is shown in Figure 1.4.

- We construct and describe the performance of a real-time image processing system for anomaly detection. This is capable of detecting vehicles parked in prohibited locations, as shown in Figure 1.5. This system responds to events within a scene by dynamically modifying the arrangement of processing elements it uses and and hence its power consumption characteristics. From this we show a clear tradeoff of event detection accuracy against power consumption. We also show the tradeoffs made when moving algorithm subtasks between heterogeneous processors; see Figure 1.6.

1.6 Thesis Roadmap

The remainder of this thesis is laid out as follows:

- **Chapter 2** describes related work. This covers the architecture of the various processors used, examples of their use in image processing to date, and relevant object and anomaly detection algorithms used throughout the thesis. We also consider techniques for mapping algorithms to architecture.

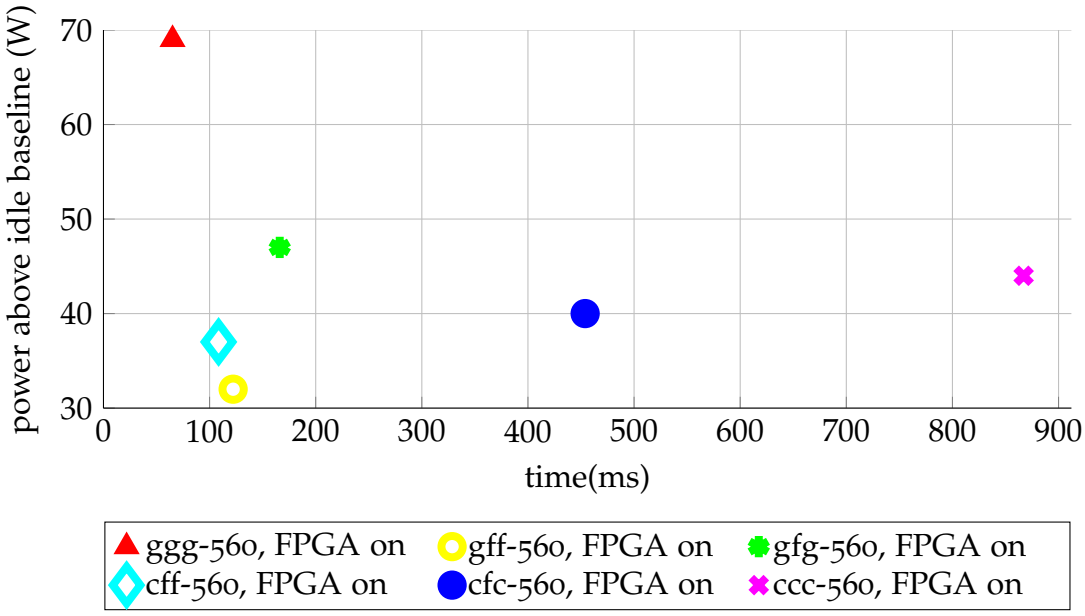


Figure 1.4: Run-time design space exploration: power vs. time for various implementations of HOG pedestrian detection using a GPU and FPGA. Power consumption shown as increase over baseline of 147W. Each version shown here can be selected at runtime. Letters denote the architecture which each algorithm segment is run on; *e.g.* for *gff*, resizing is done on GPU, followed by feature extraction and classification on FPGA.

Figure 1.5: Real-time anomaly detection. The van parked on the left-hand side of the road is highlighted with a red square, signifying an anomaly. The overlaid text shows current system performance characteristics.

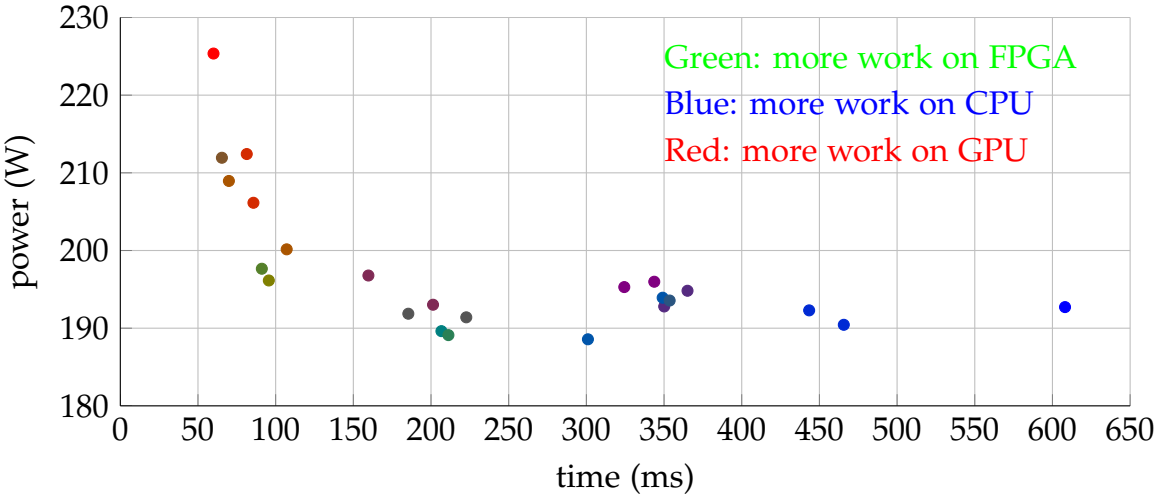


Figure 1.6: Power and time plots of all possible solutions for car, pedestrian and motion detectors across FPGA and GPU. A mainly red dot indicates most processing is done on GPU, while a dot closer to green indicates most processing is done on FPGA.

- **Chapter 3** moves on from the academic literature to consider implementation details. We explore a simulation-oriented compared to a hardware-only approach and consider whether image segmentation is required. We then focus on our choice of heterogeneous processors and discuss algorithms for exploring design space.

- **Chapter 4** is shaped by the previous chapter, and documents the system architecture we will use to perform real-time detection and hence surveillance. We give specifications of the processors used and discuss the interface for data transfer between them.

- **Chapter 5** uses the system constructed in the previous chapter. Here we perform an in-depth study of the performance characteristics which result from implementing the Histogram of Oriented Gradients algorithm for pedestrian detection on a system of heterogeneous processors: FPGA, GPU and CPU. We analyse the algorithm, identify the different types of computation involved in each stage of the algorithm (resizing, feature extraction and classification), and justify our approach to partitioning computation between architectures in this way. We then report power, accuracy and latency numbers for each of six

arrangements, and the tradeoffs involved in moving between arrangements: i.e. *if power consumption is reduced by 10%, how much longer does processing take?*

- **Chapter 6** builds on the work of Chapters 4 and 5 and describes a system for anomaly detection in video. This performs detection of parked vehicles in real time by dynamically allocating parts of the detection algorithms onto each processor (FPGA, GPU and CPU) depending on the level of anomaly seen in the frame. Again, we explore the performance of Histogram of Oriented Gradients (HOG) when running both car and pedestrian detections, and show the resulting tradeoffs between power, accuracy and processing time. As this system operates in real time, we concentrate on power and accuracy; *if power consumption is reduced by 10%, how many more parked vehicle events will be missed?*

- **Chapter 7** concludes this thesis. Here we summarise the key points of each chapter and highlight relevant results. We finish with a short discussion on directions for future work.

Note that in system architecture and processing diagrams throughout this thesis, we have tried to use a consistent colour scheme. Yellow boxes signify operations carried out on FPGA or the FPGA itself. Similarly, blue boxes represent GPU operations, red ones refer to work done on CPU, and green boxes represent accesses to host memory from any device.

2. Related Work

The problem of obtaining real-time performance from sophisticated image processing algorithms operating on large quantities of data is important and timely. This is evidenced by the ongoing focus of both industrial and academic research and development. In this chapter dealing with existing literature, we cover four relevant topics as part of this problem:

1. *current hardware architectures for generalised and parallelised data processing and approaches to programming them;*

2. *a description of certain processing-intensive image processing algorithms for object detection and classification;*

3. *a survey of higher-level algorithms for scene surveillance and anomaly detection;*

4. *a review of approaches taken to the problem of assigning algorithms to a hardware platform.*

Following this, we summarise and restate the problem around which this thesis is centred; that of dynamic mapping of algorithms to hardware.

2.1 Data Processing Architectures

In recent years, computer architectures designed for massively-parallel data processing have become more widespread and affordable; alongside this, embedded versions of these same processors have become available. Using these, tasks such as face detection [19], which would have been infeasible in real-time ten years ago, are now performed in realtime within most consumer cameras and mobile phones [20].

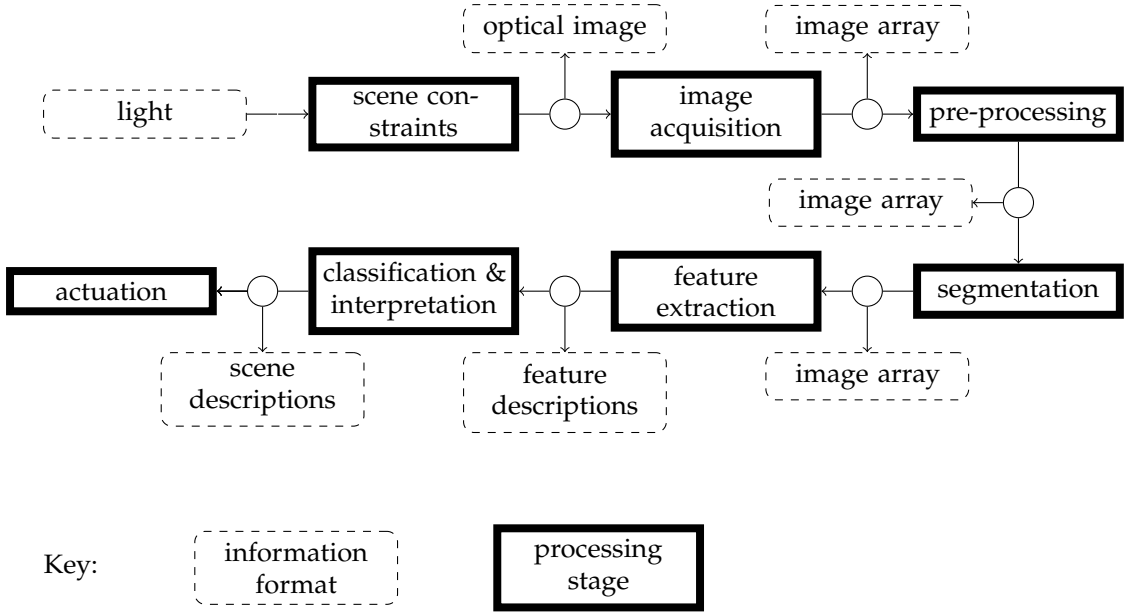


Figure 2.1: Image Processing Pipeline (from Awcock & Thomas [18]). Each stage in the pipeline can be considered another layer of abstraction.

We now review the various platforms for algorithm acceleration which were either used or considered for use in this work. Any implementation of an algorithm on one or more of these platforms will exist at some point in *design space*. This is defined as a multidimensional space with dimensions specific to the problem at hand, such as power consumption, chip area, ease of programming, processing time, and accuracy of result [21].

2.1.1 Processor Taxonomy

We start by considering the domain of image processing algorithms in more detail. Figure 2.1 shows a standard machine vision processing pipeline, as described by Awcock and Thomas in 1995, and still widely in use today [18]. Applying the *Berkeley dwarves paradigm* to this pipeline is instructive.

The Berkeley dwarves are defined as "algorithmic method[s] that capture a pattern of computation and communication" which "present a method for capturing the common requirements of classes of applications while being reasonably divorced from individual implementations" [22]. The original seven computational dwarves were: dense and sparse linear algebra, spectral methods, n-body methods, structured and unstructured grids and Monte Carlo methods. In a wide-ranging technical report from Berkeley, Asanovic *et al*. renamed Monte Carlo to the more

general MapReduce, and extended this list to thirteen to include combinational logic, graph traversal, graphical models, finite state machines, dynamic programming and backtrack and branch-and-bound.

These dwarves were based on a generalisation of existing benchmarks; this approach allows classification of signal processing operations into groups. The most relevant dwarf to image processing is arguably dense linear algebra (vector-vector, matrix-vector and matrix-matrix operations). Specifically, all processing operations described in the rest of this thesis use dense linear algebra. The only exception is the trajectory clustering algorithm described in Chapter 6 which we class as graph traversal (object property search, involving "indirect table lookups and little computation"). However, this is not computationally demanding enough to consider as a candidate for acceleration.

Other researchers note that vision processing is inherently parallel, and is one of the application domains described as "embarrassingly parallel" [23, 24], especially the early pixel-processing operations found when working at low levels of abstraction. Embarrassingly parallel applications are those which have "a high degree of parallelism and it is possible to make efficient use of many processors, [but] the granularity is large enough that no cooperation between the processors is required within the matrix computations" [25]. This situation is where Amdahl's law [26] applies:

$$s = \frac{1}{r_s + \frac{r_p}{n}}, \tag{2.1}$$

where the speedup s is determined by the ratio of the parallel section of code r_p to the serial portion r_s, in a system containing n parallel processors. For large n, the proportion of sequential code limits the overall speedup available.

Returning to the pipeline, the greatest potential for parallelisation is in its early stages: preprocessing, segmentation and feature extraction, where the same operations are performed on most pixels. Here the system must handle large volumes of data quickly; several operations are often required for each pixel, of which there can be millions in a single frame. Real-time processing requires doing this dozens of times per second, which leaves only a few nanoseconds to process a single pixel [27]. Moving from the problem domain to the hardware domain, in this section we

Table 2.1: Summarised comparison of data processing architectures (compared to a reference x86).

	FPGA	**GPU**	**X86 SSE**	**Multicore CPU**
Power Consumption	low	high	medium	medium
Clock Speed	low	medium	high	high
Ease of Programming	hard	medium	low	low
Speed gain	high	high	medium	medium
Floating-point Precision	arbitrary, fixed	single/ double	single/ double	single/ double

consider various candidate architectures, the structure of each one, methods of programming, and any other relevant information.

The processing architectures themselves can be arranged using Flynn's taxonomy, which categorises systems into the groups below [28].

SISD Single instruction single data: normal single-core processors, *e.g.* a single core of an x86 chip.

SIMD Single instruction multiple data: Flynn puts systems which express parallelism both temporally (via pipelining) and spatially (via multiple discrete processing elements) in this category. This includes x86 Streaming SIMD Extensions (SSE) vectorisation, GPUs and FPGAs.

MISD Multiple instruction streams working on a single data stream.

MIMD Multiple instruction multiple data: independent multiprocessor systems with some level of shared memory *e.g.* multicore processor systems.

This is summarised in Table 2.1. We now consider each architecture in that Table in detail.

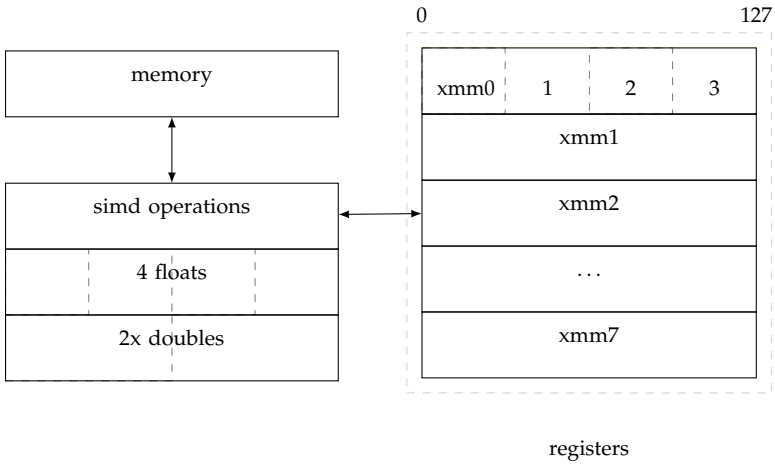


Figure 2.2: SIMD register structure in modern x86 processors. Eight 128-bit registers (right) can be used by the vector processing unit (left) for packed fixed- and floating-point operations.

2.1.2 Methods for CPU Acceleration

Intel and derivative x86 processors provide a SIMD vectorisation unit which works on 128 bits of data (see Figure 2.2). For *e.g.* single-precision floating point calculations, this can offer an up to 4× speedup in arithmetic and logic operations. This is an example of SIMD parallelism and does not require much hardware knowledge to apply; in certain circumstances, certain compilers can automatically vectorise code to make best use of this hardware.

Multithreading can also be used to spread work out over multiple cores and hide processing stalls while waiting for memory or I/O operations to complete, although in general this speedup is limited as only a few cores are available to share the work onto.

2.1.3 Graphics Processing Units

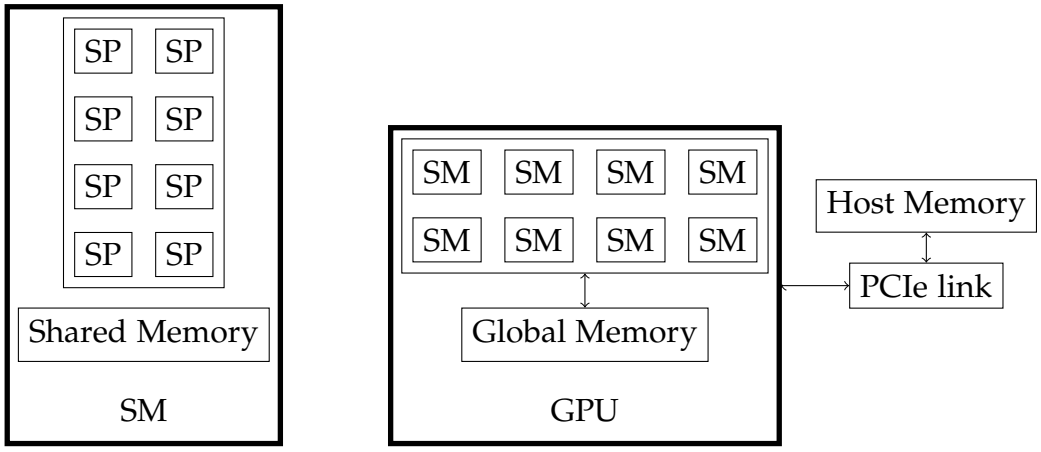
The General-Purpose Graphics Processing Unit (GPGPU or GPU) grew out of the increasing computational power available in consumer graphics cards in the mid-2000s, along with changes in the way these cards could be programmed. They have become very prevalent in the area of high-performance computing, so much so that the current Top 500 list of supercomputers contains 39 systems which are

CUDA-accelerated[1]. Early literature on GPU computing, such as a review by Owens *et al*. [29] in 2007, framed all processing operations in computer graphics terms, such as vertex buffers, fragment processors and texture memory, and relied on custom languages such as `Cg` and `Brook`. In their review the following year [30], the same authors noted that "One of the historical difficulties in programming GPGPU applications has been that despite their general-purpose tasks having nothing to do with graphics, the applications still had to be programmed using their graphics Application Programming Interfaces (APIs)". GPU-accelerated research work on certain applications was done at this point (for example on particle filtering [31]), but problems such as the floating-point calculations not conforming to the published IEEE standard were still prevalent [32].

That changed with the advent of NVIDIA's CUDA[2] and the Khronos Group's cross-platform OpenCL[3], two general-purpose C-based languages designed to expose the underlying parallelism in GPUs. Both function on the basis of *kernels*, processing functions applied to *streams* of data. As CUDA was the language used in this work, we focus on that; the extensions to CUDA beyond standard C mostly relate to choosing which architecture to run a kernel on (`host` or `device`), and arrangements for partitioning and accessing data between processing elements. Rather than using one of Flynn's taxonomy entries [28] to describe their architecture, NVIDIA describe CUDA as Single Instruction Multiple Thread, similar to SIMD.

An overview of CUDA architecture is shown in Figure 2.3; multiple Stream Processors (SPs), each with their own arithmetic and logic unit, make up a Streaming Multiprocessor. Within a Streaming Multiprocessor (SM), each SP can share data with its neighbours using a small amount of shared memory, very close to the SM and hence fast to access. Multiple SMs are arranged on chip, with each SP also being able to access slightly slower global memory (on the same board as the GPU) and, with even more latency, the host PC's main memory (Figure 2.3b). This memory hierarchy also has two levels of caching (not shown), which is managed automatically, and from the point of view of the programmer, the same mechanism is used to access the various types of memory (shared, texture, global, and host). Each SM is scheduled to run multiple groups of processing threads simultaneously; the central idea behind this architecture is that context switching between threads on a SM is very fast, and

[1]Details at `http://www.top500.org/lists/2013/06/highlights/`
[2]Available from `https://developer.nvidia.com/what-cuda`
[3]Available from `http://www.khronos.org/opencl/`



(a) Structure and memory interface of a streaming multiprocessor in a CUDA GPU.

(b) Processor arrangement and memory hierarchy in a CUDA GPU.

Figure 2.3: CUDA Architecture: (a) multiple stream processors (SP) make up a streaming multiprocessor (SM) and have access to a small, fast shared memory region. (b) SMs are arranged within a GPU and can access global device and host memory.

low cost, so thousands of threads can be queued for execution at once across a card. A group of 32 threads executed on an SM is known as a *warp*, and in the latest generation of chips, up to 32 warps can be queued at once. Thus, the inefficiencies involved in multiple levels of memory access will be hidden, because while one warp waiting for data access is stalled, another which requires processing can be run in its place. Despite this technique for *latency hiding*, memory accesses are still slower than processing operations, and CUDA cards obtain their best performance when performing lots of operations on a limited amount of data, *i.e.* maximising the ratio of computations to data transfers. This is a very brief overview of the CUDA architecture, focusing on the main benefits for general-purpose computing: a comprehensive description is given in [10].

Application to Image Processing

GPUs have now become mainstream in accelerating a wide variety of signal processing applications. There are numerous utilities available to help this process, such as specific linear algebra (`cuBLAS`) and fast Fourier transform (`cuFFT`) libraries and image and signal processing primitives (`NPP`). The most well-known in the vision

community is probably OpenCV[4], a general image processing library in which a large number of algorithms are now GPU-accelerated. This includes algorithms for segmentation [33, 34], Viola and Jones' face detection work [19, 35, 36] and medical imaging applications [37], many of which have now been incorporated into OpenCV as summarised in [38]. A general theme among these publications is that some level of knowledge of the hardware is required to gain a speedup.

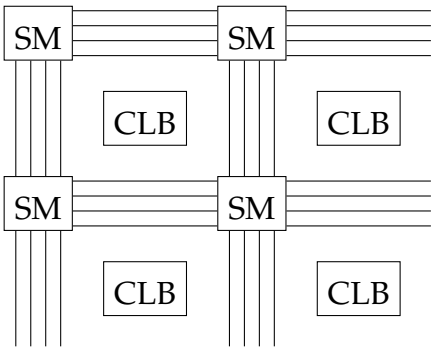
Mobile devices which are CUDA or OpenCL-capable have increased the potential for deployment of algorithms such as these on handheld platforms; accelerated SIFT (Scale Invariant Feature Transform) for descriptor generation by Wang *et al*. [39] is one recent example of this.

2.1.4 Field-Programmable Gate Arrays

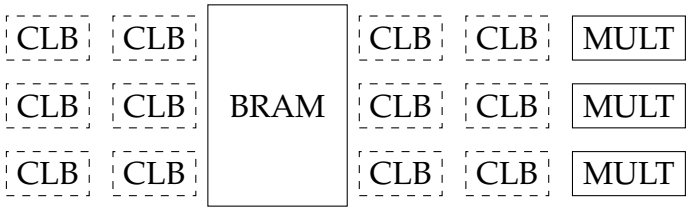
One of the alternatives to mapping large computations to a fixed hardware architecture is to adapt that hardware to the processing required – hence the appearance of FPGAs. The concept of a reconfigurable parallel processing system was first described in the 1960s by Estrin *et al*. [40], and is similar in form to modern Xilinx and Altera devices. As Xilinx FPGAs hold around 50% of the market share[5], and Xilinx devices were used within Thales, we concentrate on Xilinx devices here. However, everything discussed in this section is true for alternatives such as Altera as well.

The structure of a modern FPGA is shown in Figure 2.4. Processing is done by programming Combinatorial Logic Blocks (CLBs), as shown in Figure 2.4a, to perform application-specific logic functions. These CLBs contain programmable look-up tables followed by a storage element (flip-flops). They are connected by programmable switch matrices. Modern heterogeneous devices also contain commonly-used elements such as blockRAMs and embedded multiplier-accumulators (DSP48s in Xilinx terminology), allowing the CLBs to be used for other operations. All programmable elements within the device are configured at bootup time by pushing a *configuration bitstream* through a set of configuration registers on the chip. FPGAs may also contain specialised high-speed transceiver blocks for communications, and allow general-purpose processors to be instantiated on the fabric, whether designed for close integration with the chip (such as a Microblaze [41]) or not [42].

[4] http://opencv.itseez.com
[5] See http://www.xilinx.com/about/company-overview/



(a) Combinatorial logic blocks (containing look-up tables and storage elements) are connected by a configurable switch matrix.



(b) Heterogeneous FPGA containing fabric, blockRAM and embedded multipliers.

Figure 2.4: FPGA Architecture: (a) CLBs are connected by switch matrices. (b) Other components such as memory and embedded multipliers can also be incorporated on-chip.

Programming

An overview of the whole FPGA programming process is given by Bacon *et al*. [43]. Programming an FPGA, particularly when starting with an existing signal processing algorithm, has been described as "very time consuming" by Bailey [44]. This is especially true when considering traditional methods of capturing designs at Register Transfer Level (RTL), using Verilog or VHDL; this step has also been described by Johnston *et al*. as "difficult and cumbersome for large and complex algorithms" [27]. The gulf between a high-level algorithm description as described by MATLAB code, and one written in RTL is quite large, especially if any changes must be made to the original design. This has led to multiple methods for programming FPGAs from a high-level language or model (model-based design). Zoss *et al*. [45] compare various extensions to MATLAB which allow production of bitstreams from a Simulink model. They note that manufacturer tools (*e.g*. Xilinx System

Generator) can have closer integration with the hardware (especially hard-wired multiplier blocks) than competing alternatives, such as the Mathworks' HDL Coder for Simulink. However, they also state that there is scope for expansion in the area of automated or guided parameter selection for various design elements, a topic explored further in Section 2.4. For more details of our use of model-based design in this project see Section 3.4.

A high-level alternative to either RTL design entry or model-based design is to describe the original algorithm in a dialect of C, and many such languages are available (Handel-C, Catapult C, arguably System-C, even, to an extent, CUDA [46]). Two papers by Edwards provide a good overview of this [47, 48], but, as noted in the second, C does not have an explicit mechanism for controlling timing and hence specifying any exploitable concurrency [48].

Finally, an interesting conclusion to this latter line of thought is that OpenCL may potentially be used in FPGA designs [49]. In this work, an OpenCV algorithm is accelerated on the programmable portions of a Xilinx Zynq chip, using OpenCL for design entry and compiled by the Xilinx High-Level Synthesis tools. Such work is still in the early stages and requires lots of parametrising and use of `#pragma` instructions to the synthesis tools, however.

Application to Image Processing

A multitude of image processing applications have now been accelerated with FPGAs. However, unlike GPUs and the extensions to OpenCV, these have not been gathered together into a library [50], so any such speedup tends to be application-specific. Similar complaints emerge in other domains: Jones *et al*. note that "there is little open-source, portable firmware for FPGA [high-productivity computing systems]" [51]. We describe existing parallelised implementations of the algorithms we use in Section 2.2, but here we briefly note the wide variety of applications which have been accelerated in this way (feature detection [52], sky segmentation [53] and object detection [54]). The latter example differs from the former in that the amount of processing done may vary dynamically with scene content – in other words, what proportion of windows are still present after a given number of stages of an Adaboost classifier (the theoretical basis for this is again provided by Viola and Jones [19]). This problem is tackled by allocating the first ten classifier stages

to hardware, and running subsequent stages in software, processed by a hard CPU on the same FPGA. This is an interesting example of design partitioning across platforms applied to an image processing algorithm, albeit one where this decision is made at design time; we explore this further in Section 2.4.

Reconfiguration

An important consideration with FPGAs is the potential for reprogramming the device while it is running; standard methods require a global reset of the device post-program, while in comparison GPUs can execute kernel launches in around $3 - 7\mu s$ [13]. Both Xilinx and Altera now offer some form of support for *Partial Dynamic Reconfiguration*, *i.e.* reprogramming portions of the chip while it is running) [55, 56]. This technology can be used in a variety of domains, such as autonomous agents for processing of network data [57], and in the domain of imaging again (*e.g.* for executing each separate stage of a fingerprint scanning process on a single, small FPGA, reducing the required resources [58]. This can be considered another attempt to approach the Size, Weight and Power problem we defined in Chapter 1 as being central to the relevant question this thesis addresses. It can also be used for implementing hardware "threads", as shown in [59], although the authors note the considerable latency associated with every reconfiguration. This cost must be included when attempting dynamic reconfiguration, as explored further in Section 2.4 and in work by Quinn [60]. Xilinx described their enhancements to the internal programming process in [61] which enabled partial reconfiguration, but as of late 2013 the tools to do this require a special licence, and require considerable expertise to set up.

Happe *et al*. demonstrate real-time, dynamic reassignment of tasks between hardware and software regions on an FPGA containing a soft processor and dynamically reconfigurable regions [62]. Their paper documents real-time video object tracking using sequential Monte Carlo methods. Unlike the work described in this thesis, their self-adaptive system does not have changing constraints and only considers time and (indirectly) FPGA resource use as performance criteria.

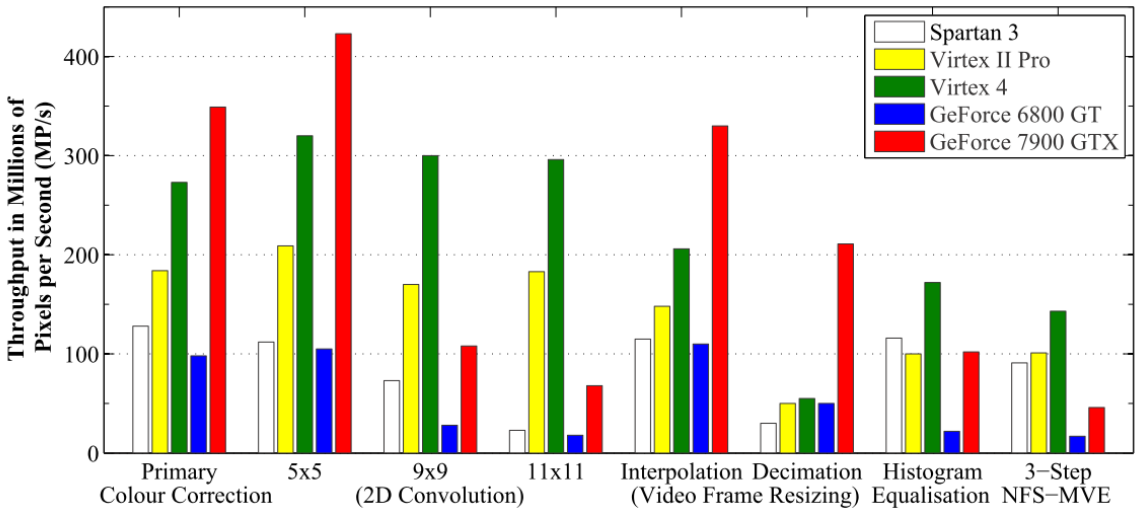


Figure 2.5: Throughput of image processing operations on 3 FPGAs and 2 GPUs compared. Image from [67].

2.1.5 Comparison and Selection of FPGA vs. GPU

As discussed above, various algorithms in various domains have been implemented on both FPGA and GPU. There have been many efforts made to approach comparisons between this platform in a structured manner, and to answer the question "given a particular algorithm, is it better to implement this on an FPGA or GPU?". The answers to these have either taken the form of empirical results from implementing algorithms on both platforms [63, 23, 64, 65, 66] or a combination of empirical results and theory [51]. This work can also be grouped according to applications, with some of the papers above applying specifically to image processing.

A lower-level analysis of this area was also the main subject of Cope's work [67]. Early work by this author and others in 2005 [68] argued that GPUs had not made FPGAs redundant for video processing, especially for applications with large numbers of memory accesses (*e.g.* large $2-D$ convolutions). In later work [64], five image enhancement and processing algorithms are analysed and the relationship between instruction set characteristics is measured. This is reproduced in Figure 2.5; note the dramatic falloff in GPU performance for $2-D$ convolution at higher kernel sizes.

Both platforms are found to provide substantial speedup in all cases. However, some results from this are worth highlighting: Cope notes the higher power consumption

of the GPU, alongside its ability to bring a higher n to bear in Amdahl's law (2.1). When fixed-point FPGA performance is compared to the potential for floating-point computation from a GPU, the GPU has a higher computational density factor (*i.e.* to achieve equivalent floating-point performance to the GPU, their FPGA would need to have 12 times the area). In common with other studies, implementation time is not considered. As noted in §2.1.3, this work was done before the advent of CUDA and OpenCL, so this comparison provides a useful overview, but certain conclusions may now be out of date.

A limited amount of work has also been done in three-way comparisons which include CPU as a possible platform, such as Grozea *et al*. who used text searching in network processing as an application [69]. Their conclusions match closely with the tradeoffs we present in Table 2.1; *i.e.* FPGA is "most flexible but least accessible", CPU is "easiest to approach" but "might sometimes be too slow", and GPU is "difficult to debug and requiring data transfers which increase the latency". However, since then GPU debugging technology has improved and this step is less cumbersome.

A different research direction to the one discussed above has also been pursued by other researchers; that of combining — rather than contrasting — FPGA and GPU computations in a single system. Bauer *et al*.'s work [70, 71] is relevant here; they manually partition the HOG algorithm between their FPGA and GPU, but do not discuss the rationale behind these choices. See §2.2.3 for a fuller discussion. More comprehensive work by Bittner *et al*. [72, 14] focuses on low-level FPGA and GPU interaction and methods for data transfer, including discussion of methods for reducing the number of times data is copied unnecessarily during transfers between both accelerators over PCI Express (PCIE); see Figure 2.6. They also described a 2.2× reduction in data transfer time by removing the standard, intermediate step of copying to host memory. They note that transfer speeds are asymmetrical, due to the quality of the implementation of the Direct Memory Access (DMA) controller on the FPGA; GPU-initiated transfers from GPU to FPGA are considerably faster than the other direction (1.6Gigabytes per second (GB/S) against 0.51GB/S). For a discussion of how these factors relate to this work see Chapter 4 of this thesis.

There is no overall clear picture from these many comparisons; FPGA and GPU can often be applied to the same application, and achieve a substantial speedup. In such circumstances factors other than raw performance — such as power consumption

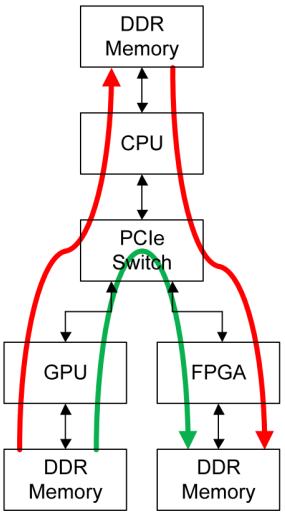


Figure 2.6: Reduced transfer times between FPGA and GPU are achievable by initiating a DMA transfer directly between the two device memories (green) rather than an intermediate copy to host memory (red). Image from [14].

or implementation time — should also be considered; however, this is often missing from published work (for example, see Bauer *et al*. [70]).

2.1.6 Alternative Architectures

Various other architectures can be considered as possible candidates, including the Cell Broadband Engine [24], and Intel's multicore compute and graphics platform Larrabee, before development of the latter was cancelled. (Intel's latest multicore accelerator Xeon Phi is a successor to Larrabee [73]). A comprehensive review of all these architectures by Brodtkorb *et al*. is given in [24]; here the authors note that one architecture is unlikely to be suitable as the desired accelerator for all applications — this is a reasonable conclusion as each platform exists at a different point in design space.

2.2 Parallelisable Detection Algorithms

Most image processing algorithms are inherently parallel and often belong to the class of dense linear algebra operations, as discussed in §2.1.1. We concentrate here on algorithms which can be deployed in some manner to suit our goals as stated in

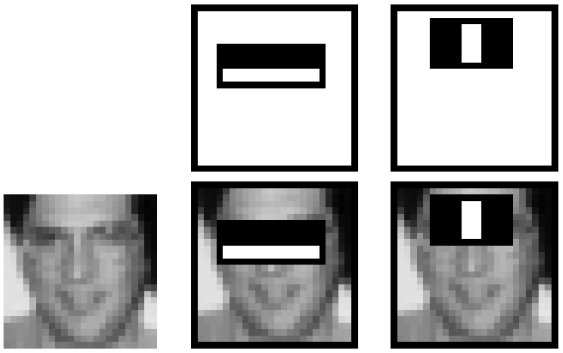


Figure 2.7: First and second Haar features in Adaboost classifier for face detection. Image from [19].

Chapter 1: that of scene analysis. This again follows the pipeline of Figure 2.1, so we concentrate specifically on the classification stage of this pipeline. At this point we also restrict discussion of algorithms to those which process visible-spectrum data.

Work on object detectors either concerns classification performance on multiple classes of objects (such as in the PASCAL challenge [74]), or the development and improvement of detectors for a specific object class. The most popular objects investigated are either human bodies or faces, due to their relevance in many fields, from human-machine interaction to road-safety and surveillance and military applications, on both desktop and mobile platforms. As this is a fast-moving field, we provide an overview of the history of human detection here.

Viola and Jones' fast detector in 2002 [19] applied Haar wavelets (see Figure 2.7) to object detection. This was combined with the development of an integral image, allowing precalculation and then fast accessing of the sum of the intensity contained within any rectangle in the image within constant time (important for evaluating Haar features). This was paired with a *cascade classifier*, allowing construction of a strong classifier using a succession of weak ones. The classifier cascade was arranged so that each stage rejected non-faces early on, thus reducing the number of classification operations needed; the efficiency of this approach has been demonstrated by various hardware implementations (see *e.g.* [75] for FPGA or [76] on GPU.)

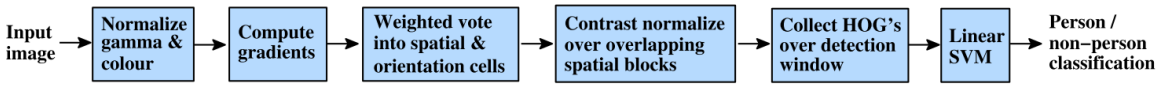


Figure 2.8: HOG algorithm pipeline. Image from [77].

2.2.1 Algorithms for Pedestrian Detection

We focus now on detection of humans in images and videos, a major area of object detection research. A significant improvement was made in this area by Dalal and Triggs' description of their Histogram of Oriented Gradients detector in 2005 [77]. Their processing pipeline is shown in Figure 2.8. We describe this algorithm in detail in Chapter 5 in order to reimplement it, but here we will only note that it consists of sliding a detection window across an image and obtaining a confidence that a person is present in that sub-window. This process is repeated at multiple scales, then non-maximal suppression is performed to group detections. Detections are made as shown in Figure 2.8 and Figure 2.9; per-pixel gradients are binned into orientation histograms and gathered over dense, overlapping local regions to form blocks. A sub-window full of blocks is then classified against a model trained on a dataset of pedestrian images (developed by Dalal and known as the INRIA dataset) using a Support Vector Machine (SVM). Classifiers are discussed in §2.2.2.

Both stages here (histogram generation and SVM classification) have localised, regular memory access, are relatively computationally dense, and are not iterative, and so we expect that they will be strongly suited to a parallelised implementation. Dalal also extended the detector to the other object classes (car, motorbike, cat, dog, *etc*.) in the PASCAL dataset, and gave parameters for this in his thesis [78]. Performance on these classes was sufficient for his work to win the 2006 PASCAL challenge.

Further progress in this area was made by Felzenszwalb *et al*. [79], who applied a deformable-parts model to HOG and improved detection accuracy on the INRIA dataset. Similarly, Tu and Perona [80] extended HOG in combination with features selected from other image channels (the LUV space) to form the Integral Channel Features detector. The latter was optimised in 2010 by Dollàr, Belongie and Perona, who recognised that feature generation over multiple dense scales was one of the major effects on the long runtimes affecting existing detectors [81]. Their approach,

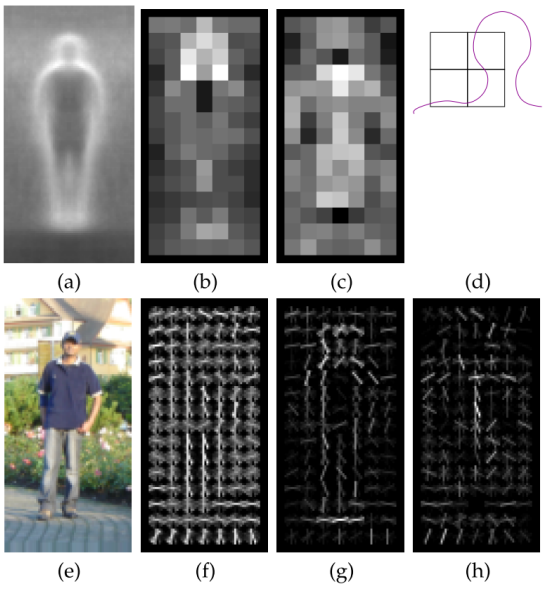
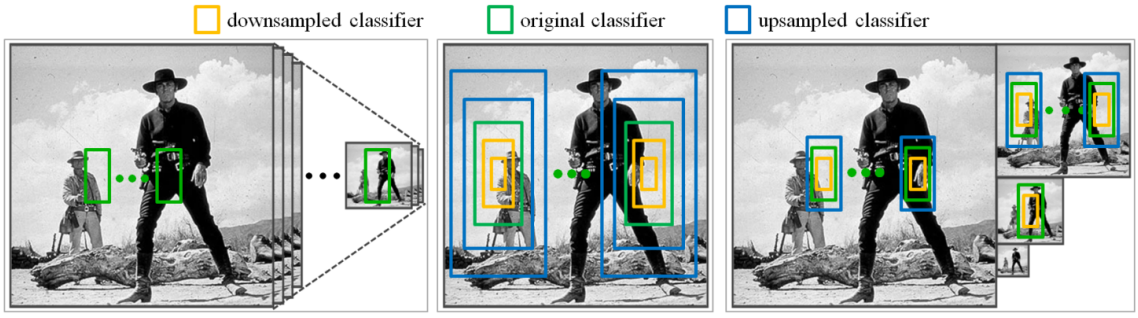


Figure 2.9: Building the HOG processing pipeline. (a) Average gradient over training images. (b) Maximum positive SVM weights in each block; (c) Maximum negative SVM weights in each block; (d) Most relevant blocks (just outside contour); (e) Test image; (f) HOG descriptor for test image; (g)–(h) Positive and negative weights for test image. Image and labels from [78].

involving a sparsely sampled image pyramid with multiscale classification, is shown in Figure 2.10.

Dollàr *et al*. conducted a comprehensive review in this area in 2011 [82]. This evaluated 16 existing pedestrian detectors in terms of their performance on the INRIA and Caltech datasets. They standardised evaluation methodology (*e.g*. MATLAB code accompanying the paper provided a robust method for non-maximal suppression), and argued for a change in the standard evaluation metric (see below). They also identified various future research directions to pursue, noting that "Performance is abysmal at far scales (under 30 pixels [in height]) and under heavy occlusion (over 35 percent occluded)". In addition, INRIA remained the dataset of choice for training most detectors [82, 79, 81]. A sample HOG detection on an INRIA image is in Figure 2.11a, and ground truth for a still from a Caltech video is in Figure 2.11b.

Since that review paper, new detectors have continued to be produced, particularly by Benenson or Dollàr. The `VeryFast` detector, first described in 2012 by Benenson *et al*. [83], achieved 50Hz detections across whole images by running on GPU and



(a) dense image pyramid (b) classifier pyramid (c) proposed hybrid approach

Figure 2.10: The Fastest Pedestrian Detector in the West moves away from the traditional approach of computing features and classifying on a densely sampled image pyramid (a). However, only scaling the classifier (b) does not work because the desired features are scale invariant. The hybrid approach (c) uses a classifier pyramid within each octave, but only samples and scales the image once per octave, considerably reducing evaluation time. Image from [81].

(a) HOG detections on INRIA dataset image. (b) Ground truthed bounding boxes on Caltech dataset image.

Figure 2.11: INRIA and Caltech dataset sample images. Image (a) from [78], (b) from [82].

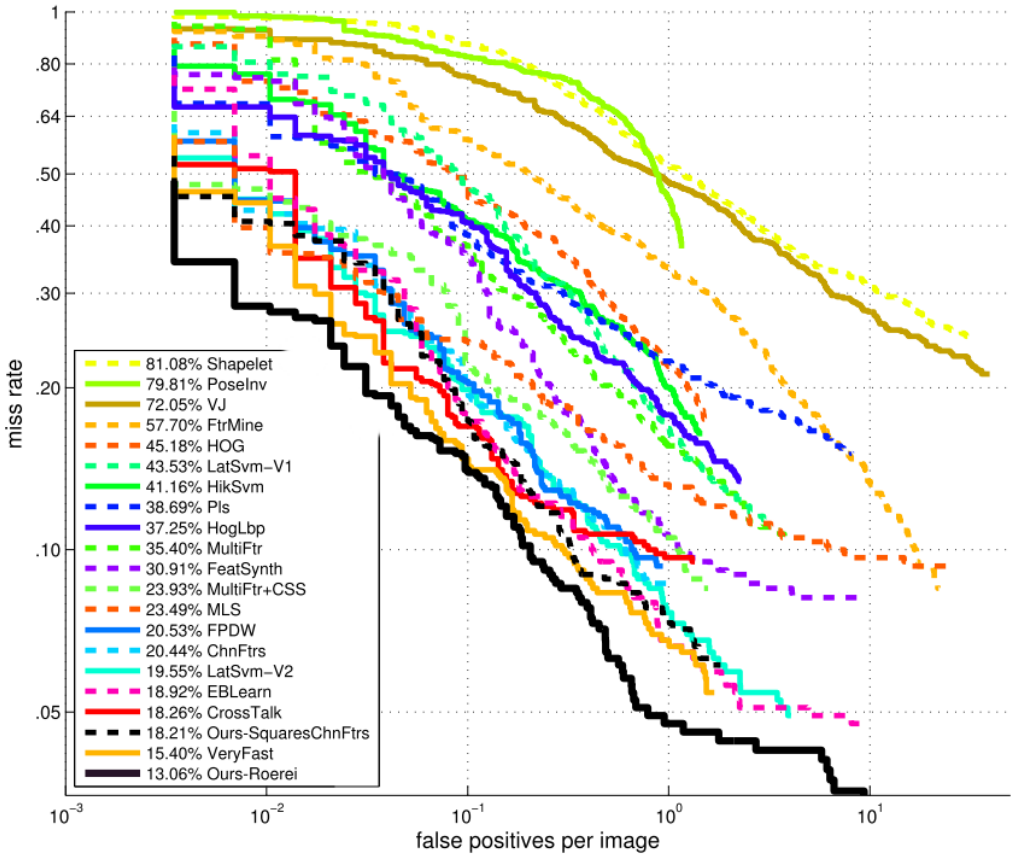


Figure 2.12: State-of-the-Art Pedestrian Detection Performance still requires improvement in many situations. INRIA dataset performance is shown in a Detection Error Tradeoff curve from [84].

only evaluating an image at a single scale (*i.e.* reversing the optimisations of [81]), and using stump classifiers and Adaboost instead of SVMs for classification. This rate increased to 135Hz by using stereo images and only classifying objects on the ground plane.

Evaluation Metrics

Figure 2.12 is a detection error tradeoff curve. These are often used instead of more traditional Receiver Operating Characteristic (ROC) curves when comparing detector performance, because the curves they present are close to linear, while ROC curves often "bunch" into a corner [85]. False positives are on the x-axis and (1−true positives) are on the y-axis. Dalal and Triggs' results and others' subsequent work used False Positives per Window (FPPW) as the $x-$axis measurement. This considers per-window performance on its own and does not take into account techniques for

grouping and non-maximal suppression of nearby and overlapping detections over many scales. Dollàr *et al*. pushed for a movement away from the FPPW performance metric to a False Positives per Image (FPPI) one, arguing that (i) it is the performance of the overall pedestrian detection algorithm which is important for comparison and deployment, and (ii) contrary to expectations, per-window detector rankings do not match per-image rankings [82]. Throughout this work, we use FPPI. However, many implementations of HOG still use FPPW [86, 87] so we compare performance against them when required.

State of the Art

The current state-of-the-art detector is by Benenson [84]. By returning to [80] and systematically evaluating feature and normalisation choices (*i.e*. using all possible squares within a classification subwindow, rather than Dalal and Triggs' manually-selected grid of fixed-size blocks), detection performance has increased still further. It is far from a solved problem, as Figure 2.12 shows; almost one-fifth of pedestrians are still missed for every false positive in 10 images. In addition, this graph shows results evaluated on the INRIA dataset, and does not include occluded pedestrians. The Caltech dataset is considered to be somewhat harder, and includes pedestrians with varying levels of occlusion. As this figure shows, HOG, although no longer considered state-of-the-art, is still commonly seen as a measure of standard or baseline performance and is "surprisingly competitive" [84]. This paper also notes that training detectors on INRIA rather than on a larger dataset such as Caltech still produces best results.

To reinforce the extent to which HOG and its derivatives generalise, we consider two further applications, starting with detection of vehicle orientation. Rybski *et al*. [88] evaluate HOG to perform two separate tasks for vehicle detection. First, given a vehicle detection obtained from another source (LiDAR in this case), they compute coarse vehicle orientation placed into one of eight directional bins, by running eight one-vs-all SVMs across the image region of interest. Secondly, they test the ability of a single classifier to detect vehicle presence. This test was successful, but as a FPPI curve is not provided, their results are not directly comparable to the results above. The authors note that, as cars viewed from different angles look much more diverse than humans from different angles (compare side-on vs. head-on), car detection is a harder problem than human detection. They also convert the classifier confidence

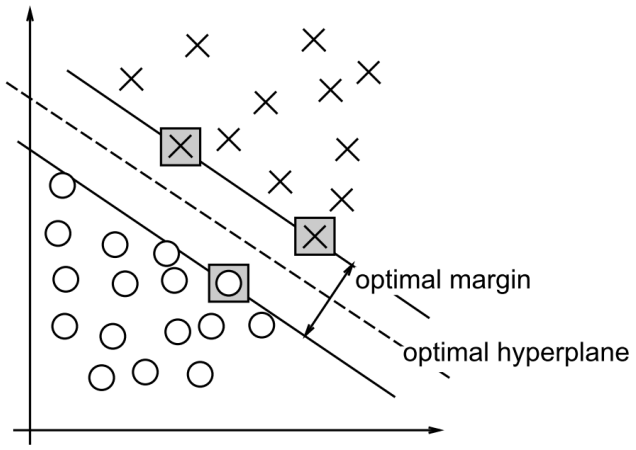


Figure 2.13: Support Vectors in a 2-D problem. The support vectors (grey squares) are placed on the optimal margin, which defines the largest separation between the two classes. Image from [89].

output into a probability value, allowing for selection of the orientation with the highest posterior probability. This uses Platt's method as discussed in §2.2.2. The second non-human detection application involves road signs; taking the `VeryFast` detector [83], Mathias *et al*. [5] apply it to road sign detection, achieving $> 95\%$ detection accuracy with little algorithmic modification.

2.2.2 Classification Methods: Support Vector Machines

SVMs are a common machine learning tool for classification; for a thorough introduction to their use in imaging and the broader discipline of signal processing see Burges [90]. They operate by learning *support vectors* which maximise the margin in some feature space between the two classes of data being classified. An example is shown in Figure 2.13.

For an input vector $\mathbf{x}$ representing a data point (*e.g*. gradient histograms in a sliding window), the output $y = \{+1, -1\}$ from a SVM acting as a binary classifier will be

$$y = \begin{cases} +1, & f(\mathbf{x}, \mathbf{w}) > 0, \\ -1, & \text{otherwise}, \end{cases} \tag{2.2}$$

where $\mathbf{w}$ is a set of n_{sv} learned weights, each of length n, and

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{n_{sv}} (\alpha_i \cdot y_i \cdot \mathbf{K}(\mathbf{w_i}, \mathbf{x})) + b\,. \tag{2.3}$$

$\mathbf{y}, \alpha$ and b are learned parameters. Different kernel functions $\mathbf{K}$ can be used here. A common one is the Radial Basis Function (RBF) kernel,

$$\mathbf{K(x,w)} = \exp\left(-\gamma \|x - w\|^2\right). \tag{2.4}$$

However, this is computationally expensive for large or dense vectors, so a linear classifier with a single vector of n weights is often used instead, giving

$$f(\mathbf{x}, \mathbf{w}) = \sum_{j=1}^{n} (x_j \cdot w_j) + b\,. \tag{2.5}$$

The linear kernel involves evaluation of a dense dot product. This is the method used in HOG [78]. The drawback of the SVM approach is that the entire calculation over n or n_{sv} needs evaluated before we can confirm the classification of x.

An alternative to evaluating the entire calculation is to use a reduced set of support vectors, as described by Burges and Scholkopf [91]. Here, a set of n_{rs} vectors, denoted $\mathbf{w}'$, are used, where $n_{rs} \ll n_{sv}$. The authors choose n_{rs} by limiting the allowable error between $f(\mathbf{x}, \mathbf{w})$ and $f(\mathbf{x}, \mathbf{w}')$ and note that this technique can reduce the number of vectors evaluated on each image patch by two orders of magnitude. This method is then applied to face detection by Rohmdani *et al*. [92]. Here, a cascaded SVM is built. Early cascade stages are trained to reject non-faces quickly. As soon as the score from $f(\mathbf{x}, \mathbf{w}')$ falls below a threshold, evaluation of that image patch is terminated. This results in very few windows which could possibly be faces remaining at the end of the cascade. These can then be evaluated by a full-length set of support vectors trained for accurate detections, while an increase in speed is gained.

In all cases, the SVM training process consists of learning $\mathbf{w}$ to maximise the margin between the classes. Thus, all the learning of the characteristics of the person (or other object that the SVM is trained to detect) is done at the training stage, and $\mathbf{w}$ is a "model" of what the object looks like. The testing or operational stage is

essentially a comparison of how close a given image subwindow is to this model. This evaluation, described by equations 2.4 and 2.5, involves a large number of matrix multiplications and additions (dense linear algebra from [22]). These map well to FPGA or GPU and allow real-time operation (as described in the following section) and it is this stage that we aim to accelerate. The support vector training stage can also be accelerated by performing many kernel evaluations in parallel, and again by performing a parallel "reduce" on GPU to select the best candidate during every training iteration [65].

Use of Platt's method [93] to convert SVM outputs into a posterior probability $p(C_i|s)$ is also possible: given a SVM score $s = f(\mathbf{x})$, what is the probability of the vehicle being in orientation bin C_i? Platt's method involves fitting parameters a and b to the data, in the form of the SVM output s:

$$p(C_i|D) = \frac{1}{(1 + exp(a \times s + b))} . \tag{2.6}$$

However, this method is empirical and has been criticised as such [94, 95]. It also leads to unduly confident classifications of test points a long distance from the margin.

2.2.3 HOG Implementations

Having reviewed pedestrian detection and the SVM classifier algorithms, we now consider existing implementations of HOG, both to provide an understanding of the research field, and to allow for comparison to our own work in Chapter 5. HOG has been implemented for GPU by moving all operations except non-maximal suppression onto GPU [96]. As Figure 2.14 shows, histogram generation takes up most processing time (compare this to the algorithm stages given in Figure 2.8). A similar implementation is now present in `OpenCV`. Various SIMD-accelerated HOG routines are also present in Dollàr's toolkit [97].

Multiple FPGA implementations are also documented in the literature. Kadota *et al*. [86] perform HOG feature extraction in FPGA, then classify the results on a microprocessor. They achieve real-time feature extraction by using ten instances of their architecture in parallel. Martelli *et al*. [87] perform FPGA-based pedestrian detection using covariance features, and Hiromoto *et al*. [98] describe a similar

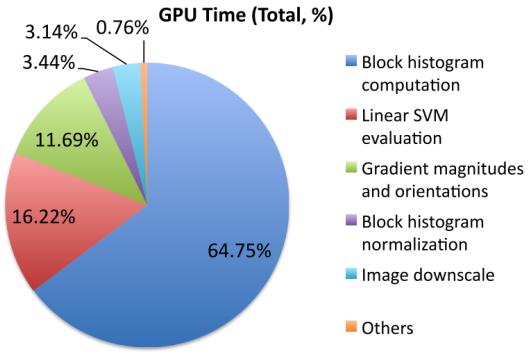


Figure 2.14: Time spent on each algorithm stage in a HOG GPU implementation. Image from [96].

system using co-occurrence HOG. Each set of authors describe their processing pipeline and techniques used to modify the original algorithm, and most reach real-time performance on small image sizes, with limited accuracy. See Section 5.5.4 for a comparison of results against our own. Similarly, Cao and Deng [99] perform HOG-based road sign detection on FPGA; this paper is helpful for details of low-level implementations.

An interesting hybrid system is described by Bauer *et al*. [70, 71]; their processing pipeline is shown in Figure 2.15. Histograms generated on FPGA based on visual data are transferred to CPU. Regions of interest corresponding to motion regions found in both visual and infrared data are then downselected (*i.e*. the number of windows selected for SVM processing are greatly reduced). This allows candidate regions which may contain pedestrians to be transferred to the GPU, where a RBF kernel-SVM is used to process them. RBF SVMs allow better detection performance (around 3% FPPW gain on INRIA [78]) at the cost of greatly increased runtime due to the large matrix multiplications needed. By capping the number of candidate windows at 1000, performance of around 10 fps is achieved. Bauer *et al*. do not provide the justification for choosing this particular arrangement of devices in design space, or discuss the possibility of arranging processing differently (*e.g*. doing all work on FPGA), especially considering the overheads relating to inter-processor data transfer. Detection performance improves on the linear-classifier HOG used in [77], but is worse than the kernel classifier version in Dalal's paper.

In a 2013 paper, Hahnle *et al*. [100] describe a high-performing implementation of the entire HOG algorithm on FPGA. This work is notable in that a HD image

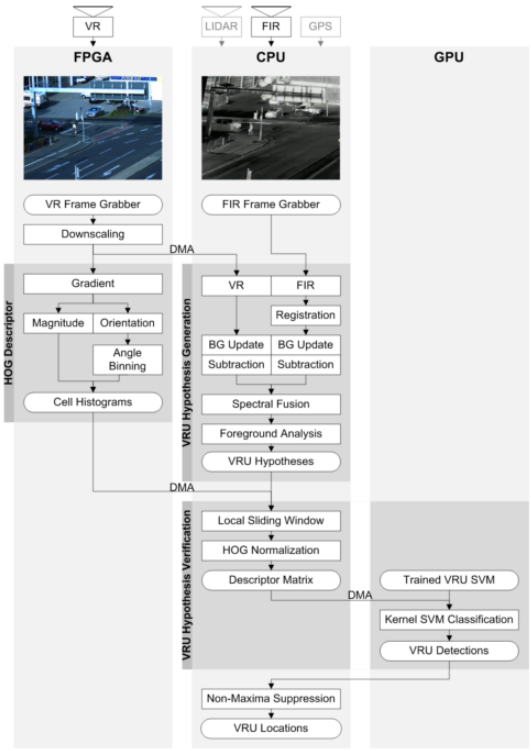


Figure 2.15: Processing pipeline in a hybrid system. Image from [70].

is used, requiring a much higher-performance implementation than other related work (where resolutions on the order of 640×480 are common). Their pipeline is shown in Figure 2.16 (for histogram generation) and Figure 2.17 (for classification). In common with other published work, the L_1Sqrt instead of the L_2Hys norm is used to reduce implementation complexity at a minor cost in accuracy. Derivations are given in Appendix A. (See also §5.2.2).

Classification performance is demonstrated on the INRIA dataset, and is shown to be similar to the original HOG, if slightly less accurate due to use of fixed point number representation and other optimisations. The authors note the difficulties involved in performing detection at large numbers of scales while working under real-time constraints. Their approach to this is to choose a number of scales to evaluate, then spread evaluation of those scales over three frames. Without details of the scaling factor used, it is difficult to evaluate the effectiveness of this approach (*i.e.* using the standard factor of 1.05 between scales, 44 scales would be required for a 1920×1080 image; only 18 are used), particularly when the FPPW evaluation method is used.

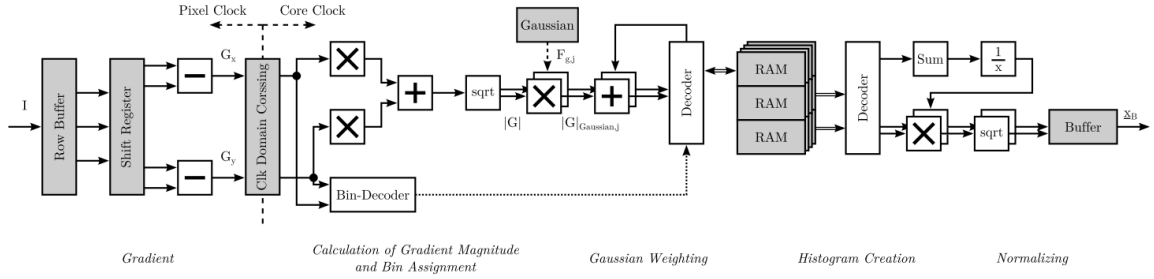


Figure 2.16: Fast FPGA HOG processing pipeline. Histogram generation steps shown. Image from [100].

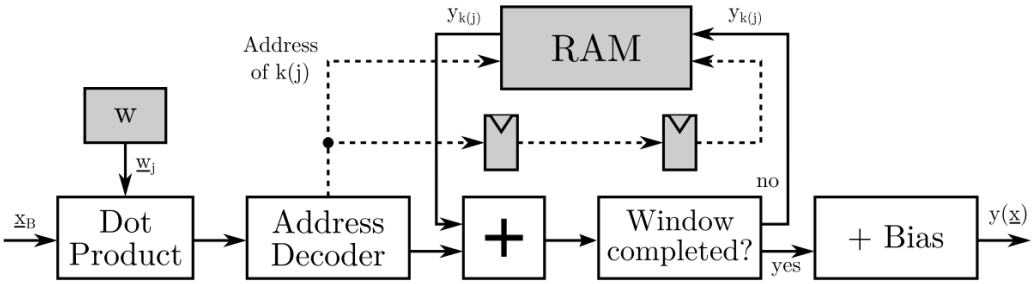


Figure 2.17: Fast FPGA HOG processing pipeline. Classification steps shown. Image from [100].

In conclusion, HOG and its derivatives are still highly relevant for human detection, as evidenced by the fact that HOG derivatives are currently the scientific state of the art in pedestrian detection. As shown by applications in vehicle and traffic sign detection, these generalise extremely well across other object classes. There are also a multitude of parallelised versions documented across multiple architectures, and as shown by the work of Hahnle *et al*. [100] this extends into 2013. All of these factors make it a good candidate for investigation of performance tradeoffs.

2.3 Surveillance Video Analysis and Anomalous Behaviour Detection

Having reviewed methods for generating object detections, we now consider literature on higher-level inferences which can be obtained by deploying these in video surveillance. We start with a discussion of a wide-ranging survey paper by Morris and Trivedi which provides an overview of the field of surveillance video

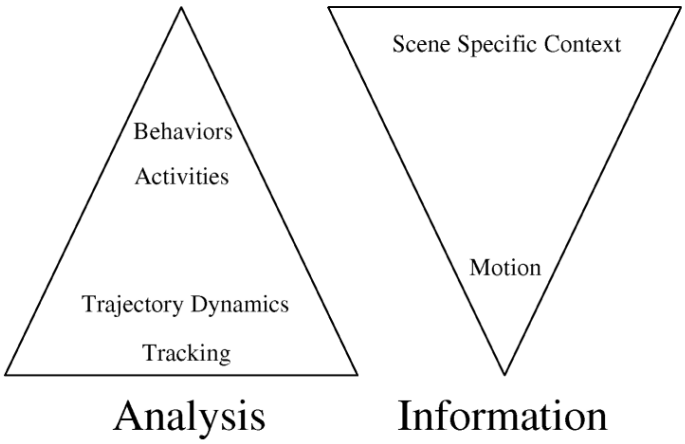


Figure 2.18: Analysis and information hierarchies in surveillance video. Image from [101].

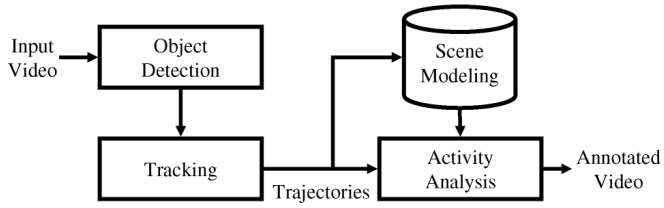


Figure 2.19: A block diagram of surveillance analysis system. Image from [101].

analysis [101]. In it they define the problem of automatic behaviour understanding from video as one of "extraction of relevant visual information, suitable representation ... and interpretation ... for behaviour learning and recognition". They note the tedium of the task, in common with our arguments from Chapter 1 [2, 3].

Their generalised processing model is shown in Figure 2.18, with corresponding block diagram in Figure 2.19; this occupies the higher levels of the machine vision pipeline (Figure 2.1). The model can be defined in terms of Points of Interest (POIs) and Activity Paths (APs). Morris and Trivedi note the challenging scale of the problem, particularly in complex or unstructured scenes. As a simpler motivating example, consider a highly structured scene such as traffic travelling in lanes in a motorway. A representation of APs can be built up; traffic tends to travel in well-defined lanes and at speed. This is not the case for traffic at intersections or urban roads; various vehicle classes (car, bicycle, lorry, pedestrian) are present and can often be stationary for some time, increasing the possibility of tracking failure or generation of incomplete tracks.

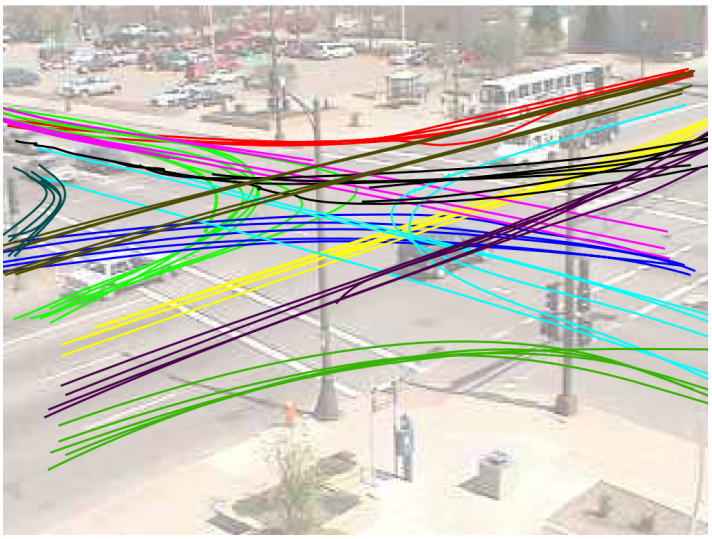


Figure 2.20: Offline filtered traffic analysis produces smooth detections. Image from [103].

This allows object tracking and trajectory generation. Tracking of detections is often done with a Kalman Filter [102], an algorithm for producing smoothed tracks of time-varying signals. This relies on a *prediction* step — estimating the position of an object $\hat{x}_t$ based on knowledge of its previous positions x_{t-1} — followed by a *correction* step, where new measurements z_t are used to update $\hat{x}_t$. The relevant equations are given in Appendix A.

The next stage in trajectory processing is clustering, and the authors note that normally, trajectory lengths must be normalised before any clustering or further processing is done to compare multiple trajectories. This can be done via techniques like zero-padding, interpolation, or smoothing via Kalman Filtering; however, these are "ill-suited for analyzing incomplete trajectories obtained via live tracking". Smoothing tracks and rejecting unsuitable ones via various heuristics ("vehicles are invariably longer than wider", "reject all objects that never move faster than 10km/h", ignore all pedestrians), followed by K-means clustering, produces very clear trajectories for traffic analysis and planning applications, as Figure 2.20 shows [103]. Generating directional histograms from trajectories is also effective [104], as is construction of trajectories using flow vectors of position and velocity data to compare new trajectories to [105].

As we aim to build a real-time causal system, (we wish to analyse and match trajectories to existing ones while they are still in progress), this approach is, un-

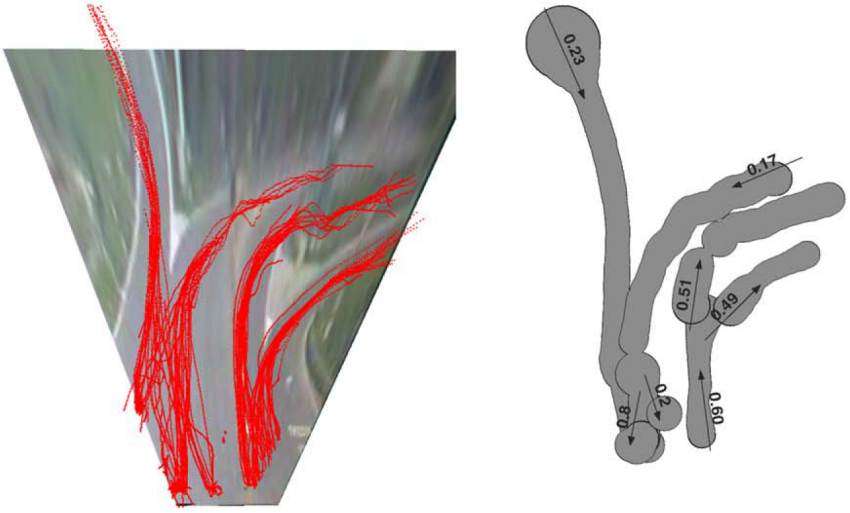


Figure 2.21: A set of clusters depicting object trajectories in a motorway scene. Tracking transits into subtrees generates a probability distribution, as on the right. Image from [106].

fortunately, unsuitable. Work by Piciarelli and Foresti [106] describes a method for online clustering of tracked points into trajectories, where tracks are matched to trajectories using a sliding temporal window which expands with track length. As Morris and Trivedi note, a large training database does not need to be collected before starting and the number of tracks does not need to be known in advance, making this ideal for "long-term time-varying scenes" [101]. This method is used in the anomaly detection system in Chapter 6 for these reasons.

Once a cluster is complete (*i.e.* the object generating it has left the area, or it has been reinforced by several transits through it), it can be used to predict the behaviour of objects as they move through the scene. Clustering into trees is the approach pursued by Piciarelli and Foresti, as shown in Figure 2.21 [106]. Here, trajectories are projected onto a ground plane rather than the image plane to minimise perspective distortion. The planar homography equation used to do this is given in Appendix A. Under their definition "an anomaly is simply defined as an event which happens rarely." The frequency of transits between a tree and leaf node is thus used as a probability distribution, and can flag anomalous events as they appear. An alternative is to use Hidden Markov Models, as in traffic flow analysis by Morris and Trivedi [107]. This is able to detect some anomalous events such as U-turns, and runs in real-time but performance or hardware information is not provided.

Finally, we consider anomalous behaviour detection as the highest stage of the analysis hierarchy (Figure 2.18). This eventually relies on thresholding; *i.e*. take some action if $p(AP|trajectory)$ is below some (possibly cluster-specific) threshold. In our surveillance patrol vehicle example from Chapter 1, this can extend to alerting the operator or changing the power consumption goals of the system, while other options include pointing a pan-tilt-zoom camera at any interesting behaviour to get more information about it [108].

Loy *et al*. provide a three-tiered definition of anomalous actions in video: Category A actions are "visually very different from the training set" (*e.g*. a fire engine running a red light), Category B are ambiguous and rarely appear in the training set, and Category C have only weak visual evidence for an anomalous event happening at all [109]. It can be difficult for humans to detect B and C class events (particularly during a prolonged vigilance task). Loy *et al*. approach the problem of urban surveillance and argue against object- and trajectory-based approaches, noting that large numbers of broken trajectories cause problems for anomaly detection. Instead, they use an approach based on segmentation and cascaded Dynamic Bayesian Networks. However, in common with other published work, a Gaussian mixture model background subtraction algorithm is used to segment out foreground objects [110]. Another approach using online clustering of spatiotemporal volumes describes real-time results and a smaller number of required initialisation frames, but no hardware specification is given [111].

The Future of Anomalous Behaviour Detection

Returning to our original survey article, in 2008 Morris and Trivedi listed the main challenges in this field. They particularly noted the lack of a standardised performance metric for evaluating effectiveness of competing systems, with some using classification accuracy and others reporting anomaly detection rates [101]. This is reiterated by Sivaraman and Trivedi in another survey on behavioural analysis of vehicles in 2013 [112], where they emphasise that the lack of benchmark datasets and metrics is mainly due to the infancy of the research area, and many research papers in this field having different aims. Unlike pedestrian detection with Caltech and INRIA, there is as yet no standard dataset or metrics. Such a dataset would, they point out, require several levels of ground truth, from object detections and tracking,

to behaviour analysis and identification of events. As we discuss in Chapter 6, this is a long way off.

Given that this field is relatively immature, real-time performance has not received much attention. As much of the parallelisation work must be carried out at the earlier (pre-trajectory generation stage) this is, to a degree, expected. As an example, a paper documenting a CPU and GPU system for pedestrian and vehicle detection (*i.e.* detection-only) shows 42 Frames per second (FPS) performance on 640×480 video [113]. A precision/recall curve is given but no details of either training dataset are provided. For a pedestrian detector based on HOG, the omission of even a FPPW curve is surprising.

Parked Vehicle Detection

In Chapter 6, we consider the task of parked vehicle detection as an example of anomalous behaviour extraction. Having sought a real-world problem to apply object-based anomalous behaviour detection to, detection of illegally parked vehicles is an example task for which training and test data is readily available. Here we must again bear in mind Sivaraman and Trivedi's comments about a lack of standard datasets and performance measures [112].

Parked vehicle detection was the subject of a conference session at AVSS in 2007. This used four clips from the i-LIDS dataset. (See Figure 6.2b for an example.) The challenge was to identify the start time when a vehicle parked at the side of the road, and the time it stayed there. We disregard some of the entries to this challenge, as they require extensive human intervention (they manually define the kerbs on both sides of the image as no-parking regions, and run blob-detection on anything stopping there) [114, 115]. The remainder, such as Bevilacqua *et al.*, consider real-time operation on PCs or embedded hardware, and are able to detect all four events in the video sequences and obtain start and stop times within a few seconds of ground truth [116].

Later work by Albiol *et al.* concentrates on detecting stopped vehicles in traffic scenes [117]. They use the full i-LIDS dataset, covering 24 hours of training and test data – much more than the AVSS clips. Similarly to [115] they rely on manual annotation of lane regions in cities and then identify times in which these are

occluded. This technique works well, with precision and recall rates of $> 95\%$ on i-LIDS. They avoid the use of any background subtraction algorithm, relying on lane masking instead, which gives more robust detection of longer-lasting objects. However, it is difficult to reconcile this approach with the anomalous behaviour model of Figure 2.18; we know exactly what behaviour is being sought, and there is no concept of objects or vehicle classes in this work. A container left on the street for 12 hours is recorded in the same way as a car. We return to this topic in Chapter 6.

2.4 Design Space Exploration and Allocating Algorithms to Hardware

Having enumerated the most common platforms for accelerating image processing operations, and having explored various algorithms relevant to the original goals of this thesis, in this Section we consider how to obtain the optimal arrangement of those algorithms on one or more hardware platforms.

The concept of Design Space Exploration is usually applied when considering partitioning of algorithms between hardware and software, on shared or discrete substrates. The two factors traded off are processing time and substrate area (for ASIC) [21] or resource use (for FPGA). Some method is chosen for assigning a score — *cost* or *fitness function* — to each possible choice and then various algorithms can be used to explore points with different characteristics in design space.

The partitioning can be fine-grained (such as partitioning at what is essentially an instruction level [118]), or more coarse-grained, placing either algorithm stages or whole algorithms on specific devices. The work in §2.1.5 can also be considered as Design Space Exploration (DSE), and our work in Chapter 6 can be interpreted as a particular case of this too.

The fitness function normally depends on the latency and area results, but can perform multiobjective optimisation by minimising several parameters; software execution time, program memory, data memory, and hardware execution time and area cost [119]. Under these circumstances the cost function can be expensive to compute in itself. Exhaustively evaluating the design space itself is NP-hard [120].

Various algorithms are therefore used to explore a subset of the design space. These can be on the level of heuristics as in Zuluaga and Topham [121], or, in similar work by Almer *et al*. [122], via standard machine learning techniques, *i.e*. neural nets and support vectors. Almer *et al*. note that neural networks are particularly effective for unseen designs for the ASIC problem. The same approach also produces good results when tasks are distributed over embedded networks [123]. Even when a single architecture is selected, work by Bouganis *et al*. [124] demonstrates that multiplier stages within a $2D$ filter can be allocated between design elements with different properties — specifically, fabric and embedded multipliers on a FPGA. This trades off higher accuracy in the former against higher resource use on the device.

Coarse-grained architecture selection is perhaps more relevant in our application, so we move to the problem of performing automated DSE where the end result is an algorithm partition between multiple devices. Focusing on work aimed at allocating processing between platforms in a single system rather than multiple distributed systems (in other words ruling out large-scale job-shop scheduling problems), the work of Quinn *et al*. [125, 60] is closest to our intended application here. Working on the problem of interactive offline image analysis, Quinn built the Dynamo system to accelerate these operations. (Although this problem would probably be tackled in software by a GPU nowadays, this approach is nevertheless instructive for the broader topic of algorithm mapping in a heterogeneous system.) Given an operator-chosen pipeline of image processing algorithms and knowledge of the source image size, a detailed *solution* pipeline of data processing, conversion and transfer operations is then generated by performing DSE; see Figure 2.22 and Figure 2.23 respectively.

Pipeline compilation is then performed using a "LEGO block approach", by connecting pre-existing component implementations for all pipeline stages. If necessary the FPGA is reconfigured, then the image is then processed by the pipeline execution unit. When moving a stage to hardware, communication or transfer costs, padding costs and reprogramming costs must be considered, and these change depending on the neighbouring pipeline stages. Hardware or software components for each stage are selected as a function of image size. This leads to a crossover point above which it is always faster to process images in hardware. The fitness function is therefore driven mostly by latency, with a hard constraint on the area of the recon-

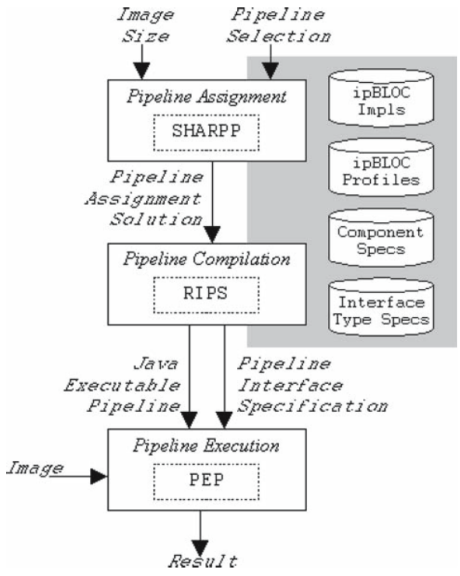


Figure 2.22: Pipeline assignment in the Dynamo system. Image from [60].

	Annotated Solution
SW/SW	(SOURCE:sw)→[none]→(mf:sw)→[none]→(histogram:sw)→[none]→(SINK:sw)
SW/HW	(SOURCE:sw)→[none]→(mf:sw)→[comm:in + reprog]→(histogram:hw:histogramRight.x86)→[comm:out]→(SINK:sw)
HW/SW	(SOURCE:sw)→[Pad:in + comm:in + reprog]→(mf:hw:mfRight.x86)→[Unpad:out + comm:out]→(histogram:sw)→[none]→(SINK:sw)
HW/HW	(SOURCE:sw)→[Pad:in + comm:in + reprog]→(mf:hw:mfRight.x86)→[Unpad:out + comm:out + comm:in + reprog]→(histogram:hw:histogramRight.x86)→[comm:out]→(SINK:sw)

Figure 2.23: Resulting allocations from the Dynamo system for a pipeline consisting of a median filter followed by a histogram.. Processing operations are in (parentheses) and transfers are in [square brackets]. Image from [60].

figurable logic (any solution which comes in above this area will have infinite cost). Algorithms for different image sizes are considered as different algorithms and are not parametrised for window size. The same base algorithm with different kernel sizes are also treated as different discrete algorithms.

All of these factors mean that design space grows rapidly with the number of algorithms, and in fact choosing the pipeline assignment problem becomes *NP*-complete [60]. To mitigate this, various search algorithms are used to explore the design space. This becomes a common problem of global vs. local search and search techniques such as dynamic programming, integer linear programming, and local search with taboo list are used at different pipeline lengths [126]. Genetic algorithms have also found some success in this area [120]. All of these involve

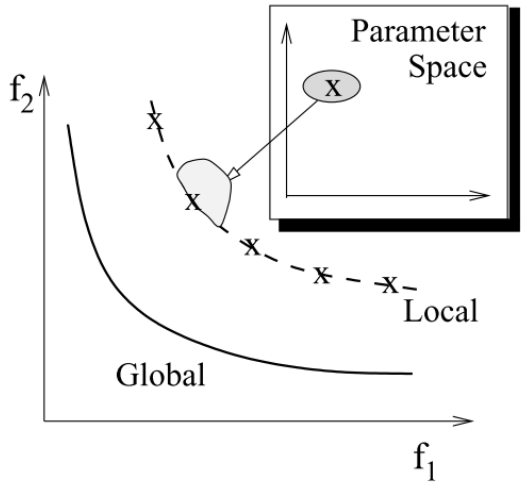


Figure 2.24: Global and local Pareto optimality. Image from [127].

various perturbations of existing solutions to produce new possibilities for which a cost function is evaluated.

A key concept here is that of Pareto efficiency for multiobjective optimisation. The two axes in Figure 2.24 represent different objective functions. As explained by Deb [127], perturbing a solution within design space using any method will cause another solution to be selected. The new solution x_2 is said to *dominate* x_1 if every objective function of x_2 is *no worse than* those of x_1, and x_2 improves upon the performance of x_1 in *at least one* objective. Eventually an optimal set of solutions will be found which cannot be improved upon, and these dominate all other positions (either locally or globally), as Figure 2.24 shows. Once this stage is reached, one of these solutions can be selected for implementation by weighting the fitness function to favour one property (*e.g.* latency) over another, based on knowledge of the target application.

As we touch on evolutionary algorithms, we provide a brief diversion into this subject. Work by Thompson [128] showed that by ignoring the normal FPGA synthesis process and running evaluations in hardware, signal processing driven by genetic algorithms generated a tone discriminator with no clock, producing "probably the most bizarre, mysterious, and unconventional unconstrained evolved circuit yet reported." [128].

Returning to Quinn's work, the fact that data transfer times between devices are explicitly considered is key, as it allows an improved estimation of the overall performance of the system before synthesis. The authors note that failing to account for the transfer and reconfiguration delays produces a 79% error in estimated vs. actual runtimes.

2.5 Conclusion

The background to this work has been described in this Chapter. Given a description of the architecture of various possible target platforms for acceleration, we then considered the applicability of each one to object classification problems within the field of image processing. These fall into the *dense linear algebra* category of Berkeley dwarves. We focus on Histogram of Oriented Gradients and its derivatives, noting recent advances in human detection in visual images, and demonstrating how those techniques apply well to other scenarios such as road sign classification. We then considered existing higher-level inference algorithms for anomaly detection. Finally, we discuss previous work in partitioning algorithms onto various hardware platforms, and exploring and selecting optimal solutions in the resulting design space.

From our evaluation of these areas of research, we note the potential for design space exploration as applied to either a changing set of tasks, or a fixed set of tasks with changing priorities. Returning to the example of vehicle surveillance, a vehicle stationary and on battery power will exist at a different point in design space than one which is moving with the engine running. The latter may require faster assessment of threats. Given that this system may deploy multiple algorithms at different times to perform different tasks, such a scenario is ripe for further characterisation.

Such a characterisation would ideally involve the HOG algorithm, given its wide applicability to various object classes. Several real-time hardware implementations on different platforms have been documented, and a principled comparison of these across one or more heterogeneous platforms within a system would provide guidance about how HOG and its derivatives should best be deployed for real-time applications under power constraints. Doing this then allows us to explore dynamic

performance tradeoffs in an anomaly detection task, again with a mobile surveillance vehicle scenario in mind.

This overall goal is the subject of our efforts in the remainder of this thesis. To set the scene, in the next chapter we consider the sensors, platforms and lower-level algorithms available or developed within Thales, which will form the basis of our heterogeneous processing system.

3. Sensors, Processors and Algorithms

Having performed an exploration of the academic literature as documented in the previous Chapter, our focus now turns to designing a system capable of real-time object and anomaly detection. Several big implementation questions must now be addressed:

1. *is a simulation-centric approach possible and does it offer any advantages, or is immediately moving to targeting hardware a better strategy?*

2. *if we aim to perform fast design-space exploration to choose the most appropriate selection of platforms to meet our power, speed and accuracy goals, how large is this design space and what algorithms are suitable for exploring it?*

3. *can existing Thales sensors be used to supply more information about any observed scene, and does doing this offer any benefits?*

4. *should any preprocessing should be performed, and if so where? Does running preprocessing or segmentation algorithms on incoming data offer any advantages?*

These questions are explored in this chapter. As we will see, the outcome from this chapter is that we can close off various — to us — unrewarding topics of study and concentrate on constructing a hardware-based system. The evidence supporting this decision is summarised in Section 3.7, the conclusion of this chapter.

3.1 Introduction

Thus far, we have not explored in detail the broad problem of how to move from an algorithmic representation of a task to its implementation on a hardware platform, preferably one running in real-time. This was a question which needed resolving in the early stages of this work. We first investigated the opportunities offered by model-based design techniques for mapping to FPGA and, to an extent, GPU. This was done in conjunction with the hardware implementation of multi-modal image segmentation algorithms, in order to explore their applicability to the bigger questions of Chapter 1, and to explore the applicability of real-time implementations of existing work at Thales.

First we consider available sensor modalities in Section 3.2. We then discuss our choice of hardware platform and explore any alternatives in Section 3.3. Section 3.4 lists the arguments for focusing on either simulation or hardware in the design process.

Following this, Section 3.5 describes the image segmentation algorithms we consider as part of an early stage in any processing pipeline (recall the location of segmentation tasks in Figure 2.1). Section 3.6 describes ways to automatically arrange these and other algorithms in a *processing pipeline*, and pitfalls when mapping this to hardware. Finally, Section 3.7 summarises our conclusions from this chapter and re-states the direction we will explore next.

3.2 Sensors

As mentioned in the Introduction, Thales is primarily an electro-optic sensor company, now contemplating a move from image enhancement to image processing. Here we summarise the equipment which they have available for image acquisition as part of a scene surveillance task.

3.2.1 Infrared

The Catherine MP is Thales' high-end infrared camera, giving an output as in Figure 3.1a. This produces images with low noise, making human monitoring

(a) Catherine MP Infrared (b) Visual Camera

Figure 3.1: A person shown on infrared and visual cameras.

less error-prone and machine vision processing of the output possible. Pedestrian detection from infrared or multimodal infrared and visual images is possible, as Bauer *et al*. showed [71]. This could be relevant in situations like in Figure 3.1, where processing handoff between visual and IR in dark or low-contrast areas could be explored; however, the added complexity makes this difficult to justify. In addition, a relevant benchmark or dataset containing hard-to-detect humans in joint visual and infrared datasets would be challenging to gather.

A polarimetric version of the Catherine MP was also available, where individual pixels on the sensor had been polarised into $0°, 90°, 45°$ and $135°$ angles to detect long-wave infrared radiation polarised at various angles. This allowed discrimination between man-made and natural objects, as documented by Connor *et al*. [129]. An example is shown in Figure 3.5a.

3.2.2 Visual

A standard visible-light sensor was available, as shown in Figure 3.1b. This operated at a resolution of 1024×768; footage from this camera was used as part of the processing time evaluation in Chapter 5. In some trials, images from a sensor head with six separate cameras were available. These faced in the same direction but operated at different wavelengths in the visual to near-visual infrared spectrum, at $450nm$, $500nm$, $550nm$, $650nm$, $700nm$ and $880nm$ respectively as noted by Letham [130]. See Figures 3.4a and 3.4b as an example.

We now discuss processing platforms and return to algorithms making use of this sensor data in Section 3.5.

3.3 Processing Platforms

Here we consider the processing platforms suitable for constructing a heterogeneous real-time system. Such a system should include FPGA and GPU as well as CPU. As discussed in Chapter 2, CPU provides a baseline for accelerated algorithms to be measured against, is required for GPU control, and is needed for processing of sequential algorithms, while FPGA and GPU were chosen because of their wide applicability, occupation of discrete and arguably complementary points in design space, and existing documented implementations of selected algorithms.

Xilinx FPGAs were chosen, based on their market share and availability of licences within Thales. NVIDIA GPUs were chosen, based on the support for CUDA present in the OpenCV library.

Various other processing platforms were considered for their viability as components in a heterogeneous system, including Thales' Ter@pix platform, discussed below. As one of the aims of this project was to consider embedded vision applications, various low-power embedded processing platforms were briefly considered. However, all of these had limited computational power and extensibility, particularly in the form of low-latency high-bandwidth links which would allow a heterogeneous system to be built and characterised.

In 2010, when hardware was being selected for this project, embedded systems with graphics processors suitable for general-purpose processing were rare (for example, they could not be programmed using CUDA and programming with the OpenCL API had only recently become available [131]). Adding an embedded processor to a Xilinx FPGA system required either instantiation of a Microblaze IP-core on-chip (reducing the number of slices available for application-specific processing), use of an embedded PowerPC microprocessor on older architectures, or an interface to an external processor elsewhere in the system, with all except the last requiring extra development time. Decisions on the hardware to use in a development system were eventually made on the basis of availability of existing code and documentation in

the image processing domain and support and existing understanding of a given architecture within Thales.

3.3.1 Ter@pix Processor

A signal processing platform created by Thales Optronics in France and known as Ter@pix was also investigated for use as an accelerator. Usually referred to as *MORPHEUS* within the academic literature, it is described publicly in a technology report [132]. It is based on a set of SIMD Processing Elements (PEs) using the underlying resources on a Xilinx Virtex-5 FPGA; each PE is arranged around one blockRAM and DSP48 embedded multiplier. It was accessed over a PCIE link and data transfer to and from the PEs was controlled by a Microblaze soft processor. SIMD operations were done using *operators*, analogous to GPU kernels, which performed operations such as morphological opening on image data. Despite Ter@pix appearing to be a promising platform for image processing work, it was not considered for further integration for several reasons:

- Specialised hardware (a platform based around a Virtex-5) was required, and arranging a loan of this from France to the UK could have been time-consuming to arrange.

- Specialised software (simulator, compiler and programmer) was required, and was in some cases only available in the original French.

- A very limited selection of operators was available, and development time would have been required to write, debug and test any additional ones (the operator library is conceptually similar to the Nvidia Performance Primitives (NPP) library, with a wide selection of low-level processing kernels optimised for the underlying hardware provided to the end user).

- Extra time would have been required to interface Ter@pix to the rest of the system (set up control and data transfer logic to and from the other accelerators).

- Any algorithm written for GPU and FPGA would then have had to be rewritten for Ter@pix to allow a proper comparison, and attempting to publish results or comparisons based around a proprietary platform such as this would have been of limited academic interest.

In short, considering that a separate FPGA platform was already present, the engineering time required to integrate Ter@pix alongside this would have been prohibitive and would have reduced the time available for algorithm development or implementation.

3.4 Choosing a Simulation- or Hardware-Centric Approach

In order to accurately characterise the behaviour of a heterogeneous system containing both the components described above, two approaches are available. First, to produce a tool for design space exploration which incorporated heterogeneity by building a virtual model of a complete heterogeneous system, we could attempt to model the entire system in Simulink, recording data transfer times between processors and GPU performance on certain scenarios and incorporating this into the model; this approach presents itself because design of the FPGA logic is one of the more complex parts of the system and expressing the behaviour of this can be done within existing model-based design tools. This approach is documented below in §3.4.1.

Alternatively, we could trade off the time required to model the rest of the system within Simulink with the time required to link the FPGA-centric model with a physical system, and obtain performance details on physical hardware, while moving away from a broader tool-based approach to system design. This approach is documented in Chapter 4.

3.4.1 Implementation Modelling in MATLAB

When considering algorithm performance analysis on multiple heterogeneous processing architectures, one of the main sources of complexity in any such project is the difference or design gap between any high-level algorithm model and its implementation across various platforms. Use of multiple discrete designs at different levels of abstraction could introduce errors in an implemented version which were not present in the original. Conversely, if model and implementation code were allowed to diverge, enhancements made to the lower-level code may not have been replicated in the model. To reduce the possibilities of hard-to-find errors such as these occurring, model-based design techniques were used wherever possible. As

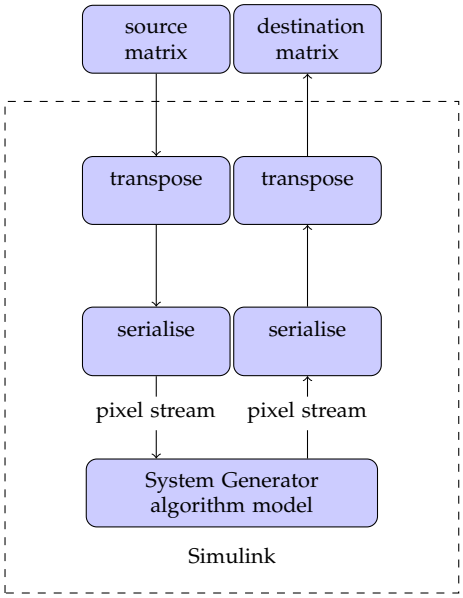


Figure 3.2: Modelling a FPGA algorithm using System Generator for Simulink from within MATLAB.

algorithm exploration and initial prototyping work was done in MATLAB, this ideally required code in MATLAB which exactly matched the downstream implementation. This approach was possible for the FPGA algorithms; using Xilinx System Generator allowed direct generation of Verilog or VHDL from the model. This modelling process is shown in Figure 3.2. This model-based design approach is frequently seen in industry, and is discussed further in §2.1.4. This generally works well because at the early stages of a design, changes to the model can be made and the results evaluated in Simulink within a few minutes. The underlying architecture which supports the model is easily described and emulated within a prototyping environment. Working with this is considerably faster than re-simulation within a HDL simulator such as Modelsim, and orders of magnitude faster than re-synthesis, chip reprogramming and evaluation of changes on the target hardware.

All of these conditions are not true when considering CUDA code. NVIDIA no longer supplies a CUDA emulator and generation of CUDA code for software emulation on a PC without a hardware GPU is similarly no longer supported [133]. Indeed, there is little demand for a simulation option considering the relatively low cost and easy availability of GPUs currently. A further drawback to simulation is that NVIDIA's architecture is proprietary and information about the underlying elements is not provided in the same way that Xilinx provides details of its low-level ar-

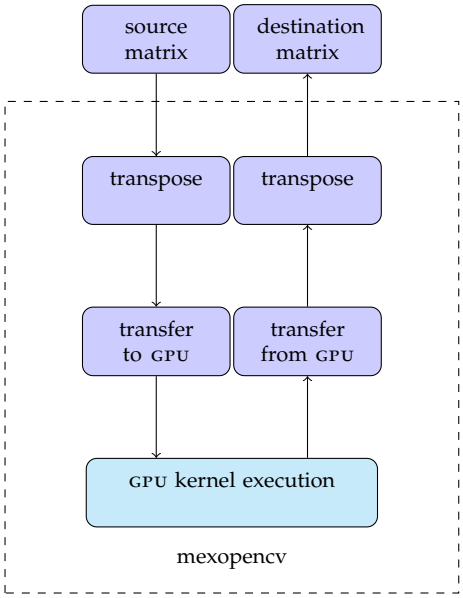


Figure 3.3: Running a GPU kernel in an OpenCV framework from within MATLAB.

chitectural building blocks. The most effective way of evaluating the behaviour and performance of a CUDA kernel is therefore to run it on target hardware. This is especially true when relatively fast compile times and ease of including debug information are considered. A model-based approach is not particularly appropriate here; CUDA kernels can't be emulated easily and there is no pressing reason to do so. When prototyping, a common practice is therefore to run CUDA kernels from within MATLAB. There are several extant approaches to this: both MATLAB toolboxes and external suppliers provide a limited selection of GPU library functions and allow evaluation of custom CUDA kernels.

The approach used in this project, in keeping with standard model-based design goals, was to use the same code to describe model and implementation for as long as possible within the design process. To do this, we extended the `mexopencv`[1] library to allow execution of OpenCV's GPU functions from within MATLAB. This ensured that any differences between CPU and GPU implementations for both OpenCV library functions and our code had a minimal effect, and made moving from MATLAB to C++ code supported by OpenCV considerably easier. Once a GPU implementation of an algorithm had been written and tested, only a small C++ function and M-file wrapper was needed before that algorithm was usable from within MATLAB. The prototyping process within MATLAB is shown in Figure 3.3. Thus, instead of

[1] Available from `http://www.cs.stonybrook.edu/~kyamagu/mexopencv/index.html`

implementing a design from a model, we ensure that our model accurately reflects the implemented design.

3.5 Algorithms for Scene Segmentation

The real-time implementation of image segmentation algorithms was considered for two reasons: first, as a continuation of another research project at Thales (described in Letham [134, 130]), which investigated algorithms for contextual anomaly detection; second, as a preprocessing stage which could generate contextual information. This could be used to segment or mask image regions, with the goal of reducing execution time by removing non-viable image regions. For example, for aircraft or fast-moving object detectors, regions which were not sky should be ignored, while sky regions should be discarded for pedestrian and vehicle detectors. This could have resulted in processing a smaller image with each of the more expensive object detectors. Various scene segmentation algorithms were identified as candidates for inclusion in a preprocessing stage as part of a real-time scene understanding platform. Details of each are given below. Each algorithm was implemented on GPU within an OpenCV framework, and was modelled for FPGA within System Generator for Simulink. Simulation details are given in the following Section.

3.5.1 Vegetation Segmentation

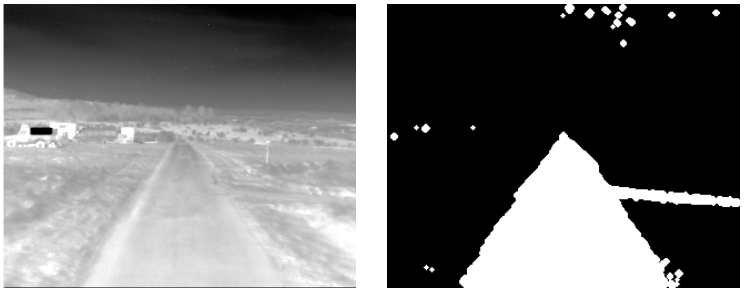
Data for this test was provided by the 650nm and 880nm sensors from the six-sensor head. As Tucker [135] notes, a suitable measure for identifying areas of vegetation from satellite imagery is the *normalised vegetation index:*

$$NDVI = \frac{NIR - RED}{NIR + RED}, \tag{3.1}$$

where NIR is the intensity obtained from the 880nm camera, and RED that from the 650nm sensor. Thresholding $NDVI$ at 0 then selects all vegetation regions in the image. A real-time GPU version of this was trivial to implement, as was the FPGA version to model. Output from the GPU version is shown in Figure 3.4.

(a) 650nm RED (b) 880nm NIR (c) NDVI output

Figure 3.4: Registered source cameras and vegetation index.



(a) Polarimeter (b) Segmented road

Figure 3.5: Road segmentation from IR polarimeter data.

3.5.2 Road Segmentation

Road segmentation has a long history in Computer Vision, and previous investigations made use of lane markings [136] in conjunction with vanishing points [137] for detection on structured roads. Detection on unstructured roads has also been demonstrated in a desert environment [138]. However, the method investigated used images from a Thales Catherine MP long-wave infra-red camera, as described in §3.2.1. Tarmac roads, and other man-made objects, are polarised differently to natural objects, allowing segmentation of roads with no markings from the surrounding vegetation. Using intensities from light polarised horizontally (i_0) and vertically (i_{90}), and some basic logical and morphological operations, a region corresponding to the road was extracted, as shown in Figure 3.5. FPGA and GPU versions were modelled and implemented respectively, then characterised.

3.5.3 Sky Segmentation

A blue-sky segmentation algorithm operating on colour images has been described in [139] by Zafarifar *et al.*, with a FPGA adaptation by Gaydadjiev *et al.* documented

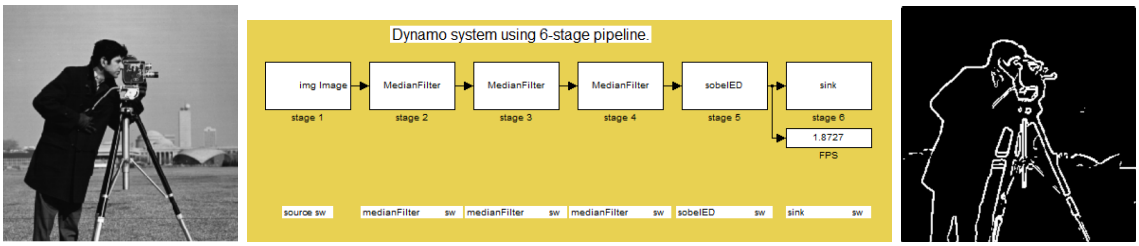
Figure 3.6: Sky region (highlighted in red) segmented from visual camera.

in [53]. This generates a probability that a pixel belongs to the sky region using colour and texture information and a vertical position heuristic. The results of this on a colour source image are shown in Figure 3.6. A GPU-accelerated implementation of [139] was written for this project, along with a partial re-implementation of the FPGA version described in [53].

3.6 Automatic Processing Pipeline Generation

A key component of the early stages of this work was a feasibility study of an allocation algorithm; this was designed to decide which platform to assign various image processing algorithms to. The previous work most closely aligned to this was done by Quinn *et al*. [60]. They described a system for selecting hardware or software implementations of image processing operators at runtime, given an input set of algorithms to run and an image to process. Quinn *et al*. used various search algorithms (dynamic programming, tabu search) to efficiently search the design space at runtime to generate an optimal solution. Their methods are covered in more depth in Chapter 2.

Based on this work, we reimplemented various search algorithms within MATLAB, and created a library with CPU, GPU and FPGA versions of the image processing algorithms described in the previous Section, as well as a small set of standard low-level image processing operators such as median filters, morphological operators, *etc*. The latter were either modelled in Simulink or abstracted as calls to manufacturer-supplied libraries (such as NPP) where possible. The aim behind this was to evaluate the feasibility of creating such a system for running dynamically-selected operators across a heterogeneous system in real-time.



(a) Source (b) Pipeline (all in software) (c) Sink

Figure 3.7: Sample image processing pipeline in Simulink, generated from implementations which were selected automatically. This performed repeated median filtering followed by Sobel edge detection, with each pipeline stage implemented in hardware or software.

In more detail, the system behaved in the following manner:

1. **search**: given an input list of image processing operators to apply, a search algorithm was used to explore the design space. Given a constraint (device utilisation or latency), the algorithm selected the best possible pipeline within that space, for a given search algorithm runtime;

2. **generate**: a Simulink pipeline is constructed by connecting the appropriate implementations of each operator together, and adding any appropriate transfers between platforms;

3. **process**: the image is processed through the generated pipeline.

An example of steps 2–3 is shown in Figure 3.7, using an all-CPU pipeline and a subset of available operator blocks. The purpose of this was to evaluate automated interconnection of components rather than design state-of-the-art edge detection algorithms.

In short, this system simulated automatic generation of image processing pipelines made up of auto-coded operators; candidate algorithms are shown in Table 3.1. To generate this pipeline, we then evaluated the performance of a variety of search algorithms including dynamic search, genetic algorithms, and local search using steepest-descent and tabu methods, within various runtime constraints. These were compared to exhaustive search and various naïve search methods.

All of these used a fitness function which was evaluated for each candidate solution; this relied on properties of each candidate implementation such as runtime and transfer time (*i.e.* the time taken to transfer data between processors). There were several drawbacks with this approach:

- As each stage in this pipeline was simulated within MATLAB or Simulink, runtime for software-based platforms (CPU and GPU) could not be confirmed, and times for data transfer between platforms could only be estimated.

- The algorithms themselves, particularly those for segmentation, were in most cases insufficiently computationally complex (*i.e.* having a low ratio of computations to data transfer) to obtain much benefit from accelerating them.

- In addition, for the segmentation algorithms described in Section 3.5, multiple image sources at various wavelengths would have been required to generate scene segmentations, requiring a non-linear processing pipeline to be generated and increasing the complexity of the pipeline generator.

- Several of the algorithms in Table 3.1 were not yet implemented by the time the pipeline system was written. Even with autocoding to HDL, at least two versions of every algorithm would have been required to adequately explore the design space. Writing and testing each individual implementation would take a considerable amount of time, and in the meantime performance estimates would have been required by the search algorithm.

- One of the aims was to compare the effectiveness of several types of search algorithm. To adequately exercise the search algorithms in a large, well-populated design space (one where the runtime required to exhaustively search all possible combinations of implementations would have been prohibitive) would have required a large number of implemented algorithms.

- There was no obvious path between creating a modelled system (as shown in Figure 3.7b) and automatically deploying that implementation on hardware in real-time, as we discussed in Section 3.4.

Based on the lack of concrete performance data for the fitness function, the time required to code multiple implementations of multiple segmentation algorithms,

Implemented and Envisaged Simple Image Processing Algorithms		
sky segmenter	road segmenter	vegetation index
people detector (HOG)	background subtractor (MOG)	median filter
hotspot (motion) detector	histogram generation	

Table 3.1: Simple Image Processing Algorithms – either implemented or considered for implementation within a Simulated Processing Pipeline.

and the difficulty of modelling a complete heterogeneous system in Simulink, the decision was taken to move to a more in-depth examination of complex algorithms in a physical platform consisting of an FPGA and a GPU.

3.7 Analysis and Conclusions from Early Work

The work performed in the early stages of this project has been described in this chapter. After covering available sensors from Thales, in Section 3.3 we discussed our choices for processing platforms, eventually concentrating on FPGA and GPU. These findings were reinforced by studies of a simulated Ter@pix system; the same algorithms would need to be reimplemented *again* on a more restricted processor, which would need integrated into what would be a four-way system for transferring data between heterogeneous processors. This time could arguably be more productively spent by producing better implementations of existing algorithms.

As discussed in Section 3.5, when investigating image segmentation algorithms on multiple platforms, the algorithms were not sufficiently complex to produce a compelling argument for accelerating such processing stages on FPGA or GPU. In addition, the need to use multimodal sources to generate segmentations complicated any opportunities for data collection and benchmarking and would have reduced the clarity of the argument in any resulting thesis. The comparatively large number of algorithms envisaged – and the time required to implement each one – further added to complexity but did not add to the potential to make a contribution in the field of real-time object or anomaly detection.

As expanded upon in Section 3.6, attempting to model such a system without a well-characterised physical platform and without adequate implementation performance data would not have produced results representative of a real system.

The conclusions reached after these early explorations and simulations therefore served as a natural break point within the project. The overall goals were then reformulated and the focus narrowed considerably, subsequently concentrating on evaluating the performance of a single complex algorithm with components allocated across multiple heterogeneous platforms, before building on that to reach the level of abstraction needed to perform anomaly detection. The next chapter describes the architecture required to achieve this.

4. Selection of System Architecture

Chapter 3 investigated and closed off the possibility of preprocessing and filtering images prior to detection, and by demonstrating the unsuitability of a simulation-based approach, moved the focus of the work towards building a hardware-based system for object detection and, ultimately, scene analysis and anomaly detection.

We now describe the platform on which algorithms described elsewhere in this work were evaluated, in particular the specifications of the three main processing elements. This is followed by an overview of the PCIE *network used for data transfer and communication, and a description of a sample* FPGA*-accelerated image processing operation. The system constructed in this chapter is used for the experiments in Chapters 5 and 6.*

4.1 Processor Specifications

Following the decision to use physical hardware wherever possible, discussion of the development system is now appropriate. This consisted of a desktop PC with an Intel x86 processor (a dual-core Xeon 2.4GHZ chip, referred to throughout as the host or CPU).

This system had 4GB RAM and was running 32-bit Windows XP. Two accelerators were connected: a NVIDIA GeForce 560Ti GPU, and a Xilinx ML605 development board containing a Virtex-6 XC6VLX240T FPGA. The GPU had 384 CUDA cores, grouped into SMs of 32 cores. This unit belongs to NVIDIA's Fermi family, which is defined as having a compute capability of 2.1. This means it has certain features designed to support general-purpose processing, such as limited hardware support for double-precision floating-point arithmetic. The FPGA had 768 fixed-point embedded multipliers available, compared to the 384 single-precision floating point ones

Figure 4.1: ML605 FPGA card (upper) and GPU card (lower) in development system.

on the GPU. Here, we treat both FPGA and GPU as accelerators in a larger system, rather than complete image processing systems themselves. This allows the reading and display of images and videos to be done in software, without handling this in firmware; this greatly reduces the development time required to make use of a variety of image sources.

All discrete GPUs in PC-based systems are connected via a PCIE link, and this model used a 16-way PCIE 2.0 connection. PCIE was chosen as the common method of data transfer between the memory of all three processors. A brief description of a PCIE network and methods of data transfer within it is given in §4.2.1.

4.2 System Architecture

When work on this system began, different options for data transfer between the various processors were considered. The ML605 FPGA development board had several high-bandwidth communications interfaces (gigabit ethernet and various high-speed serial protocols) available, as well as PCIE support. However, as the GPU was only designed to transfer data to and from host memory (and optionally to a display), the only interface available was PCIE. This was therefore chosen as the mechanism for transferring data between all three processors; any others would have incurred either more discrete data transfer steps between buses and devices, or

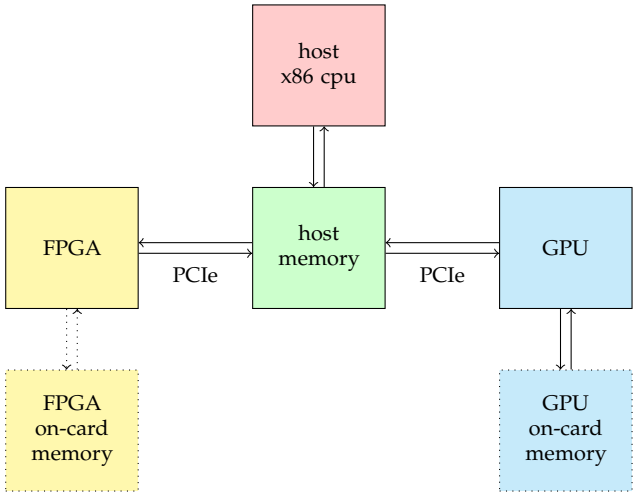


Figure 4.2: High-level system functional diagram showing arrangement of processors. Each processor can access host main memory, and the two accelerators have private access to on-card memory.

additional complexity and possibly decreased system reliability as various drivers and interfaces were modified to directly transfer data between each other in ways which they were not designed for. Figure 4.1 shows both accelerator cards running inside the PC.

A high level arrangement of processors is shown in Figure 4.2. §4.2.1 describes the arrangements for data transfer, and §4.2.2 describes an example FPGA-accelerated transfer and processing operation.

4.2.1 DMA Controller for PCI Express

The PCI-express subsystem within a computer system can be thought of as a network, made up of a root complex and several child nodes or *endpoints*. Connections between nodes are by means of one or more serial full-duplex links, in contrast to the older PCI architecture, which was a traditional bus arrangement with data broadcast to every device on the bus. A PCIE 1.0 $\times 8$ link describes an 8-way full-duplex link operating at 2.5 Gigatransfers per second (GT/S) or 2.0 GB/S, while a 2.0$\times$16 link describes a 16-lane link at 5.0 GT/S or 8.0 GB/S. In an x86-based system, the root port is coupled closely to the processor, from which the memory controller allows access to RAM[140, 141]. A PCIE network topology diagram is shown in Figure 4.3. In Figure 4.2, for conceptual purposes, the accelerators are shown as being able to access host memory directly. Architecturally, only the processor is

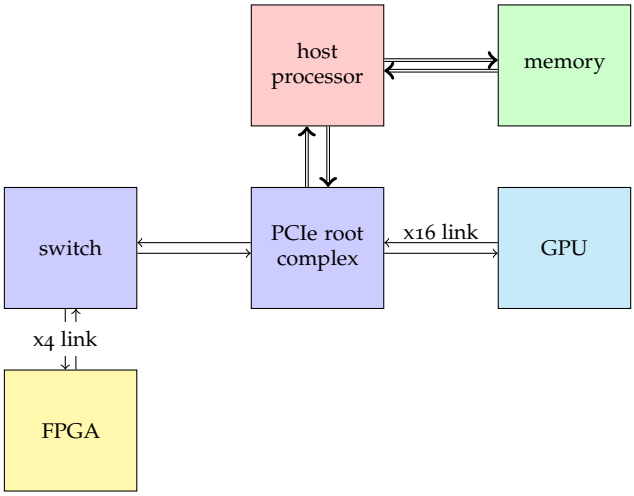


Figure 4.3: System architectural diagram showing interconnections between processors over PCIE network.

directly connected to memory. It contains an integrated memory controller, and the PCIE accelerators access host RAM through it.

Memory or registers on an endpoint can be mapped into the system address space and will appear in the host memory map. Mapping a device's Base Address Register (BAR) in this way allows its registers or memory to be accessed by the host processor in the usual fashion. This allows the endpoints to read and write system RAM through DMA transfers. These have the following format:

- source address to start reading from
- transfer size
- destination address to start writing to
- transfer direction (host→device or device→host)

These BARS do not need a 1:1 mapping between the physical memory they represent and the range in the system address space they take up: for devices with gigabytes of on-board RAM, this would be prohibitive, particularly with a 32-bit address space on the host. In a full implementation, this allows memory attached to one endpoint — and not present in the system address map — to be accessed from another endpoint. For example, if a request to read data *from* system memory is

passed to an endpoint, the destination write address given may only exist or have meaning in the endpoint's private memory. However, here we are only concerned with transfers between a device and host memory. When considering a NVIDIA GPU, all these details are abstracted away behind the `cudaMemCpy` function in the CUDA API, and DMA transfers are handled by dedicated circuitry on the GPU. However, the Virtex-6 FPGA has hardware support for PCIE transfers (in that its high-speed transceivers are connected to PCIE lines) but no built-in firmware support for performing transfers above the link level. A DMA-capable controller was required to interface the FPGA to the rest of the system, and this was adapted from an existing example documented in an application note for the Virtex-5 [142]. As the DMA controller forms a significant part of the FPGA logic, its operation is summarised below.

The function performed by a DMA controller in a PCIE network is to break down a complete DMA transfer, as described by the parameters above, into individual groups of transactions which are then streamed across the PCIE link. The data to be transferred is always split into Transaction Layer Packets (TLPS); as a PCIE link is used for both data and control functions, saturating it for long periods by transferring large volumes of data is not allowed. This is described fully in the PCI-Express Base Specification [141].

DMA transfers can be device-to-host, or host-to-device. As we consider these from the point of view of the controller on the device, these are known as *egress* and *ingress* respectively. At system startup, the device registers its BARS with the host controller. This is one small BAR containing control registers and one or more larger data BARS. The maximum amount of data which can be transferred in a single transaction or Maximum Payload Size (MPS), is dictated by a combination of the system and device capabilities, and is established during link initialisation after system reset.

Consider an example 1MB ingress transfer. At the beginning of the transfer, the host issues a DMA request by writing the transfer details to registers within the control BAR: in this case, a read address within system memory, a write address inside one of the device's data BARS, and a transfer size of 1 MB. The controller then issues a *read request* TLP to the host for a small contiguous volume of data, usually around 1-4KB. The host then responds with a number of *completion packets* containing the requested data, with each one being a known maximum size. As each packet arrives,

it is checked and the data is extracted and written to device memory or an on-chip buffer. The controller must keep track of all outstanding read requests, re-issue them if they time out, and re-order the data if it arrives in an unexpected order. This procedure continues until the entire 1MB has been transferred.

The process for an egress transfer is similar, except the *host* issues a series of read request packets, which the device must respond to by reading data from internal memory or a buffer, forming completion packets and sending them. A mechanism for keeping track of timeouts and signalling the host that a request could not be completed was not implemented in this system, so all read transfers had to be sized so that they did not request more data than would be generated by the device during an operation: if this happened, the DMA controller would assert the `transaction_source_ready` line in the DMA core forever while waiting for data which would never arrive. This would then crash the host OS.

Achieving high throughput for a controller requires extensive pipelining and the capacity to keep track of a large number of packets at once, particularly during full-duplex operation. Complexity here is also increased by the relatively high clock speeds that the DMA logic had to operate at; this required constraining much of the DMA logic to be close to the hardware high-speed transceivers on the FPGA. This was simplified slightly by only using onboard blockRAMs, rather than instantiating a memory controller and transferring data to and from the external SDRAM on the FPGA board; this also removed the need for arbitration between application and DMA controller for access to memory, and between memory controller and application for access to the DMA controller.

The `WinDriver` library[1] was used to generate a Windows device driver for the FPGA; this handled BAR allocation and passed register reads and writes of the DMA registers to and from the FPGA. The DMA controller interfaced to the application logic through two First-In First-Out buffers (FIFOs), one for image data being streamed in and one for data being streamed out. A control register system was also implemented. This allowed parameters and options for the algorithm used to be configured from the host. A schematic is given in Figure 4.4. The DMA system was clocked at $250MHz$ as required by the PCIE link logic, while the application

[1] Available from `http://www.jungo.com/st/windriver_usb_pci_driver_development_software.html`

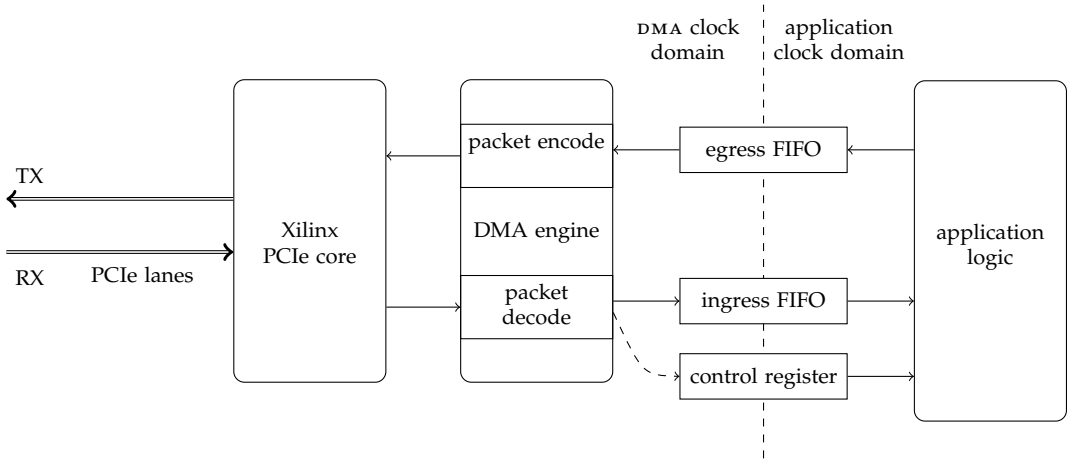


Figure 4.4: System internal FPGA architecture.

logic was clocked separately. The bitstream was synthesised using Planahead and ISE 13.4.

4.2.2 Exemplar FPGA-accelerated OpenCV Operation

Having described the operation of the DMA subsystem, a description of an exemplar data transfer will give an understanding of its operation within a complete image processing application. The principal data type for operations on images in OpenCV is the `CvMat` matrix. This consists of a header structure containing basic information (height, width, row stride length, pixel data type (`char`, `float`, *etc.*)) and a pointer to the image data. The FPGA interface therefore had to operate on images stored in this data structure to allow interoperability with the rest of the application. Here we consider an operation of the form `operation(source, destination)` where `source` and `destination` are `CvMat` types with pointers to data in host memory, and `operation` is one for which a streaming implementation exists on the FPGA. In order to perform the calculations described by `operation` on FPGA, the following steps must be performed by the host:

1. Perform any relevant preprocessing steps such as colour-space conversion or padding.

2. Allocate an area of page-locked memory[2] and copy the source matrix into it.

[2]Memory which must stay in physical memory and which the operating system is prevented from paging to disk.

3. Calculate data transfer sizes. For ingress transfers, this must be sufficient to transfer all the original data, plus sufficient to push the data through any pipelined application logic and push the resulting processed data into the output buffer. The egress transfer size must be large enough to contain all the resulting data, but not more than would be generated by the application logic, otherwise the egress transfer would stall while waiting for output data from the application logic, and crash the host OS.

4. Obtain a lock on a mutex controlling access to the FPGA. (Required if the host is running multiple worker threads, all of which access the FPGA).

5. Write DMA transfer details into registers, and write to the control register to start the full-duplex transfer.

6. Poll FPGA status register until transfer is completed. The results returned by the operation will now be in the output buffer in host memory.

7. Copy destination matrix into another location for further processing, or perform further processing as part of the operation in-place then place the results elsewhere.

8. Unlock the FPGA mutex.

9. Construct and return a `CvMat` header structure for the destination matrix, where the data pointer points to an area of page-locked memory accessible by both the FPGA and host.

During step 5, the DMA controller streams the data from the PCIE lanes to the input FIFO, from where it is read by the application logic. This can include unpacking individual pixels from the 128-bit words which the DMA controller works on, to the 8- or 32-bit words used by the application. The DMA controller monitors the amount of data in the FIFO; once it rises above a predefined level, read requests to the host for more data are paused until the application consumes more data. The application then performs the defined operation on the data, repacks the output into 128-bit words, and loads the results into the output FIFO. Similarly, transfers of egress data are not started until there is enough data in the output buffer to perform a group of transfers. This method achieved two simplifications: the DMA controller was not

required to access the onboard memory on the FPGA and was thus less complex, and performance of the overall processing operation was governed almost completely by the rate at which the application logic consumed and generated data (save for the initial delay in filling the input FIFO).

This abstraction thus allows FPGA-accelerated image processing operations to be transparently integrated with the rest of a C++ application relying on OpenCV.

4.2.3 Interface Limitations

There are several limitations associated with the interface. These were the result of decisions made to reduce time spent developing the interface rather than application-specific logic. As OpenCV stores colour images in planar format, delivering a colour image to the FPGA would require transferring all three channels of the image to FPGA, either: sequentially by storing them in memory then synchronising delivery to the application logic; or in parallel, by increasing the number or complexity of DMA engines on-chip, or converting the image from planar to interleaved on the host, thus increasing transfer time. Each transfer was also limited to a maximum of 1MB. The external memory on the ML605 board was also not accessed, due to the lack of a straightforward abstraction to access it from within System Generator.

4.3 Conclusion

This Chapter describes the construction of a system consisting of multiple heterogeneous processors for accelerated image processing — a key part of real-time scene analysis. The specifications of the system we used are given; we then focussed on the mechanisms for data transfer within it, and described the PCIE protocol and its application in this case. This platform is used in subsequent chapters as a platform to run all human and car detections on, as part of our broader goal of power-aware scene surveillance.

5. Algorithm-Level Partitioning in a Heterogeneous System

This chapter describes partitioning of a pedestrian detection algorithm into separate operations, and an exploration of the mapping of those operations onto separate processors, using the heterogeneous system architecture described in Chapter 4.

First we describe the algorithm, and state the criteria for deciding where to partition it, then we describe in detail the software and hardware implementations required to do this. A description of the test methodology is then followed by presentation and analysis of the results using algorithm accuracy, system power consumption and processing time as metrics. This includes exploration of design space at both design-time and run-time, and in particular the tradeoffs required when moving between different elements. We compare our accelerated versions to other published implementations and show that, where data is available, our version is more accurate than existing FPGA*-accelerated implementations. Finally we explore some modifications made to the implementations which change the performance characteristics to some degree.*

The power-aware anomaly detection task in Chapter 6 uses the detection algorithm and partitioning information described in this Chapter to choose the optimal mapping of algorithms to architectures. This chapter is an expanded version of a paper published in the IEEE *Journal of Emerging and Selected Topics in Circuits and Systems in 2013 [16].*

5.1 HOG Algorithm Analysis

The Histogram of Oriented Gradients algorithm was described briefly in Section 2.2. It has become one of the standard object detection techniques in computer vision and

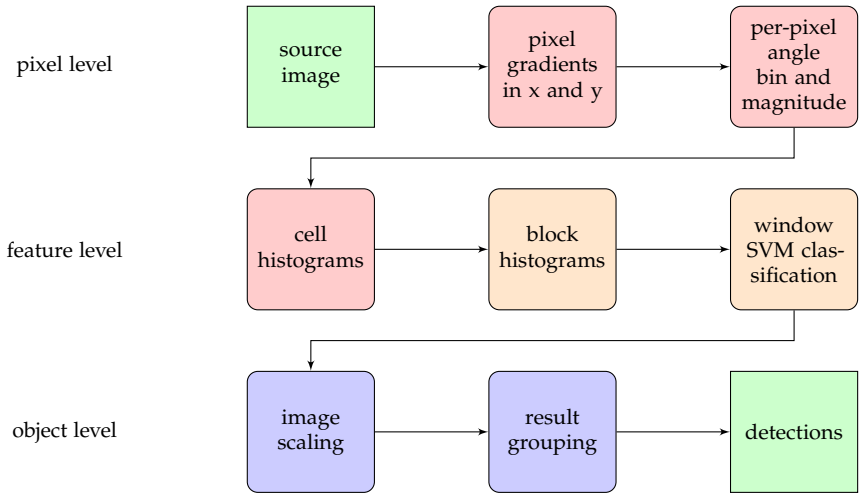


Figure 5.1: HOG algorithm stages: (i) gradients; (ii) angle & magnitude calculation; (iii) generation of oriented histograms over cells; (iv) block concatenation and normalisation; (v) linear SVM scoring; (vi) image scaling; (vii) result grouping.

versions have been implemented on several platforms, allowing for comparison with existing implementations. In addition, the current state-of-the-art object detectors are extensions of HOG in some form. The algorithm itself can be split into two compute-intensive steps, as shown by the colour groups in Figure 5.1: feature extraction and SVM classification. This expands to three if we consider image resizing as compute-intensive also.

A description of the algorithm is provided here. While HOG has been described repeatedly elsewhere in the literature, a comprehensive description here is nonetheless necessary in order to explain the behaviour of the firmware implementation and to understand the interaction of operations between different architectures. §5.1.2 describes the ideal places within the algorithm to consider partitioning between multiple processors. Section 5.2 describes details specific to hardware implementation including providing a full annotated description of the firmware algorithm flow, and Section 5.3 does the same for software details.

HOG is a sliding window algorithm. The steps below all work on an image patch which is 128 pixels high and 64 wide; this is sufficient to detect pedestrians which are around 96 pixels high. Conceptually, the window is broken down into an overlapping set of *blocks*, each with 2×2 *cells*. Each cell represents a set of 8×8 pixels; thus one sliding window contains 7×15 blocks and 8×16 cells (see Figure 5.2).

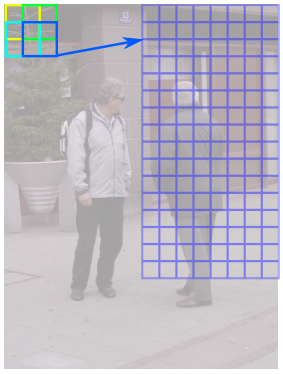


Figure 5.2: HOG blocks (left) overlap by 8 pixels in each direction, and these make up a sliding window (right).

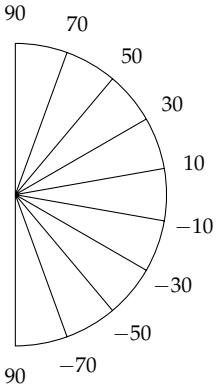


Figure 5.3: HOG histogram bins over 180°.

5.1.1 Algorithm Steps

The steps below broadly correspond with the stages in Figure 5.1.

1. *Gradients:* Gradients in x and y are taken by first normalising the input range by taking the square-root, then convolving with a $1D$ kernel $[\ 1 \quad 0 \quad -1\ ]$ in both directions. For colour images, each channel is processed separately. Pixel gradients g_x and g_y are generated.

2. *Orientation and Magnitude:* For each pixel, the magnitude $M = \sqrt{g_x^2 + g_y^2}$ and angle $\arctan \frac{g_y}{g_x}$ are calculated. Dalal used 9 bins, split over $0 - 180°$; he noted that this is effective for pedestrian detection, but can be changed to $0 - 360°$, which is more effective for man-made objects. Constants $b_0, b_1, \ldots, b_9$ representing the edges of each bin $B_0 \ldots B_8$ are set as $\tan 0°, \tan 20°, \ldots, \tan 180°$. For each pixel, $\frac{g_y}{g_x}$ is evaluated and *e.g.* bin B_0 is chosen if $b_0 \leq (g_y/g_x) < b_1$. See Figure 5.3.

3. *Cell Histogram generation:* M is then weighted based on the difference between the gradient angle and the angle of the closest bin edge, and added to the eight surrounding bins (bins B_n and B_{n+1} in all four cells in a block); this prevents quantisation errors caused by large-magnitude weights close to a bin edge. For each cell, B_n thus contains accumulated magnitudes of all pixel gradients, where the direction of the gradient falls close to b_n.

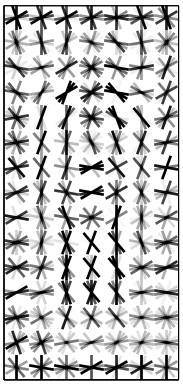


Figure 5.4: SVM person model generated by HOG training. Note the strong vertical responses on arms and sides of body and strong diagonals at shoulders.

4. *Block Histogram Generation:* Four cell histograms at indices $C_{(x,y)}$, $C_{(x+1,y)}$, $C_{(x,y+1)}$ and $C_{(x+1,y+1)}$ are concatenated to produce a 1×36 block histogram vector bv. This is then normalised to produce bn in two stages:

$$bn' = \frac{bv}{\sqrt{|bv|_2^2 + \epsilon}}$$

where $\epsilon \ll 1$. bn' is capped at 0.2, then:

$$bn = \frac{bn'}{\sqrt{|bn'|_2^2 + \epsilon}}.$$

This is L2Hys normalisation as described in Appendix A. All 7×15 block vectors in the window are concatenated to generate a feature descriptor fv.

5. *Window SVM Classification:* The above feature descriptor is multiplied by linear SVM weights of 7×15 elements with 36 weights each, to generate a window score s:

$$s = \sum_{i=1}^{n=3780} (fv_i \cdot w_i) + b\,. \tag{5.1}$$

The value of $sign(s)$ represents the presence or absence of a detection, while the magnitude of $|s|$ denotes the level of confidence in the result. Figure 5.4 shows a visual representation of the weights used.

6. *Image Shifting and Scaling:* The steps above are now repeated on a window offset by 8 pixels from the previous one. This overlapping allows better

Table 5.1: Size of data generated during each algorithm stage, per 1024×768 frame.

Algorithm Stage	Single-scale data (kB)
source	768
gradients	6144
magnitude & angle bins	3840
cell histograms	432
normalised blocks	1697
window scores	38

detection, but a pixel and hence a block may belong to up to 105 windows and calculations will be duplicated. In practice, block histograms for the entire image are calculated, then the classifier window is slid over the result. Scoring is repeated over all windows, then the image is downscaled by a factor $sf = 1.05$ and the process is repeated for n scales, or until the resulting image is too small to classify.

7. *Result Grouping:* Any resulting positive scores are then grouped into a single detection if they are of similar size and in a similar location.

5.1.2 Partitioning

The exploration of how to allocate algorithm operations to candidate architectures was done manually, based on knowledge of the calculations required for each of the above steps, and dimensions of data generated at each step, as shown in Table 5.1. We could investigate the time taken to allocate each of these stages to a separate processor, but as each transfer of data between processors takes a finite time, some of these (*e.g.* accelerating only the gradient computation stage) would yield little benefit. Instead we focus on stages which can group together, and from which we expect to see some benefit if they are accelerated.

Stages 1–3 in §5.1.1 involve stream processing and histogram binning, which map well to a heavily-pipelined FPGA, particularly when we can construct a subsystem for *e.g.* single-cycle histogram binning. Stages 4 and 5 involve repeated vector multiplications, and are well-suited to the dense arrays of floating-point multipliers on a GPU. At stage 4, memory access patterns change from streaming pixels which can be read and retained for a few lines then discarded, to cell histograms which

must be copied to multiple blocks, then classifier weights which are reused many times over. Conversely, at this stage data access on FPGA becomes more complex and requires more development time. Due to the smaller amount of data produced by the window scoring stage, the low amount of further processing required on it, and the need to make use of the result in further calculations, in all cases we perform stage 7 on the host CPU.

Thus, considering the properties of both the algorithm and the data it generates, we can generate a list of dataflows through the algorithm which are viable candidates for implementation and further investigation. Based on Table 5.1, the only data which can reasonably be transferred between different processors is input pixels at our chosen scale, cell histograms or window scores. We split these paths into 3 stages: scaling, cell histogram generation, and classification, and refer to them by the processor used to perform each task, *e.g. gfg* means "scale on the GPU, generate cell histograms on the FPGA, then normalise and classify on the GPU". All six of these paths are shown in Figure 5.5, where these mnemonics are also defined. Using this information, the question from Chapter 1 we wish to answer is *"how does the performance of an algorithm when partitioned temporally across a heterogeneous array of processors compare to the performance of the same algorithm in a singly-accelerated system?"*. Sections 5.2 and 5.3 describe platform-specific details of these algorithm implementations which answer this question.

5.2 Hardware Implementation

The firmware implementation is autogenerated from a System Generator model and fits into the OpenCV-accelerated framework described in §4.2.2. The 128-bit wide words output from the ingress FIFO shown in Figure 4.4 are shifted out pixel-by-pixel and fed into the HOG logic.

The application logic consists of multiple stripe processors placed side-by-side (see Figure 5.6: 16 stripes are required for a 1024×768 image). Magnitude and orientation information is calculated for each pixel as it is streamed in, then this information is fed to all stripe processors. Each processor operates on a 64-pixel wide stripe of image data and generates cell and block histograms for all pixels within it. These are detailed in §5.2.1 and Figure 5.7. If the data is then being transferred

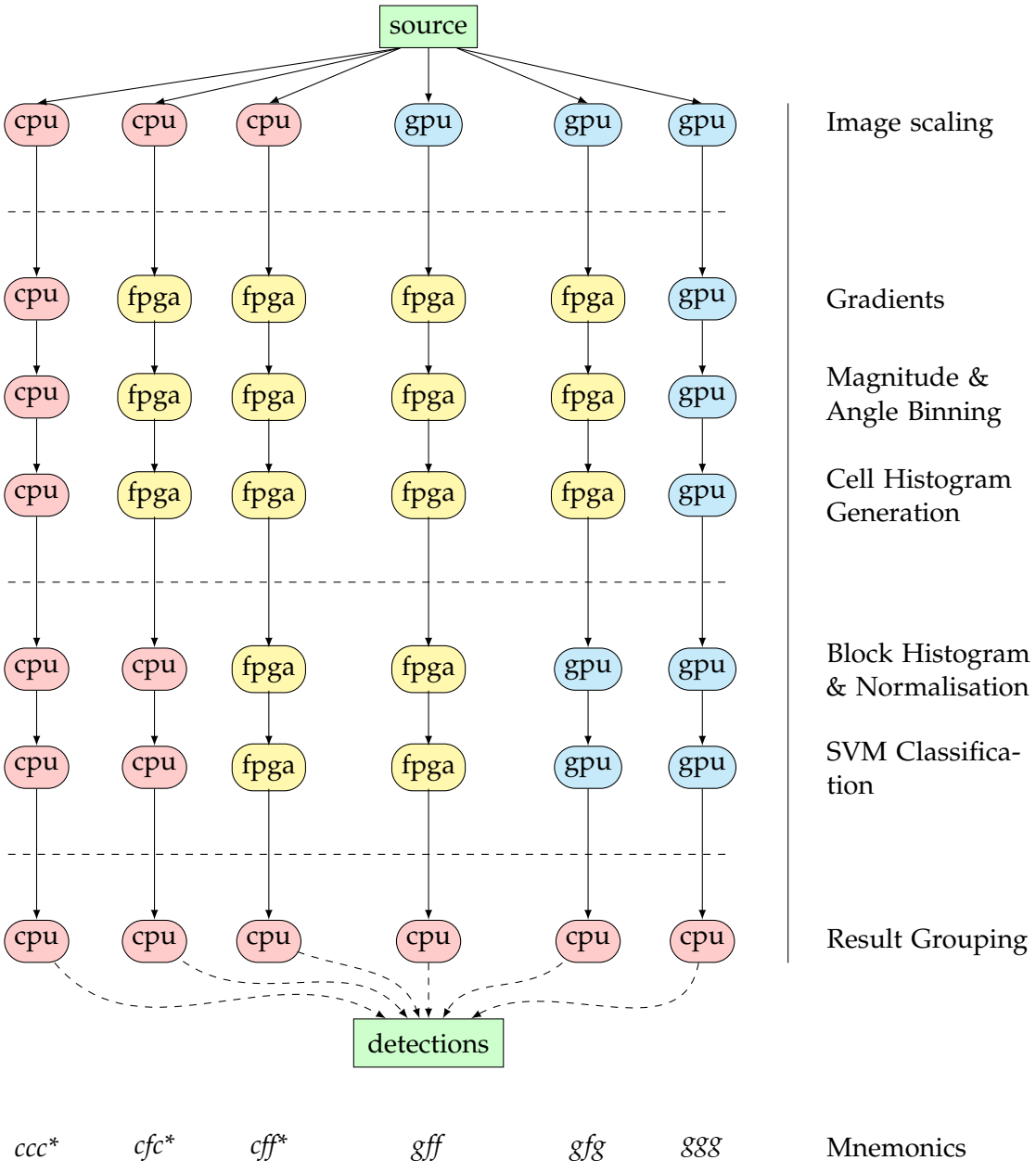


Figure 5.5: Six possible processing paths through the algorithm. Any one of these can be selected and used to generate detections. We define these using the three-letter mnemonics above in the format *resize-histogram-classify*. Starred mnemonics denote multithreaded versions.

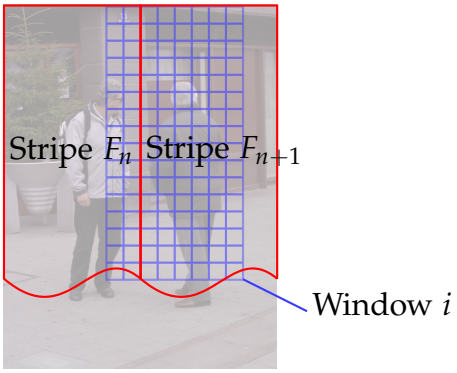


Figure 5.6: Processing regions for two stripe processors, overlaid on an image. F_n passes cell histograms and column scores for the left portion of window i to F_{n+1}.

to the host or GPU, the cell histograms can then be read out and discarded from the FPGA. Alternatively, §5.2.2 and Figure 5.8 describe classification on FPGA. For multiscale evaluation at n scales, the frame is scaled on the GPU or host CPU then padded to the original image width and passed to the FPGA for processing. While this involves more data transfer and would be slower than image scaling on-board the FPGA, complexity is reduced, as we do not have to consider changing image sizes (particularly if many scales per octave are required), or construct a memory interface and arbitration logic between the application and PCIE interface.

The sections below detail specific modifications made to the hardware implementation from the original algorithm.

5.2.1 Cell Histogram Operations

Orientation and Magnitude: Due to limitations in our PCIE interface we convert the colour image to grayscale before transfer to the FPGA. For magnitude generation, we use the magnitude approximation described in Wilson *et al.* [143]:

$$M_{approx} = \frac{1}{1+\sqrt{2}}(|g_x| + |g_y| + \sqrt{2} \times \max(|g_x|, |g_y|))$$

to avoid square-roots. We then select an orientation bin without using division or trigonometric calculations, using the method described in Bauer *et al.* [70]: we retain constants $b_0, b_1, ..., b_9$, flip the angles of any pixels with $g_x < 0$ by

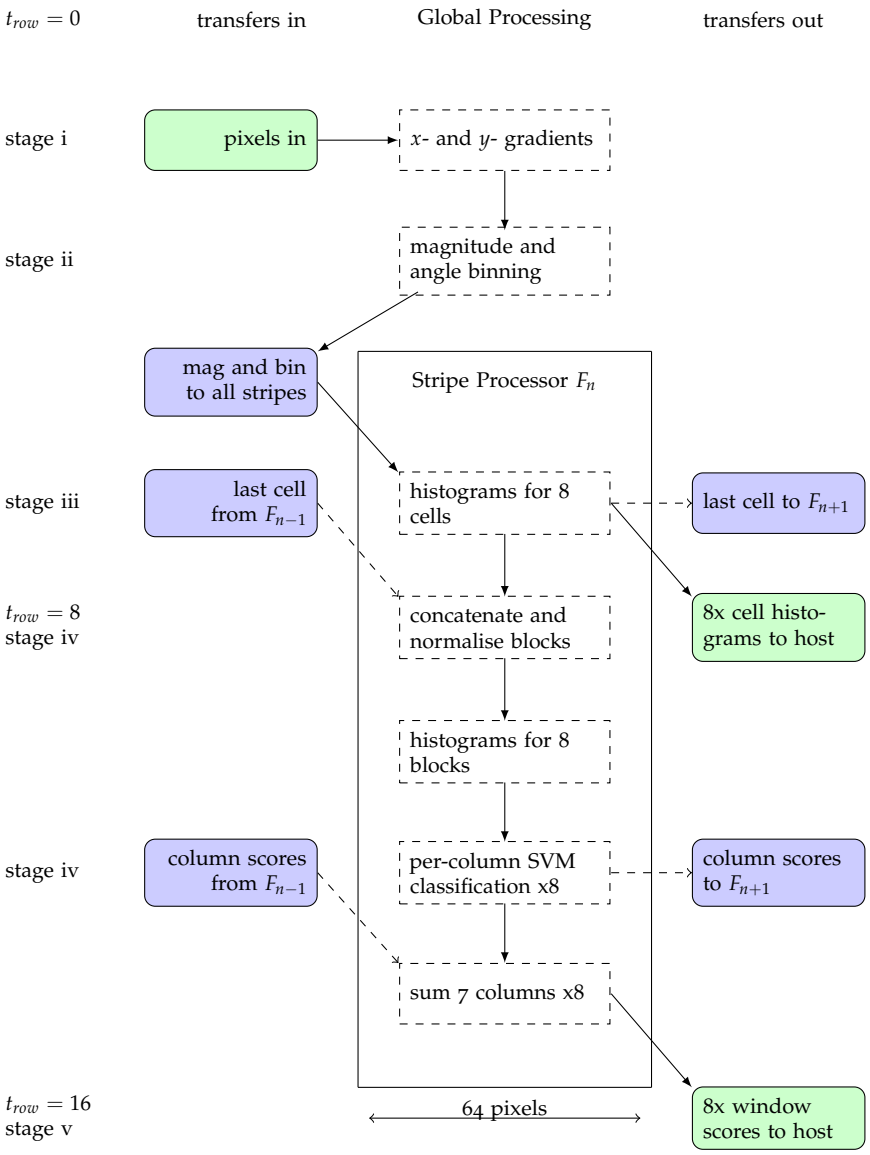


Figure 5.7: HOG stripe processors extract histograms from cells sequentially. Each stripe shares the normalisation logic, classification control logic and SVM weights between its block classifiers. As pixels are fed in, cell histograms are generated, stored in an addressable shift register, and read out after 8 rows. Internally, these are then concatenated and normalised into block histograms. Cells from neighbouring stripes are included for blocks which overlap between stripes.

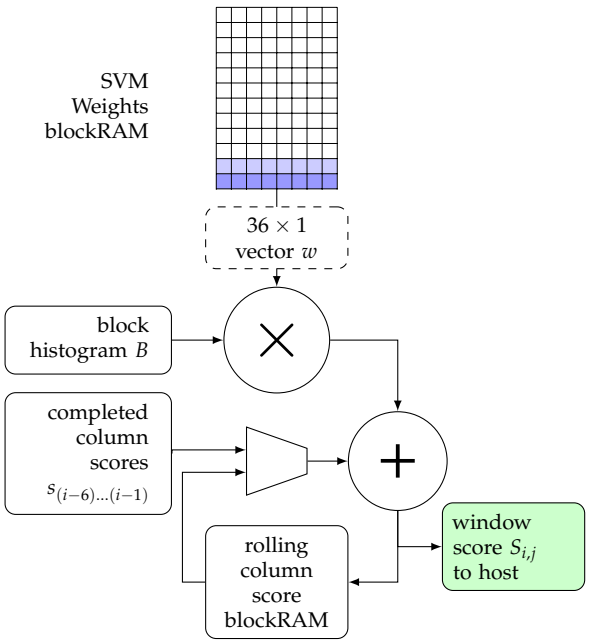


Figure 5.8: C_i, one of eight block classifiers in a stripe. A block histogram is evaluated against each weight vector in turn and the result is accumulated into the classifier blockRAM. On readout, columns are summed together before adding completed column scores from the stripe to the left.

setting $g_x = -g_x$ and $g_y = -g_y$, then select bin B_0 if $b_0 \times g_x \leq g_y < b_1 \times g_x$ *etc*.

Cell Histogram: Once an angle bin for a pixel is selected, the pixel's magnitude is then added to the relevant bin for that cell. Weighted voting into several angle bins is omitted; this simplifies the calculation and allows re-use of cell histograms in adjacent blocks. The cell histograms are stored in accumulators and, after every 8 rows, are either read out to host memory or passed to Addressable Shift Registers (ASRs) within the block normaliser (stage iv in Figure 5.7). As each stripe operates on 8 cells, we need to retain ten cell histograms (eight from row i and two from $i-1$) in the ASRs at any one time to generate a block histogram.

5.2.2 Window Classification Operations

Block Histogram: All cell histograms in each stripe are loaded from the ASRs filled in the previous stage, and then normalised. Some of the overlapping blocks span two stripes so cells are shared between stripes when necessary, as shown by the transfers in Figure 5.7. The normalisation logic is shared between all

blocks in a stripe due to its complexity. The L1-norm is taken instead of the capped L2-norm:

$$bn = \frac{bv}{\sqrt{|bv| + \epsilon}}$$

This only requires one division and square root, instead of two square-roots, divisions and dot products.

SVM evaluation: For histogram classification, 16 rows of cells must be retained for each window of $7 \times 15 = 105$ blocks; this is an impractical amount of data to store in blockRAMs. We avoid this problem by normalising the cells into the appropriate block histograms B, immediately classifying that block histogram against each of the 105 overlapping windows it belongs to, then discarding that block and retaining only the 105 partial sums. This is done in the block classifiers within each stripe (Figure 5.8).

For a window at location $m_{i,j}$, seven partial sums $s_{i-6}, s_{i-5}, \ldots, s_i$ representing columns made up of blocks $m_{i-6}, \ldots, m_i$ will be stored in the individual blockRAMs of classifiers $C_{i-6}, \ldots, C_i$. These will be updated as each new row of blocks is processed. Once all rows forming window $m_{i,j}$ have been processed, s_i will contain partial sums for block columns $B_{i,(j-15,j-14,\ldots,j)} \cdot w$ and so on. Columns $s_{i-6,\ldots,i}$ are summed in C_i to form a window score $S_{i,j}$, which is then transferred to the host. However, because the sliding windows overlap vertically and horizontally, the RAM in C_i also contains partial sums for all windows which contain the location $m_{i,j}$. As before, these are read out as each window finishes, and transferred from the preceding stripe F_{n-1} if necessary. Thus, instead of sliding an image window through a classifier, we evaluate all elements in the support vector for each new block then gradually sum the results as new blocks are presented. This requires one blockRAM and one embedded multiplier for each row of blocks in a stripe, plus an extra blockRAM per stripe to hold classifier weights. Normalisation of 8 blocks per stripe and the 3780 multiplications required by each block must be completed before the next cell histograms arrive eight rows later; this prevents the FPGA version from being scaled down to smaller image widths, but can work on larger image widths by adding more stripes.

The cell histogram pipeline is relatively short, and for *e.g. cfc* can output histograms after seeing only eight rows of image data. The window classifier requires 128 rows

Table 5.2: Resource Utilisation for HOG application and PCIe link logic on FPGA.

Resource	Percentage Used
Registers	33%
LUT	56%
Slice	81%
BlockRAMs	23%
Embedded Multipliers	18%

before it is fully filled, although after this point it generates a row of window scores for every eight rows seen in the image. Finally, the output data is converted to single-precision floating point format for easier processing on the host, concatenated into 128-bit words, and fed into the egress FIFO.

The application logic was clocked at $200MHz$ and synthesised using Xilinx ISE 13.4. Resource use is shown in Table 5.2.

5.3 Software and System Implementation Details

The software implementations of both CPU and GPU rely on the `HOGDescriptor` class in the OpenCV library. This corresponds closely to the algorithm steps given above, with the main difference being that instead of conceptually following a sliding-window model, gradients then block histograms for all areas in the image are computed at once. Block histograms corresponding to each part of each window are then selected from the resulting matrix.

For the *cfc* and *gfg* versions, OpenCV was modified to process histograms which were generated on the FPGA using the system described in §4.2.2. This required reading in the cell histograms, rearranging them into blocks, and passing them to the existing normalisation and classification code. The *cfc* version was then speeded up further using SSE commands for the SVM classification stage. *Cff* and *cfc* are multithreaded, with one thread spawned for each image scale in a frame. This allowed the CPU to process window evaluations for *e.g.* scale n while the FPGA was generating histograms for scale $n + 1$; mutexes were used to prevent multiple

simultaneous accesses to the FPGA. For *cfc* and *gfg*, we normalise with the original clamped L2-norm method.

The FPGA is configured at system boot time from on-card flash memory. Dynamic FPGA reconfiguration is not used, and algorithm selection (histogram or score outputs) is done by toggling a flag at the same time as the DMA request is made. For data transfer to and from the FPGA, the DMA transfer is triggered by the FPGA driver as described in §4.2.2. Using this processing model, the FPGA acts as a stream processor, moving data directly from host memory, operating on it, and returning the results to the host all in one operation. This avoids a separate data transfer step, so the time taken to complete the operation is governed by application processing not transfer time. The CPU periodically checks a completion flag, which is set when output data from the FPGA is finished being transferred into main memory. Depending on the version, this can then be transferred to GPU memory for further processing. GPU kernels may then be launched in the standard manner by the CUDA driver.

In summary then, *ccc* and *ggg* are unmodified from OpenCV and closest to the original algorithm. *Cfc* and *gfg* omit the magnitude voting into histogram bins, and *gff* and *cff* further simplify the normalisation technique.

5.4 Classifier Training

Classifier training was done in the manner described in Dalal's thesis [78]. The classifier was trained on positive and negative examples from the INRIA dataset[1]. This was in two stages: first on positive images and a selection of negative windows, and then re-trained using all negative errors. As with [78], this used a modified version of SVMLight[2] for training. Two sets of SVM weights were generated, one for *cfc* and *gfg* and one for *cff* and *gff*. (The *ccc* and *ggg* versions used the weights from [77]). Multiple linear support vectors were condensed to form a single support vector w with $l = 3780$. Training was accelerated by performing window evaluations on the target architecture, but parameter calculation was performed on the host, unlike the method described in [65].

[1] Available from `http://pascal.inrialpes.fr/data/human/`
[2] Available from `http://pascal.inrialpes.fr/soft/olt/`

5.5 Results

Two separate tests were performed, one to evaluate algorithm accuracy and another to evaluate performance for processing time and power consumption. Classifier accuracy was measured on the test images in the INRIA dataset, and performance was measured using several video clips containing one to three pedestrians at medium range, taken from the Thales camera described in §3.2.2.

Dollár notes that HOG detects pedestrians at around 96 pixels in height, placing them at 20m or more away from the camera [82]. Scaling allows pedestrians much closer than this to be detected, at the expense of detection runtime. (Detection of pedestrians of less than 96 pixels in height is discussed in Chapter 6.) The application we aim to target involves detection at a distance (given that we are interested in images from a camera mounted on a vehicle), thus we evaluate performance at $n = (1, 3, 13, 37)$ scale levels, scaling the image by $1.05\times$ between each level. This is sufficient to detect pedestrians of 96 up to 105, 170 and 560 pixels in height respectively.

5.5.1 Performance Considerations

Time and power measurements were taken for the system described in Chapter 4. In addition, these tests were repeated using a smaller GPU, a Quadro 2000 with 192 CUDA cores. This was done to extend the analysis to include decisions which can be made during the system design phase (design-time) rather than only at run-time: (for example, for a given power budget, is this best distributed between *e.g.* a single large GPU, or one GPU and one FPGA?).

Using this system, we performed pedestrian detection on a $w = 1024 \times h = 768$ video at our chosen scale levels ($n = 37$ is the maximum number of scales for this size of video). Overall processing times for each version are given in Table 5.3. From these, the FPGA versions are fastest, taking $5.09ms$ at $n = 1$ (*cff* and *gff* are identical as no scaling is done at this level). In these cases, most of the time taken is spent moving the image data through the FPGA (1024×768 pixels at $200MHz = 3.91ms$). Performance of the heterogeneous version is comparable to that of the *cff* version until large n_{scales}, while for $n > 3$ the *ggg* version is consistently faster.

Table 5.3: Mean processing time (milliseconds) when running HOG on 1024 × 768 video using a NVIDIA GTX560 GPU. (Labels defined in Figure 5.5. See §5.6.1 for kernel SVM details.)

	Processing time (ms)			
Linear SVM	**1 level**	**3 levels**	**13 levels**	**37 levels**
ccc	151.6	290.1	678.0	867.1
cfc	52.3	101.7	288.1	453.9
cff	**5.10**	**13.9**	57.1	108.5
gff	**5.09**	16.4	59.6	122.2
gfg	7.82	24.7	87.8	166.1
ggg	5.90	16.4	**42.7**	**65.1**
Kernel SVM				
gfg	2200	6540	28280	81180
ggg	3540	9570	25700	35345

Table 5.4: Mean processing time (milliseconds) when running HOG on 1024 × 768 video using a NVIDIA Quadro 2000 GPU.

	Processing time (ms)			
Linear SVM	**1 level**	**3 levels**	**13 levels**	**37 levels**
ccc	153.0	288.0	672.0	873.1
ggg	13.4	36.2	98.6	140.0
cff	**5.1**	**13.8**	**58.5**	**111.5**
gff	**5.1**	16.6	60.0	121.4
gfg	11.9	37.1	129.4	236.8
cfc	51.0	102.8	282.1	430.0

Table 5.4 shows processing times for the system using the Quadro 2000 GPU; the *gfg* and *ggg* versions are considerably slower.

Figure 5.9 shows processing times of individual algorithm stages and transfers between them, at the original image size and the 37th scale, where the image is around 128 pixels high. In Figure 5.9b, the *gff* version appears faster than *cff*, and this is true at a single, 37th scale. However, when run over multiple scales, *cff* is multithreaded, so the FPGA processing time dominates. This accounts for the shorter *cff* times seen over multiple scales in Table 5.3. The "FPGA histograms" and "FPGA scores" bars in Figure 5.9 also include transfer to and from the FPGA. There is no discrete "data transfer to or from the FPGA" step; after the ingress buffer is

Table 5.5: Idle and load power consumption using ML605 FPGA and GTX560 GPU.

(a) System power consumption when idle using ML605 FPGA and GTX560 GPU.

CPU	GPU	FPGA	Power Consumption (W)
idle	idle	off	129
idle	idle	on	147

(b) System power consumption for each execution path at 1, 3, 13 and 37 scaling levels.

Version	FPGA	Power Consumption (W)			
		1 level	3 levels	13 levels	37 levels
ccc	off	156	172	170	171
ccc	on	180	191	191	191
ggg	off	170	180	198	201
cfc	on	180	185	189	187
cff	on	181	175	185	184
gff	on	**170**	**170**	**179**	**179**
gfg	on	189	190	192	194
ggg	on	193	205	215	216

filled initially, FPGA transfer time is hidden and is dictated by the rate at which the HOG logic consumes and generates data. The GPU implementation avoids this by storing each frame in its own global memory, performing on-board scaling, and only transferring individual detections back to main memory. The compute to I/O ratio when the GPU↔host link is considered is thus higher.

Power consumption of the whole PC system for each version is shown in Table 5.5b. Idle power consumption of the system with and without the FPGA turned on is shown in Table 5.5a for reference. These were obtained by using a mains power meter to measure the power drawn by the entire system. It was possible to turn off power to the FPGA but not the GPU as the latter was used for image display as well as processing. The bottom half of Table 5.5b compares each accelerated method. Consumption in *gfg* and *ggg* appears to increase with the number of scales, while power for the versions where the FPGA does most of the processing remains constant. At all scales, the *gff* version draws the least power in cases where all three processors are switched on. Table 5.6 shows the same information as an increase over baseline power consumption.

Table 5.6: Power consumption above baseline (156 Watts) for each execution path at 1, 3, 13 and 37 scaling levels.

Version	FPGA	Power Consumption (W)			
		1 level	3 levels	13 levels	37 levels
ccc	off	0	16	14	15
ccc	on	24	35	35	35
ggg	off	14	24	42	45
cfc	on	24	29	33	31
cff	on	25	19	29	28
gff	on	**14**	**14**	**23**	**23**
gfg	on	33	34	36	38
ggg	on	37	49	59	60

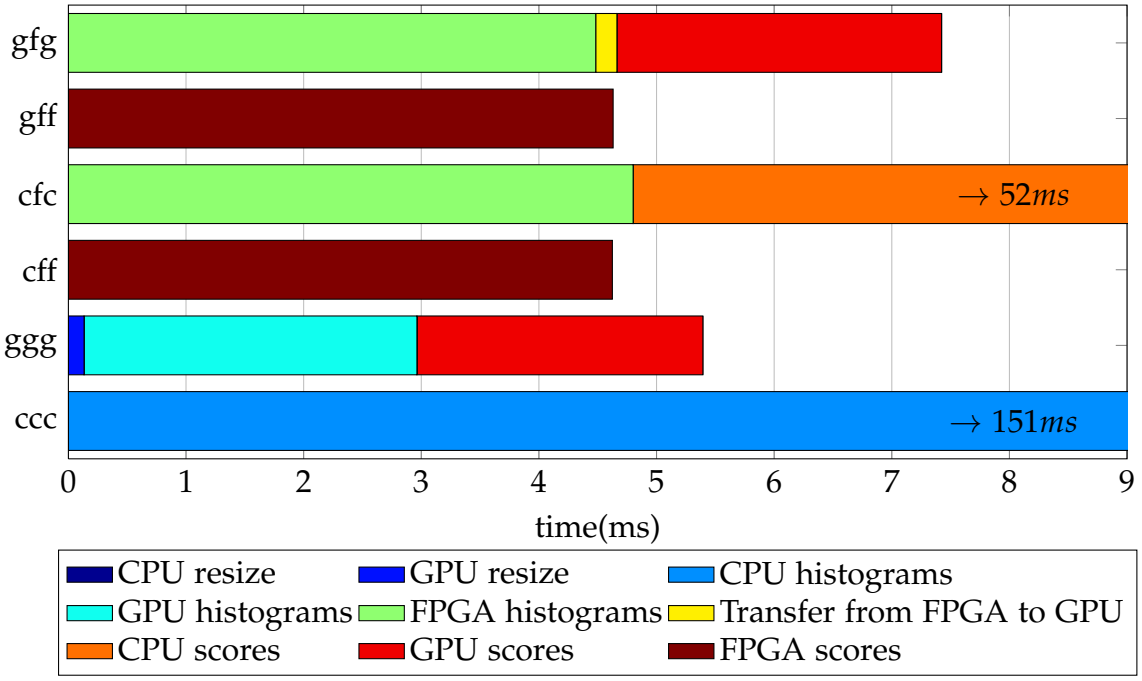
Table 5.7: System power consumption using ML605 FPGA and Quadro 2000 GPU.

(a) Idle system power consumption using ML605 FPGA and Quadro 2000 GPU.

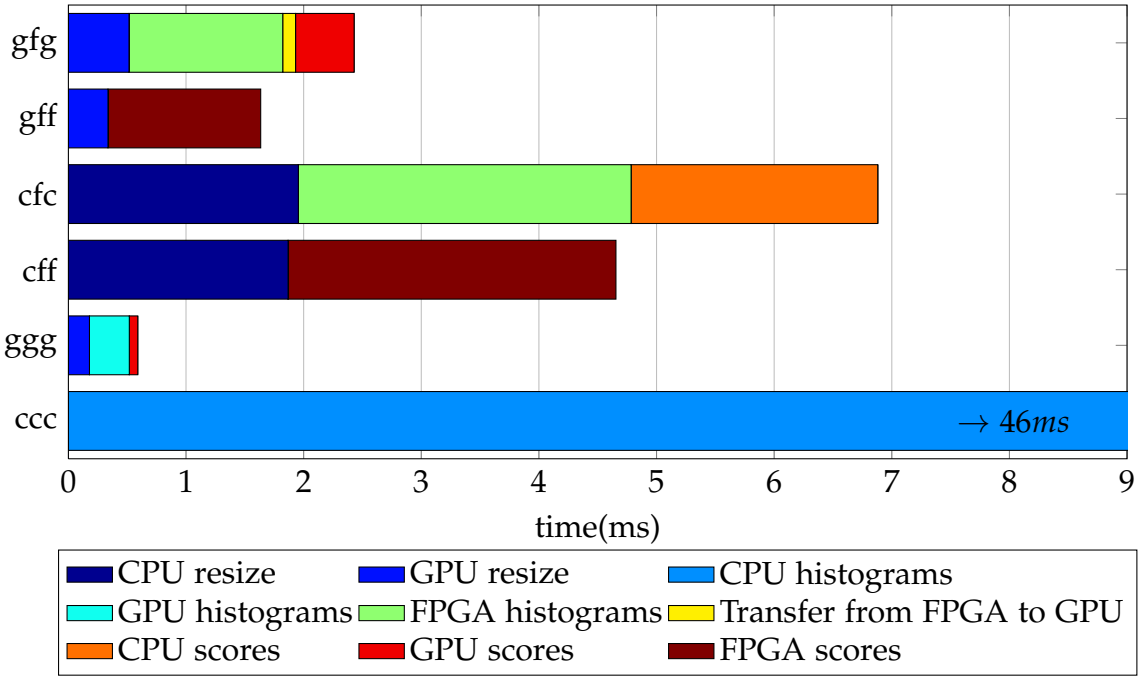
CPU	GPU	FPGA	Power Consumption (W)
idle	idle	off	77
idle	idle	on	97

(b) System power consumption for each variant at 1, 3, 13 and 37 scaling levels when running HOG on 1024×768 video using a NVIDIA Quadro 2000 GPU.

Variant	FPGA	Power Consumption (W)			
		1 level	3 levels	13 levels	37 levels
ccc	off	122	132	135	136
ccc	on	141	153	154	156
ggg	off	130	135	140	139
ggg	on	149	155	160	160
cff	on	**129**	133	**141**	141
gff	on	**129**	**131**	150	**137**
gfg	on	135	141	145	149
cfc	on	135	141	147	147



(a) Processing time (in ms) for each algorithm stage at $n = 1$ (no resizing).



(b) Processing time (in ms) for each algorithm stage at 37^{th} image scale only.

Figure 5.9: Time (in ms) spent on each algorithm stage for each version, at (a) first and (b) 37th scale, using GTX560. Transfers to and from the FPGA are contained within FPGA measurements, which also includes extra non-image data transferred to flush the buffer. Some CPU processing times are $> 9ms$.

Table 5.7a shows power consumption at idle for the Quadro 2000 system; this is considerably lower than for the larger GPU version. Table 5.7b shows consumption of the same system under load. Here the *gff* version is faster than *ggg*, while usually also drawing less power than a system with no FPGA present. Thus for a system with a small GPU, a heterogeneous arrangement of processors *always* offers both a power and speed advantage over GPU-only acceleration unless accuracy is a high priority. A given power budget is therefore better spent on a heterogeneous processor arrangement unless it is capable of including a large, fast GPU.

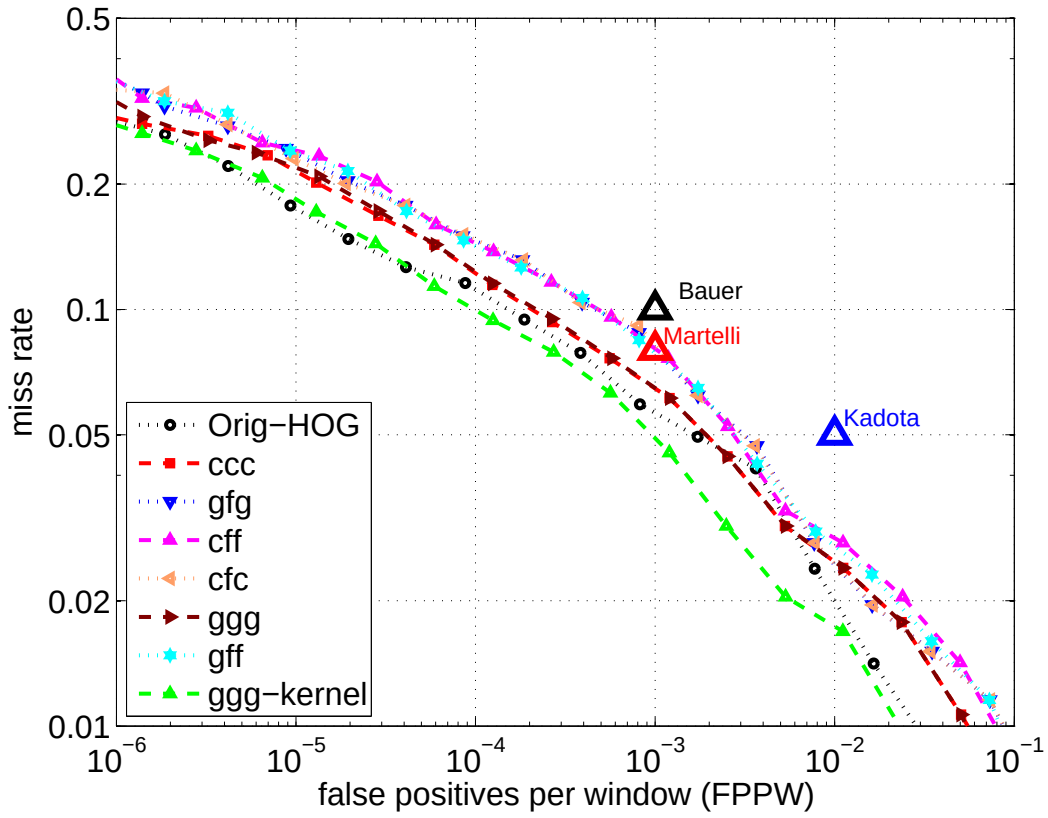
5.5.2 Algorithm Detection Performance

Figure 5.10a shows a Detection Error Tradeoff (DET) curve for the INRIA pedestrian dataset, evaluated on the basis of FPPW. All images are padded to 1024×768 before evaluation; images which are larger than this in any dimension are cropped for FPGA evaluation. The results are comparable to the original, with a slight decrease in accuracy in the *gfg* and *cfc*-versions due to simplified block weighting, and a further decrease for *gff* and *cff* due to the simplified normalisation.

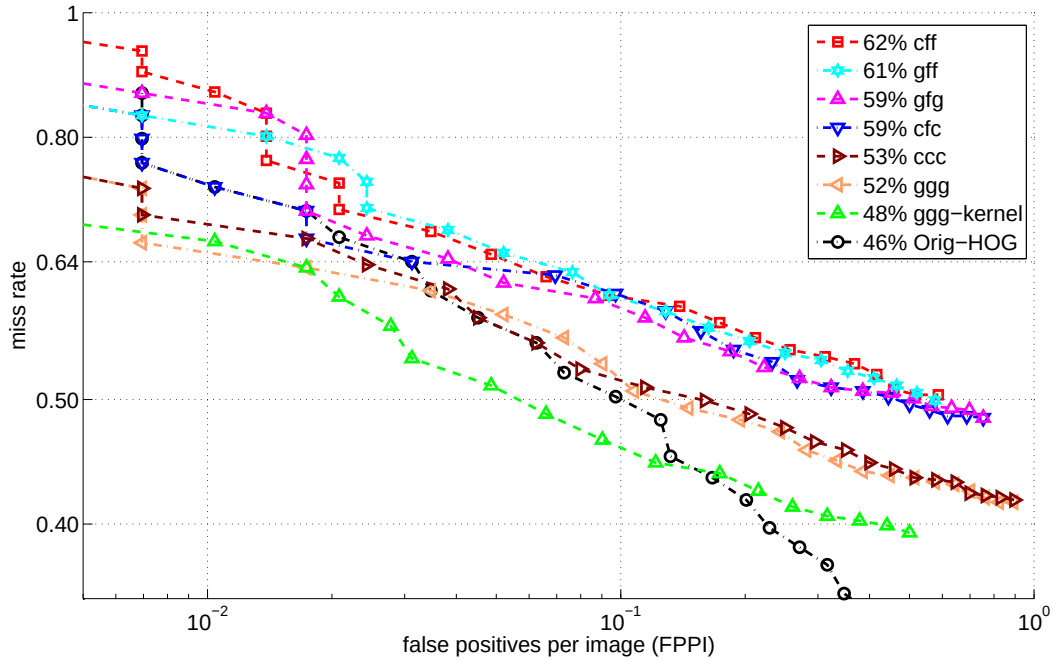
Figure 5.10b compares each of our implementations to the performance of the original HOG algorithm on the large positive test set from INRIA, using the evaluation code from Dollàr [82]; this allows comparison to other algorithms and implementations, and evaluates detector performance, including Non-Maximal Suppression (NMS) over multiple scales. On this graph the differences between our versions are more pronounced, but still have the same ranking as in Figure 5.10a. Both figures here show that as we move more processing to the FPGA, the accuracy decreases slightly; this is as expected, given the simplifications we make when implementing the algorithm here. The graph legend also includes the log-average miss rate. This is obtained by sampling the miss rate at a logarithmically-spaced set of points along the x-axis, then taking the mean of those points.

5.5.3 Performance Comparisons and Tradeoffs

Based on the accuracy results in Figure 5.10a and the power and latency information for a high-power and lower-power system given in Tables 5.3, 5.4, 5.5a, 5.5b, 5.7a and 5.7b, we now compare execution time, accuracy and power consumption of each implementation.



(a) FPPW DET curve for HOG on INRIA dataset.



(b) FPPI DET curve for HOG on INRIA dataset. Percentages in legend denote log-average miss rate.

Figure 5.10: Detection Error Tradeoff curves for algorithm implementations.

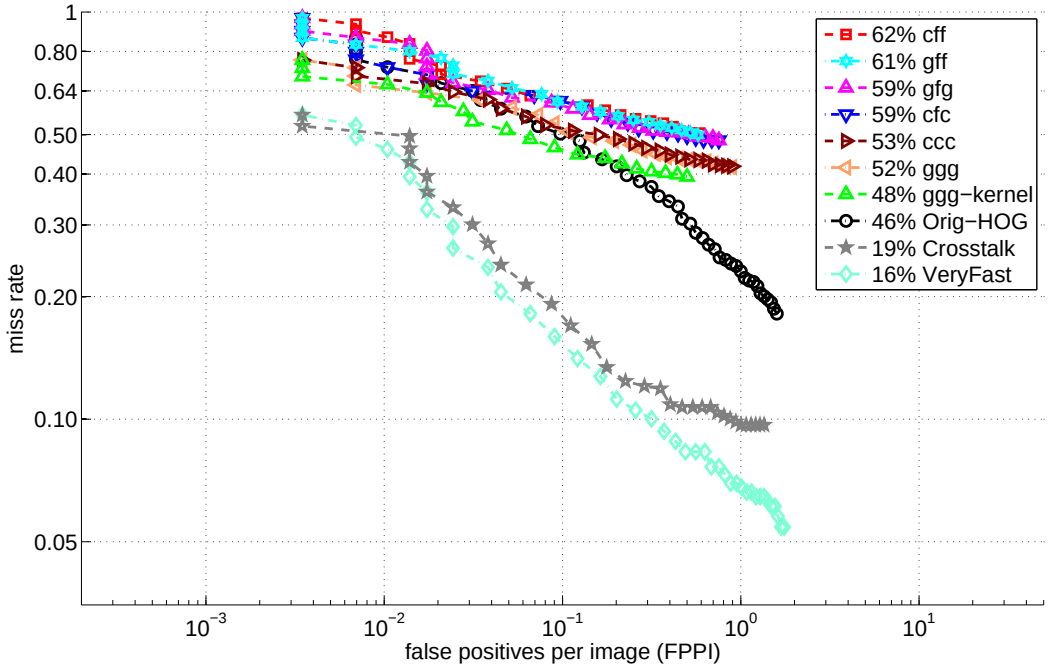
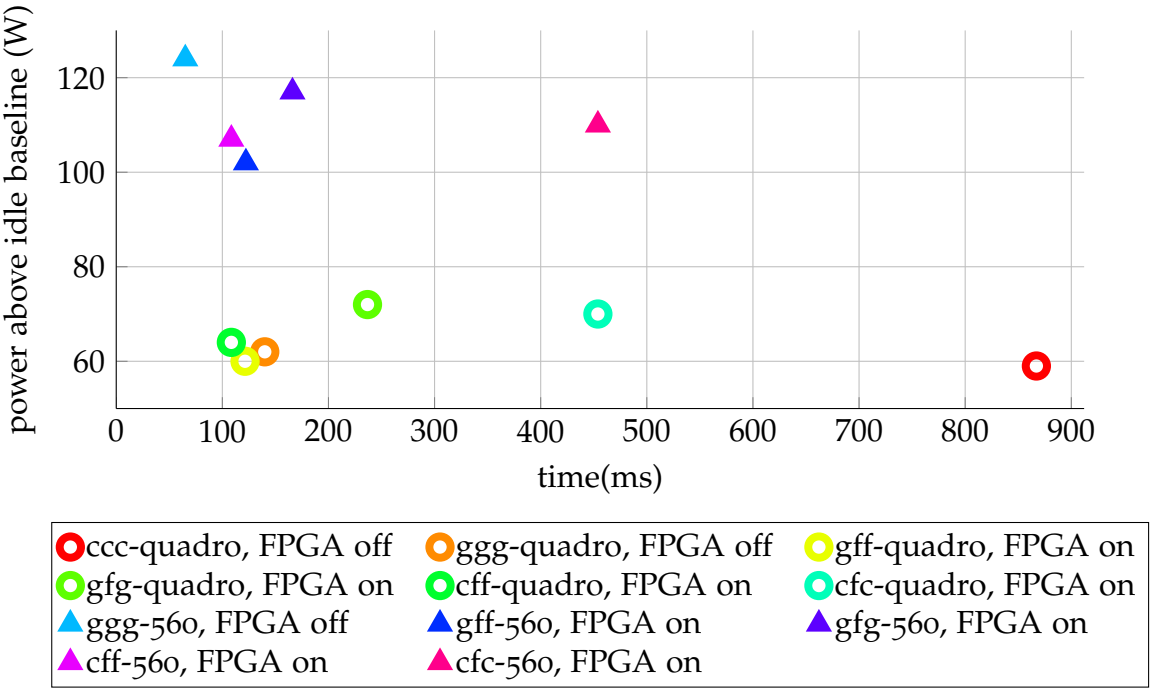


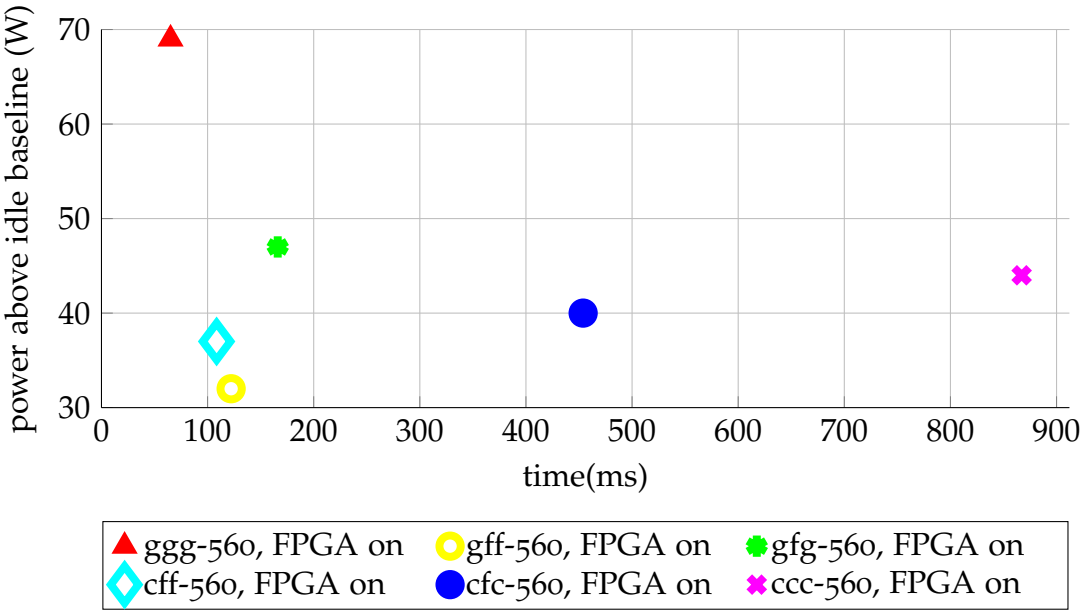
Figure 5.11: FPPI DET curve for multiscale HOG on INRIA dataset, showing comparisons to current state-of-the-art detectors (VeryFast and Crosstalk). Percentages denote log-average miss rate.

We can make a design-time choice between the two systems here; given that the accuracy of each implementation between the two systems is the same, we are only concerned with power and latency at design time. Figure 5.12a shows power consumption against the lowest idle baseline of 77W, versus processing time for both systems. Processing time for all FPGA-centric versions is largely similar but idle power consumption changes considerably, and the ranking of each option changes as we move from a low-power (Quadro 2000, where *cff* is faster) to a higher power system (where *ggg* is faster). This move from 111ms to 65ms processing time costs an extra 75W; a 42% speed improvement draws 53% more power.

As we are concerned with something which we mean to target at real-time applications, and where we would ideally perform additional post-detection processing each frame, we choose the faster, higher-power system for further analysis; this gives more opportunities to *e.g.* vary the number of scales processed to reach an arbitrary limit in FPS. Given the ability to select any of the implemented versions at any time (see §5.6.3), we can quantitatively match system priorities (processing speed, power consumption, and accuracy) to particular versions, and evaluate the costs of selecting one configuration over any other.



(a) Design-time design space exploration: power vs. time plot using small Quadro 2000 and large GTX560 GPU. Power consumption shown as use over baseline of 77W.



(b) Run-time design space exploration: power vs. time plot using GTX560 GPU. Power consumption shown as increase over baseline of 147W.

Figure 5.12: Design space exploration: processing time (ms) vs. power consumption (W) for HOG at $n = 37$. In (a) two different GPUs are tested with the FPGA turned on or off as required. (b) shows options for selection at runtime. The large GPU is used and the FPGA is always on.

Table 5.8: Choice of algorithm version and compromises for a given priority at 13 scales. Tradeoffs shown as percentage differences from their best measurement. Accuracy is measured as % change in log-average miss rate.

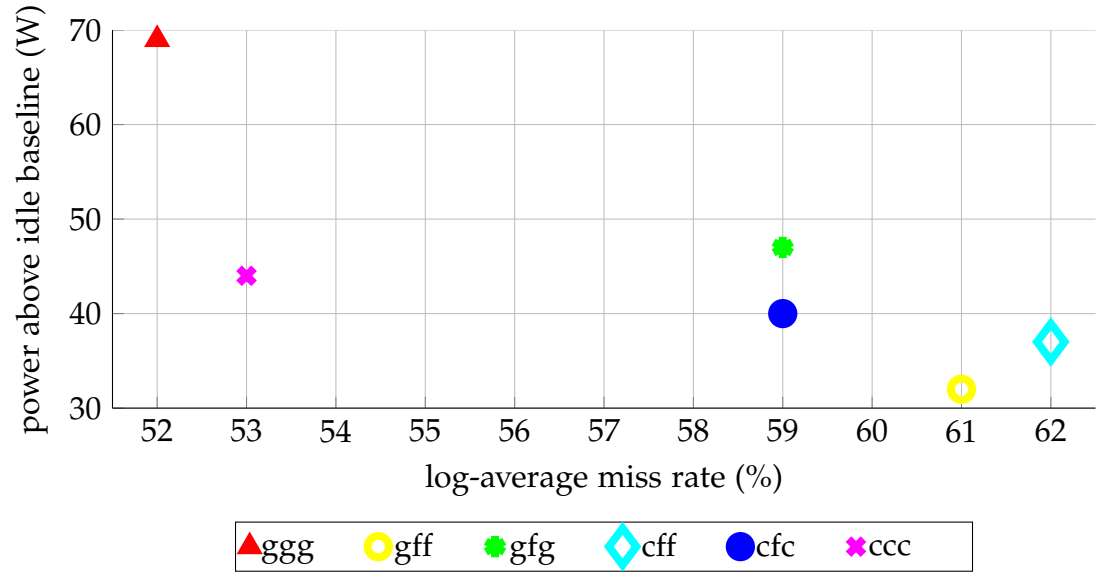
Priority	**Best Choice**	**Tradeoffs**		
high speed	*ggg*	power ↑ 20%	accuracy ↓ 4%	
lowest power	*gff*	time ↑ 26%	accuracy ↓ 13%	
high accuracy	*ggg-kernel*	power ↑ 20%	time ↑ 54600%	
power & speed	*gff*	time ↑ 26%	accuracy ↓ 13%	
accuracy & speed	*ggg*	power ↑ 20%	accuracy ↓ 4%	
accuracy & power	*ccc*	time ↑ 1290%	accuracy ↓ 1%	power ↑ 7%

The costs and tradeoffs for these three parameters are shown in Table 5.8 for $n_{scales} = 13$. Here we focus on viable implementations; we discuss the *ggg-kernel* implementation briefly in §5.6.1. Although it is the most accurate, this configuration does not have the potential to run fast enough to be usable in any scenario, so we do not consider it further here.

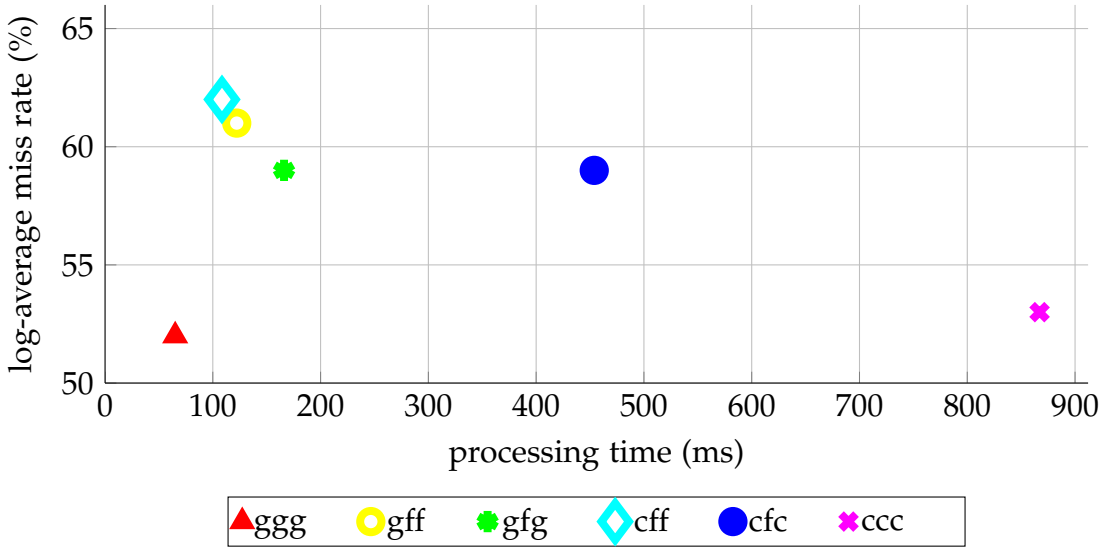
For the fastest available system we have, we now focus on its runtime characteristics. Although our design space is three-dimensional, given the difficulty of representing a very small number of widely scattered discrete points in a $3D$ space in a $2D$ medium, we discuss each set of tradeoffs separately, as $2D$ pairs.

Power consumption above idle versus processing time for runtime-selectable versions is shown in Figure 5.12b. Here, *ggg* and *cff* and *gff* form a very steep Pareto curve for these characteristics. Thus, *ggg* provides a fast, high-power version and *gff* provides a lower-power, slower alternative. For all other versions, only processing histograms on the FPGA is not Pareto-efficient.

Dollàr *et al*. discuss the *log-average miss rate* metric, which allows comparison of detection algorithms using a single number to express accuracy. This was calculated using code from [82]. This takes into account performance at various points on the x-axis seen in Figure 5.11. Figure 5.13a shows power consumption over the idle baseline, against log-average miss rate. This forms another Pareto front between *ggg*, *ccc* and *gff*, showing that the increase in accuracy offered by *cfc* and *gfg* does not offer any Pareto-efficient benefits. Figure 5.13b shows that *ggg* is always the best choice when only considering accuracy and time.

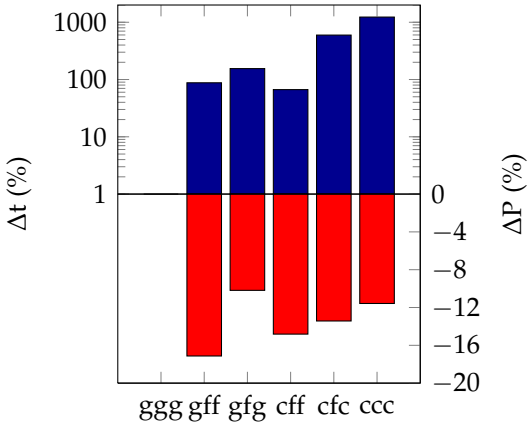


(a) Power vs. log-average miss rate plot using GTX560 GPU. Power consumption shown as increase over baseline of 147W.

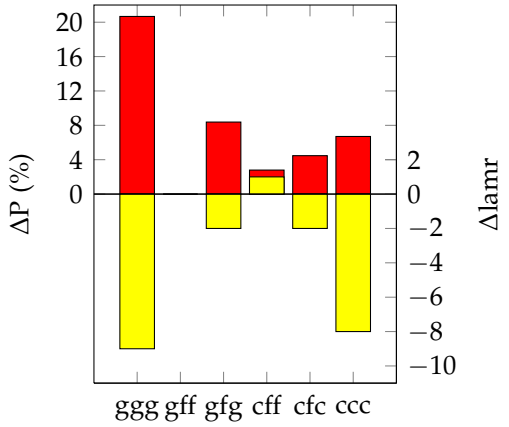


(b) Processing time vs. log-average miss rate plot using GTX560 GPU.

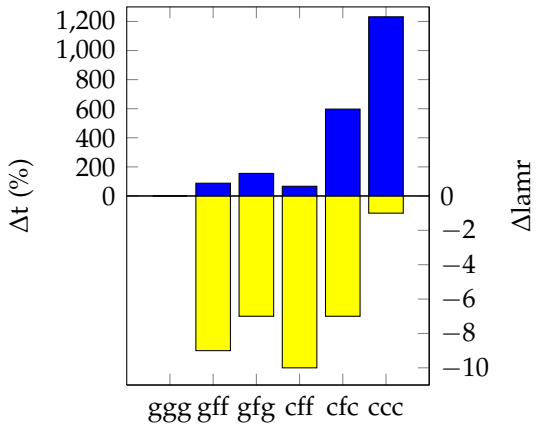
Figure 5.13: Run-time design space exploration for various pairs of characteristics.



(a) Relative change in processing time and power consumption of each configuration compared to *ggg*. A red bar below zero indicates decreased power consumption.



(b) Relative change in power consumption and decrease in log-average miss-rate of each configuration compared to *gff*. A red bar above zero indicates increased power consumption and a yellow bar below zero shows decreased miss rate (*i.e.* smaller error).



(c) Relative change in processing time and *increase* in log-average miss-rate of each configuration compared to *ggg*. A blue bar above zero indicates increased runtime and a yellow bar below zero indicates *increased* miss rate (*i.e.* larger error).

Figure 5.14: Relative tradeoffs between individual characteristics.

The charts in Figure 5.14 express this information as relative tradeoffs; Figure 5.14a shows that to get a 15% reduction in power consumption from the maximum we need a ∼80% increase in processing time (from *ggg*→*cff*). From here, a further 2% decrease in power consumption using *gff* only requires an extra 10% increase in time. Figure 5.14b shows that, starting from *gff*, to get a 8% improvement in accuracy requires 5% more power, or 20% more power to get 9% more accuracy (using total system power consumption rather than increase from idle power consumption). Figure 5.14c echoes the contents of the other time against accuracy graph; as *ggg* already outperforms other implementations for both measurements, any move away from this results in a significant decrease.

5.5.4 Analysis, Limitations, and State-of-the-Art

A comparison to the current state-of-the art detectors (VeryFast and Crosstalk) is also given in Figure 5.11. Two conclusions can be drawn from this figure: the first is the advancement made in the underlying algorithm in the eight years since HOG was first published; and the second is that modifying an accelerated HOG implementation to more faithfully match the original algorithm offers fewer opportunities for fast, accurate detections than starting with a baseline algorithm offering better performance, such as VeryFast [83] (particularly when such an algorithm already makes use of an accelerated processor such as the GPU).

There are three reasons for the decrease in performance of the FPGA seen in §5.5.1 when considering high numbers of scales. First, the FPGA implementation is not capable of performing the multiple image scaling per octave that HOG requires for accurate detection; this has been mitigated by rescaling on the CPU or GPU and padding and transferring the result. Second, the PCIE interface has a maximum size limit of 1MB for one DMA transfer: each scaled image must be transferred separately and incurs its own overhead, in the form of additional data transferred after the frame data to push remaining window scores or cell histograms through the pipeline. This means that at multiple scales, the large pipeline is emptied between each scaled image. Finally, the row buffers used by the FPGA are fixed at a standard line length of 1024; smaller images are padded by adding blank space to the right to fill each row. Thus, as image size decreases, the number of clock cycles required to process all pixels in the image scales sublinearly, unlike the effectively linear scaling seen on GPU and CPU. These effects are shown in Figure 5.9b.

The inefficiencies associated with the multiple transfers to FPGA mean that all FPGA versions run slower than *ggg* at multiple scales. In addition, for *gfg* the cell histograms must be transferred twice (from the FPGA back to main memory then to the GPU). However, the PCI-express specification allows for direct endpoint-to-endpoint transfer, allowing an FPGA to transfer data directly into GPU on-board memory. This technique is currently only possible with Kepler-class GPUs on Linux hosts [144] but is an option to consider for future work.

Various fixes could be implemented to improve performance; if we resized and processed all 37 scales in one transfer it would take around 69ms per-frame, but as Figure 5.11 shows, this time would arguably be better spent taking a more advanced algorithm to work from. The same figure shows that, in contrast to the large differences in runtime and power consumption, differences in accuracy between implementations on various platforms are relatively minor. The analysis above has used FPPI and log-average miss rate throughout, as they provide a means of comparing to other algorithms. Dollàr *et al*. [82] argue that FPPW measurements are flawed, as detector ranking can change between window and image measurements, and is affected by the NMS method used. However, for existing FPGA implementations where detection accuracy is discussed, only FPPW information is available, so we compare against that. Figure 5.10a also shows comparisons to other FPGA implementations, showing that our implementation is more accurate than three existing FPGA versions. Kadota *et al*. [86] perform HOG feature extraction on FPGA at 30FPS on VGA video, with 5% miss rate on the INRIA dataset at 10^{-2} FPPW. Martelli *et al*. [87] perform FPGA-based pedestrian detection using covariance features, achieving 20% miss rate at 10^{-4} FPPW on INRIA.

Hiromoto *et al*. [98] describe a similar system using co-occurrence HOG. They do not provide accuracy information, but evaluate a QVGA image at scale ratio $s = 1.2$ with 3615 windows per frame at 38FPS or 139166 sub-windows per second, whereas our *gff* version described above evaluates 20868 windows per frame at 13FPS (271284 sub-windows per second) for the same s. Kadota *et al*.'s implementation [86] evaluates 56466 windows on 10 parallel elements, taking 5.7μs per window. In contrast, *cff* and *gff* take 657μs to generate all histograms over a single-scale window, or a further 40μs to generate window scores, where up to 121 parallel elements are used to generate a row of scores across the image at once. This longer window period is dictated by the pixel clock and our larger frame width. In both cases, our slower

framerate is a limitation of our PCIE architecture; if the multiscale evaluation was fully pipelined, multiscale evaluation at $s = 1.2$ would take around $20ms$.

Finally, the performance information given above, the advantages of a heterogeneous system become apparent: in situations where speed and power consumption are both desired, and power consumption is the most important constraint, *gff* provides a suitable compromise, while still offering a $7.1\times$ speed-up over *ccc*. As Table 5.5b shows, *gff* requires less power than a GPU-accelerated system when no FPGA is present. As shown by Figures 5.12b and 5.14, *gfg* and *cfc* are always outperformed by other configurations in all scenarios, showing the effect of increased communications delays between processors. This is confirmed by the transfer overhead graphs in Figure 5.9: in particular, in Figure 5.9b, although the resize step for *gff* on GPU takes considerably less time than on CPU, the extra transfers required means that *cff* is faster overall. The power consumption analysis for runtime switching done in §5.5.3 is done on the assumption that all processors are running constantly and cannot be powered down when not in use. Again, this is a limitation of our PCIE implementation.

One could speculate that, as moving more algorithm stages to FPGA increases speed and reduces power, a worthwhile comparison would be to include an all-FPGA implementation. Until now we have only considered run-time characteristics, but at this point we would have to include increased development time as another element in the design stage exploration process; at this stage, again, the time would be better spent implementing an algorithm which does not require implementation at multiple dense scales.

The analysis here also assumes that we always evaluate over a fixed number of scales. If our target application involves hypothesis generation via motion or infrared hotspot detection, we would confirm pedestrian detections at a certain pixel height rather than performing exhaustive evaluation over n scales. For detections closer to the camera than this, we aim to use motion-cued detection on a smaller region as this considerably reduces the number of scales which are required to process an image. However, the tradeoffs do not change significantly unless $n_{scales} \leq 3$, where *cff* is slightly faster.

5.6 Variations

The previous section describes the main body of tests which were performed on the heterogeneous system; the conclusions drawn from Section 5.5 were the main factor influencing the selection of algorithm implementations used in Chapter 6. This current section details various other tests which were performed on the system, but which were not found to improve performance enough to be considered for further use.

5.6.1 Radial Basis Function Kernel Classification

Based on the suggestion of a journal paper reviewer, the linear classifier in the SVM was compared with a RBF kernel. To generalise, RBF kernels offer increased accuracy at the expense of runtime, as noted by Dalal [78]. A variation of this method for classification in HOG was used by Bauer *et al.* [71], where image histograms were generated on FPGA; however, motion detection was used to select a maximum of 1000 windows per frame to evaluate.

SVM classification with a linear kernel on a feature vector $\mathbf{x}$ is shown in Equation 5.1. The RBF kernel instead uses:

$$s = \sum_{i=1}^{n_{sv}} (\alpha_i \cdot y_i \cdot \mathbf{K}(\mathbf{w_i}, \mathbf{x})) + b \tag{5.2}$$

where the kernel $\mathbf{K}$ is:

$$\mathbf{K(w,x)} = \exp\left(-\gamma \|w - x\|^2\right) \tag{5.3}$$

Around 4000 support vectors were generated during training. Using a linear kernel, it is possible to condense all generated support vectors into one vector w by summing the weights, but this is not possible with RBF kernels.

RBF kernel evaluation was implemented on GPU, and both the *gfg* and *ggg* algorithm versions were evaluated using this kernel. Weights were generated as described in Section 5.4. The reviewer had also requested an implementation and analysis of this classifier on FPGA. However, this was judged not to be feasible based on the quantity of calculations required to produce a result in real time over the whole frame. Examples do exist in the literature of RBF kernels implemented on FPGA,

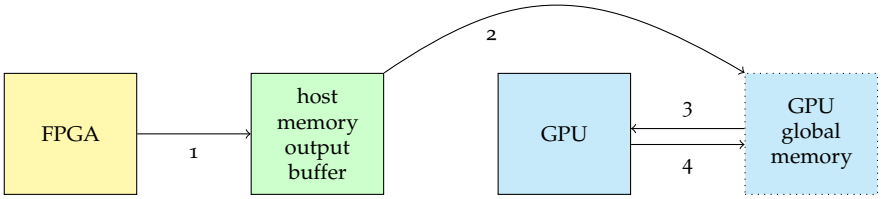
such as Irick *et al*. [145] which works on small windows, or Cadambi *et al*. [146], which classifies at 14×10^9 Multiply-Accumulate Operations per second (MAC/S). However, if we were to use every optimisation listed in [146] (such as making every calculation with 4-bit precision in order to pack multiple calculations into one hardware multiplier, and double-clocking the multipliers[3]) *and* assuming we were able to use all 768 multipliers on the FPGA, we would still take around 700*ms* to classify a 1024×768 image at one scale, and several seconds to classify a whole frame over multiple scales. The relatively long processing times for *gfg-kernel* and *ggg-kernel* in Table 5.3 confirm this; again, this delay is mostly due to the large number of support vectors which must be evaluated.

Note that this version differs from Bauer *et al*.'s [71] in that we evaluate the whole image and do not generate motion-cued hypotheses to reduce the number of candidate windows to be evaluated. As discussed in Chapter 6 we use motion detections to target both histogram extraction and classification to specific regions instead. For an application with the goal of real-time operation in mind, these processing times are arguably not useful, and so we do not consider RBF SVM classification outside of this section.

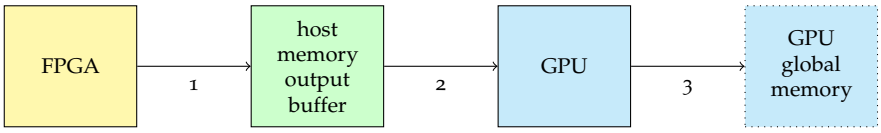
5.6.2 Transfer Tests with Pinned Memory

A test was also performed to ascertain whether performance could be improved using pinned memory. This is a GPU memory optimisation technique whereby an area of page-locked host memory is defined as being directly accessible from the GPU, and is mapped into GPU global memory [147]. In this mode, each running kernel fetches data from the pinned memory area on-demand, rather than including a explicit step of copying data to GPU global memory before the kernel is run. Thus, in the *gff* and *gfg* stages, it appeared possible to reduce the transfer delays when moving FPGA data to GPU by removing a copying step. The memory region which was pinned was the output buffer which held data returned from the FPGA, as described in §4.2.2. This was allocated and marked as page-locked by the FPGA driver at initialisation, then mapped into GPU memory. A schematic is shown in Figure 5.15. As this would have affected mainly the *gfg* and *gff* versions, processing time results for these variants only are given in Table 5.9.

[3]This may already be infeasible given that our application logic base clock is 200MHz rather than Cadambi's 133MHz.



(a) Non-pinned memory transfer.



(b) Pinned memory transfer.

Figure 5.15: For non-pinned transfers, data is written by the FPGA(1) into host memory, then read by the GPU(2) into GPU global memory, loaded from global as it is processed(3), then results written back to global(4). For pinned transfers, data is written as before(1) then read straight from the host by the GPU(2), processed, then the results are written to global memory(3).

Table 5.9: Mean processing time (milliseconds) of HOG for 1024 × 768 video while using pinned and non-pinned memory.

Variant	Pinned?	Processing time (ms)			
		1 level	3 levels	13 levels	37 levels
gfg	no	7.82	25.4	111.0	303
gfg	yes	8.60	27.8	129.0	341
gff	no	4.88	17.4	78.5	196
gff	yes	5.10	21.0	79.8	231

From these results, it becomes apparent that pinned memory does not offer an advantage in this case; at 37 scales, the pinned *gfg* version is 10% slower than non-pinned; *i.e.* a separate copy step to GPU memory is faster than allowing each kernel to load data from the FPGA output buffer as required. Based on these results, non-pinned memory was used throughout the other experiments.

5.6.3 Version Switching

Switching times between different versions were also measured to determine if any noticeable delay was incurred due to version switching. Processing times were averaged for transitions between both version $a \rightarrow a$ (same) and $a \rightarrow b$ (different)

Table 5.10: Processing times for different HOG versions on a 1024×768 image, where the version used to process timestep $t = t_n$ was either the same as at $t = t_{n-1}$ (column 2) or different (column 3). The final column shows times for the $a \rightarrow b$ transition as a percentage of the $a \rightarrow a$ transition.

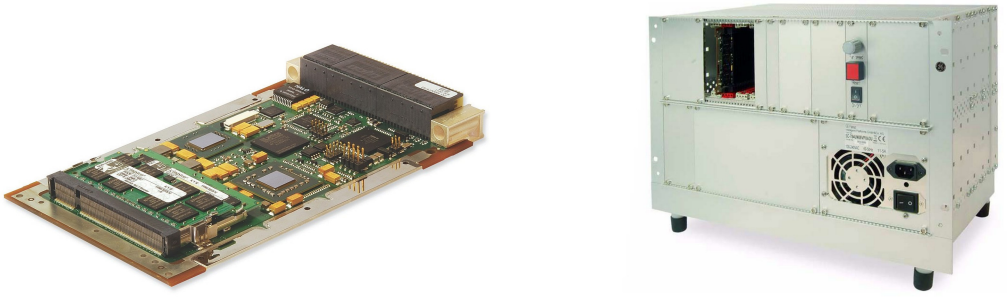
Variant	Processing time (ms) for transition: **a → a (same)**	**a → b (other)**	**Time for a → b, normalised to a → a** (%)
ccc	776.6	765.1	98.5
ggg	55.5	55.7	100.3
gfg	119.1	123.1	103.3
cfc	646.4	640.1	99.0
cff	96.6	95.1	98.4
gff	85.5	88.2	103.1

versions at the $t = t_{n-1}$ to $t = t_n$ timestep. These are shown in Table 5.10. As shown by the final column of this table, when switching between two versions, all processing times are within 4% of times when not switching. This means that, when constructing a cost function to determine which implementation to select at any point in time, we do not need to assign a cost to the time taken to switch between different versions, and need only consider scene content and current system priorities.

5.6.4 Evaluation of Embedded Implementation

In order to test performance of this algorithm partitioning method on an embedded system, an evaluation was performed on ruggedised hardware suitable for installation in a vehicle for image enhancement and surveillance tasks. The embedded system consisted of a mobile Intel Core2 processor on a Single-Board Computer (SBC), mounted on a 3U-high OpenVPX board. This was connected to another OpenVPX board containing a NVIDIA mobile GPU via a backplane, which allowed a connection via PCIE. The SBC and OpenVPX housing are shown in Figure 5.16. The GPU was a compute capability 1.2 GT240 with 96 cores, an older generation than the system described in Section 3.3. Both of these boards were ruggedised, and had variants which were certified for use over the full military temperature range and required only external convection cooling.

A schematic is shown in Figure 5.17. The backplane also had breakout connections for external PCIE cables, to allow links to other equipment. As no OpenVPX FPGA board was available, the existing ML605 development board was connected to the



(a) Single-board computer. (b) OpenVPX card housing and backplane.

Figure 5.16: Single-board computer used in embedded system. OpenVPX backplane connection shown at top right.

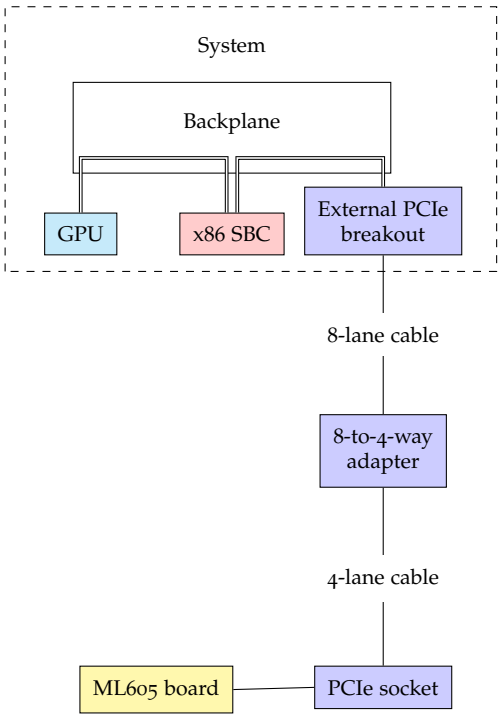


Figure 5.17: Embedded system connected to ML605 over PCIE. Packet loss suspected between system breakout and socket.

system via a PCIE extender: essentially, a PCIE socket connecting to the development system via an external cable and an 8-way to 4-way adapter. The extender was limited to a PCIE 1.0 4-lane connection, providing a theoretical bandwidth of 1 GB/S. As the application logic consumed data at 200 Megabytes per second (MB/S), this should in theory have been more than sufficient to keep the input buffer filled. However, when the system was tested, it was found that most of the data transfers to the FPGA failed or timed out, meaning that histograms for each frame were not generated, as the FPGA stalled because the pipeline emptied. As described in §4.2.1, the DMA controller had very limited capability to re-request lost packets. Due to this problem, detections could not be generated for any of the implementations using the FPGA. The cause for this problem was believed to lie with one of the external PCIE cables, the socket or the adapter, but it was not possible to definitely confirm this or to work around it. All other performance evaluation in this project was therefore done on the desktop system described in §4.2.1.

5.7 Conclusion

This chapter has described the implementation and analysis of a well-studied object detection algorithm across multiple architectures. Recall that our motivation here was to answer the question posed in the Introduction and §5.1.2: *"how does the performance of an algorithm when partitioned temporally across a heterogeneous array of processors compare to the performance of the same algorithm in a singly-accelerated system?"*. We analysed the calculations performed and volume of data generated at each stage of the algorithm and used this to temporally partition HOG onto multiple heterogeneous processors, writing our own implementations where necessary. Having described the implementation methods for each platform, we evaluated this running on a system using a low-power and high-performance GPU separately. The GPU-only version proved to be the fastest and most accurate overall, while the *gff* version consistently drew the least power, even less than the reference CPU implementation. The versions with more transfers associated with them were significantly slower than the others, showing that data transfers between processors have a significant impact on performance.

The main outcomes from this chapter are as follows. Given this set of algorithm implementations, we can state which implementation to select to best match a particular priority (high speed, low power, or high accuracy) and the tradeoffs associated with this decision, expressed in terms of other characteristics. We also demonstrated that switching between different implementations in this system incurs no time penalty.

In the next chapter, we move up one level of abstraction and consider the tradeoffs between FPGA and GPU implementations at the task level when given a specified characteristic to prioritise. Following this we evaluate how this affects overall performance in a real-world computer vision task.

6. Task-Level Partitioning for Power-Aware Real-Time Anomaly Detection in Video

In the previous Chapter, we implemented an object detection algorithm in multiple arrangements across heterogeneous architectures in a system, and tabulated its performance in terms of processing time, power and detection accuracy, making use of the system architecture described in Chapter 4. In this Chapter, we now move up a level of abstraction and apply our knowledge to a real-world image processing problem; that of performing anomaly detection in video.

In doing so, we return to the grand aim of this thesis; building then evaluating the performance of a system able to adjust its performance characteristics in response to events in the environment it monitors. This applies mainly to the mobile vehicle surveillance scenario described in Chapter 1, where the electrical power available and system performance priorities will change over time. In any case, we wish to balance conservation of power against the ability to obtain fast scene understanding in uncertain or possibly threatening situations. This is a broader concern for any electrical system, battery-powered or otherwise.

A paper based on an abridged version of this chapter was presented at the 2014 VISAPP International Conference on Computer Vision and Applications.

6.1 Introduction

This chapter describes the components and behaviour of a system for performing power-aware real-time anomaly detection in video. This uses various accelerated

algorithms running on one or more platforms in a system of heterogeneous processors, and uses the presence of anomalous objects or behaviour in the video to prioritise power, speed or detection accuracy — and hence the set of algorithm implementations to use — at any particular time. In other words, we quantify the level of anomaly in a scene and alter the processing used in response. From this we evaluate the performance of a heterogeneous processing architecture compared to a single-accelerator model. We also compare performance of this system when running with dynamic mapping between algorithms and processors, against static mapping.

Understanding power consumption, and particularly the power and speed trade-offs or changes associated with using one heterogeneous processor over another, is an important, if under-represented, problem worth studying. The quantitative information given by such a study allows systems with greater endurance, intelligence and autonomy to be built. In any embedded system, Size, Weight and Power (SwaP) are the limiting factors, and reducing power consumption affects all of these metrics. As we discuss in Section 6.3, this is not addressed by current work in this area.

The high-level structure of this system is shown in Figure 6.1. The rest of this chapter describes each component in that diagram. We provide details of the dataset in Section 6.2, and summarise related work and define the task in Section 6.3. Section 6.4 describes control flow in the high-level algorithm. Section 6.5 describes each of the algorithms used throughout the system and provides examples of the performance of each. The 'detection algorithms' step in this Figure relies on the system architecture described in Chapter 4 and the accelerated pedestrian detection algorithms described in Chapter 5. As we are not using state-of-the-art object detectors, many of the errors observed in the final output can be attributed to individual detector errors at this stage, so we discuss example errors alongside each detector. That section concludes with a description of the reasoning methods used to obtain an anomaly level for every frame. In Section 6.6 this measure is used to determine which of our three characteristics (power, speed and accuracy) to prioritise, and by how much. It also describes how the detectors for use in the next frame are chosen. Section 6.7 gives testing methodology, and is followed by results in Section 6.8. The Chapter concludes with analysis of the outcomes and a summary in Section 6.9 and Section 6.10 respectively.

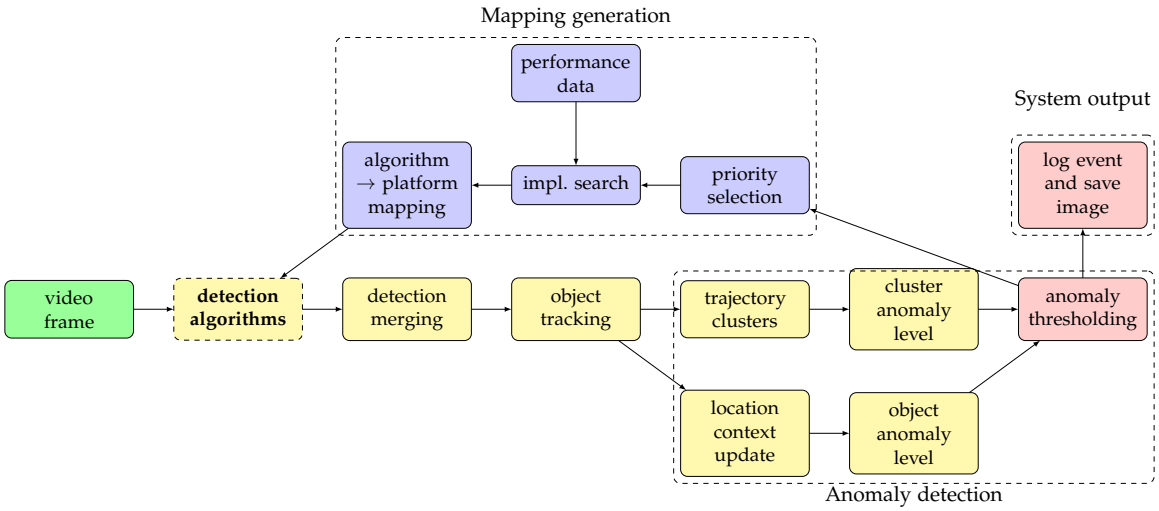


Figure 6.1: Frame processing and algorithm mapping loop in anomaly detection system. Figure 6.3 shows an expanded version of the "detection algorithms" block.

At this point, we note that the FPGA version of the histogram generator in the car classifier described in this chapter (§6.5.2) was done by Scott Robson, a summer student at Thales Optronics, specifically for this project (Specifically, the FPGA component of the *cfc* and *gfg* detectors). It is a modified version of the FPGA implementation of HOG described in Chapter 5, with parameters changed to perform car detection and with the structure altered to reduce device area. This work was not done by me, but I integrated it into this system and it is used as part of the processing pipeline. Power, accuracy and latency results of this FPGA car detector are therefore reproduced in this chapter, to allow for its performance to be understood in the context of this system.

6.2 Datasets

One of the possible applications envisaged for the system described in this thesis is as a real-time processing system onboard a vehicle. Data collection from a moving vehicle of video sequences featuring other traffic participants behaving anomalously presents various operational and safety concerns. Indeed, detection of pedestrians from a moving platform, particularly when pedestrians are occluded or far away, is still regarded as a difficult problem and is the focus of ongoing research efforts [82].

(a) BankSt Scenario (b) Parked Vehicle Scenario 3 (PV3)

Figure 6.2: Sample images with traffic from each dataset used.

Adding a real-time processing constraint to this list adds yet more complexity. We reduce the complexity of this problem somewhat by using two video scenarios gathered from a static camera. Both datasets are described below.

6.2.1 Bank Street Dataset

The *BankSt* scenario involves a static camera overlooking a busy four-way intersection in Glasgow, taken in August 2013. This was recorded using a consumer point and shoot camera at 1280×720 and 30FPS, then downscaled to 700 pixels wide. Video quality is generally good, and there is little camera shake. A sample image is shown in Figure 6.2a.

6.2.2 i-LIDS Dataset

Imagery Library for Intelligent Detection Systems (I-LIDS) is a dataset collated by the UK Government's Home Office[1]. It is a collection of annotated video clips dealing with "scenarios central to the Government's requirements," including: abandoned baggage events, movement in restricted areas, tracking via multiple cameras, and detection of parked vehicles. The Home Office runs yearly evaluations for companies and universities to submit their systems for evaluation on these scenarios. Here, we focus on the last scenario and use video clips from one of three datasets in the Parked Vehicle task, PV3, to detect anomalous events.

Note that the work described in this chapter is *not* attempted as an entry into the i-LIDS evaluation itself, but as an evaluation of the performance of a heterogeneously-

[1]Details at `http://www.ilids.co.uk`.

accelerated real-time system, as applied to a real-world vision task. Our definition of anomalous events in this case is simply *"observed behaviour which is absent or rarely present in the training data."* This can include pedestrians moving into unexpected areas of the scene or vehicles moving in an unexpected way, *e.g.* entering or exiting a rarely-used car park. In practice, however, it involves vehicles parking in forbidden areas. In the complete PV dataset, videos of three separate locations are provided. There are 24 hours of training and 24 hours of test footage, split across the three locations. Each individual clip within a scenario required registration points to be obtained manually. For this reason and to reduce evaluation time (3 repetitions of 8 hours each per run), PV3 is used as representative of the entire dataset.

Parked Vehicle Scene Three

A sample image of PV3 is shown in Figure 6.2b. Similarly to *BankSt* this involves colour, visible-light surveillance videos of urban road scenes taken from an elevated position, at 720×576 resolution and 25 FPS. This camera overlooks a two-way road with several turnings on the left and right. A roundabout off-screen below the image often causes traffic on the right of the picture to queue. Pedestrians are usually found on the pavements but can appear anywhere within the image. A white van is parked by the side of the road in Figure 6.2b; this is an example anomalous event as defined by the Home Office. Timestamps and descriptions of all such anomalous events are given. Ground-truthed locations of vehicle and pedestrian data are also provided for some sequences in the training dataset. Pedestrians can appear at small scales far away in the images. Working with this dataset presents some challenges in the different weather conditions and day and night conditions are provided. Strong shadows, camera shake, camera repositioning between sequences, camera noise, and tape artefacts interfere with reliable scene analysis. In this respect, PV3 is of considerably poorer quality than *BankSt*. The intention of the Home Office was presumably to expose researchers to the level of accuracy which an algorithm working on real-world surveillance videos would require. Nevertheless, the evaluation of performance on better-quality images was the main factor motivating our collection of data in *BankSt*.

6.3 A Problem Description and Related Work

We discussed the wider challenge of anomaly detection in Section 2.3 and listed previous progress in stopped vehicle detection. As we note in §2.3, this field lacks standardised measurements and datasets. Other researchers have looked at detecting events such as illegal U-turns in video [107], but we address the challenge of detecting parked vehicles. These events are anomalous in that they are not representative of normal traffic flow, and are not present in the training data. They thus mostly fall into Category A ("very different from the training set") of Loy's three anomaly categories [109].

We are working with i-LIDS as it is a dataset which "accurately represents real operating conditions and potential threats" [148]. This data has been used for parked vehicle detection by other researchers. Albiol *et al*. [117] identify parked vehicles in PV3 with precision and recall of 0.98 and 0.96 respectively. As discussed in more detail in Section 6.9, their approach is considerably different from ours, in that they:- (a) require all restricted-parking lanes in the image to be manually labelled first, (b) only evaluate whether an object in a lane is part of the background or not (and so detect non-vehicle objects left in no-parking areas), and (c) do not work in real-time or provide performance information. In addition, their P and R figures do not denote events, bur rather the total fraction of time during which the lane is obstructed. Theirs is the only work we are aware of which evaluates the entire i-LIDS dataset. Due mainly to limitations within our detectors, our accuracy results alone do not improve upon this state-of-the-art, but bearing in mind the points noted above, we are trying to approach a different problem (real-world anomaly detection under power and time constraints) than Albiol *et al*.

6.4 High-level Algorithm

The high-level frame processing algorithm is given in Algorithm 1 Part 1 and continued in Algorithm 1 Part 2. It works on offline videos and maintains real-time performance by continually calculating the number of frames to drop. We do not count time taken to decompress the video, or time taken to mark up and display the output image as part of processing time, as these take up a significant fraction

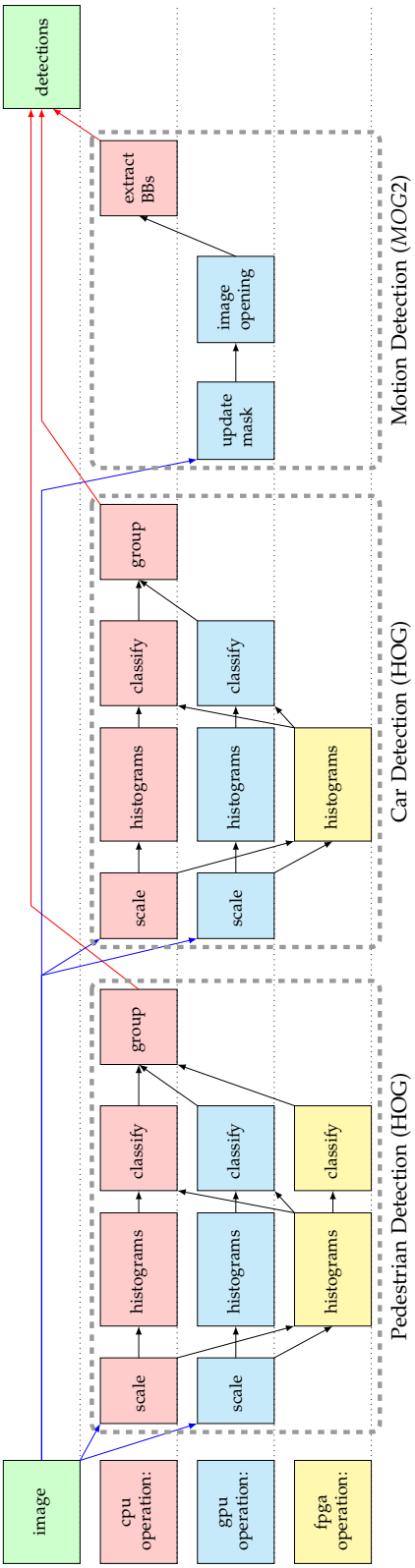


Figure 6.3: All possible mappings of image processing algorithms to hardware. Data transfer steps not shown.

Algorithm 1 Part 1. High-level frame processing algorithm for anomaly detection. Compute-intensive algorithm steps are marked with a black triangle ▶.

$I \leftarrow$ candidate implementations and performance data
$C \leftarrow$ learned trajectory clusters
load velocity and presence heatmaps H_v, H_p and ground-plane homography h_C
initialise list of Kalman-Filter based trackers T
for all frames f_i in video **do**
  if $f_i < f_{next}$ **then** ▷ must skip to maintain realtime processing rate
    $process\_this\_frame \leftarrow false$
  else
    $process\_this\_frame \leftarrow true$
  end if
  if *process_this_frame* **then**
    $A \leftarrow$ user-selected algorithms (motion and object detection by default).
    $P \leftarrow$ user/auto-selected priorities.
    $D \leftarrow 0, D_F \leftarrow 0, D_B \leftarrow 0$: initialise list of detections for this frame.
    run exhaustive search of all I matching A, using P as weightings for cost function.
    choose mapping M using lowest cost.
    for all implementations M_i in M **do** ▷ perform full-frame detections
      ▶▶ generate detections d_i from frame by running M_i
      append d_i to list of initial detections D
    end for
    for all detections d_j in D **do** ▷ improve detections
      remove duplicates and overlapping detections in D
      if d_j is from motion detector and still unclassified **then**
        ▶▶ run all object detectors in M on magnified region around d_j.
      end if
      remove duplicates and overlapping detections of d_j in D.
      append d_j to list of final detections D_F.
    end for
    predict all T.
    ground plane detections $D_B \leftarrow perspectiveWarp(D_F, h_C)$.
    match D_B to T (*i.e.* correct T using measurements D_B).
  else ▷ skip frame. Don't run expensive detectors.
    predict all T. ▷ only update the tracker positions
  end if
  continues . . .

Algorithm 1 Part 2. High-level frame processing algorithm for anomaly detection.

continues . . .
for all tracked objects t_j in T **do**
match t_j to a trajectory cluster C_k in C
update C_k ▷ update contextual knowledge and heatmaps
update H_v and H_p from t_j
cluster anomalousness $UC_j \leftarrow$ clusterAnomaly(C_k, C) ▷ calculate anomaly levels
► object anomalousness $UO_j \leftarrow$ contextAnomaly(t_j, H_v, H_p)
anomalousness $U_j \leftarrow weighted\ sum(UO_j, UC_j)$
if $U_j > U_{thresh}$ and $\tau > \tau_{thresh}$ **then**
snapshot and log detected anomaly at t_i
end if
end for
$U_{max} \leftarrow \max(U)$
if *automatically prioritise* **then**
update P based on U_{max}
else
update P from user callback
end if
if *process_this_frame* **then** ▷ allow realtime processing by skipping frames
$fskip \leftarrow \lceil t(current\ frame\ processing)/t(source\ frame\ duration) \rceil$
$f_{next} \leftarrow f_i + fskip$
display annotated frame
end if
end for ▷ end frame processing

of the 40*ms* available to process each frame at 25 FPS. Compute-intensive algorithm steps are marked with a black triangle ▶ in the listing. These were identified with a profiler during development.

These steps are also shown in Figure 6.1 and an expanded version of the 'detection algorithms' step showing all permutations is given in Figure 6.3. As all post-detection stages depended on trusting the detections generated by the individual detectors, any false positives or negatives at the detection stage could potentially cause anomalies to be missed, or flagged incorrectly. The main cause of our system not approaching the performance of Albiol *et al*. is the detectors we use. In the following section on implementation, we discuss false positive and negatives at the detector level.

6.5 Algorithm Implementations

This section gives details of each algorithm used within this system, summarised in Table 6.1. Here we describe the task performed by each algorithm, if not already discussed elsewhere, and its hardware implementations, if relevant. This Section begins with the detection algorithms run on the image, and concludes with generation of the image anomaly level. The following Section describes the methods used to select processing implementations to run on the next frame based on this measurement.

6.5.1 Pedestrian Detection with HOG

The pedestrian detector used is the accelerated version described in Chapter 5. The FPGA versions of HOG are scaled down to work faster on images 770 pixels wide, and to use fewer resources to allow inclusion of the car detector logic on the same physical FPGA.

As seen in Figure 6.4, the error rate of the HOG detectors led to false positives in certain circumstances. This affected the performance of the contextual motion detection further in the processing pipeline.

Table 6.1: Details of algorithms used in the system, sources and implementations.

Algorithm (source)	**Platform**	**Implemented by**
Pedestrian Detection (via HOG [78])	CPU	OpenCV
	GPU	OpenCV
	FPGA	this project: *gff* and *cff* versions
Car Detection (via HOG)	CPU	this project
	GPU	this project
	FPGA	intern: *gfg* and *cfc* versions
Background Subtraction (via Mixture of Gaussians [149])	GPU	OpenCV
Object Tracking (via Kalman Filter [150])	CPU	OpenCV
Anomaly Detection (via Trajectory Clustering [106])	CPU	this project
Anomaly Detection (via Bayesian motion context)	CPU	this project

6.5.2 Car Detection with HOG

We required an implementation of a vehicle detector which could be used in close to real-time, and ideally one which made use of an accelerated processor. The `LatentSVM` object detector as part of OpenCV has a detector trained for vehicles, but only offers a small improvement in detector accuracy over HOG. It is also not optimised and extremely slow (taking over one second per frame on CPU, even once the slowest parts are vectorised using SSE). In his thesis, Dalal gave implementation details for HOG for use with other object classes than pedestrians [78]; these are given in Table 6.2. See Section A.1 for details of normalisation methods used. Based on this, HOG was chosen once again due to the presence of existing accelerated versions and a framework to run them in. From this, the CPU implementation of HOG on OpenCV was modified to use the options given in Table 6.2. A GPU version was also implemented by modifying OpenCV. A version incorporating histogram generation on FPGA was implemented by a student intern by modifying the *cfc* and *gfg* implementations discussed in Chapter 5.

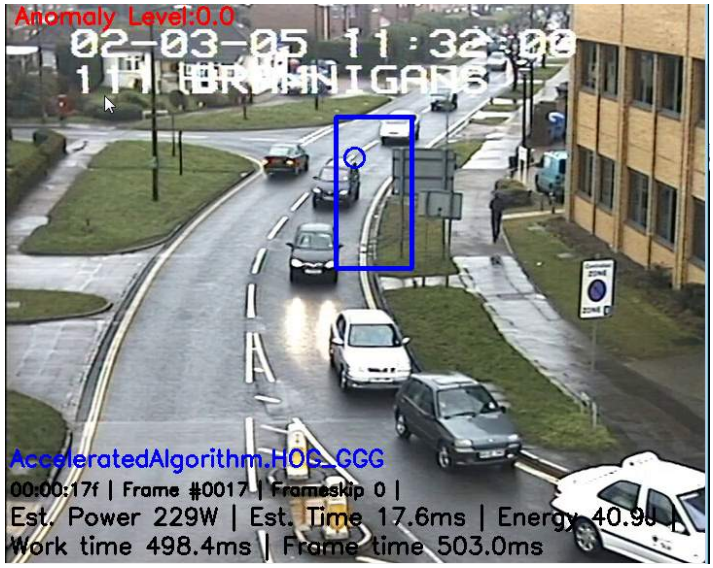


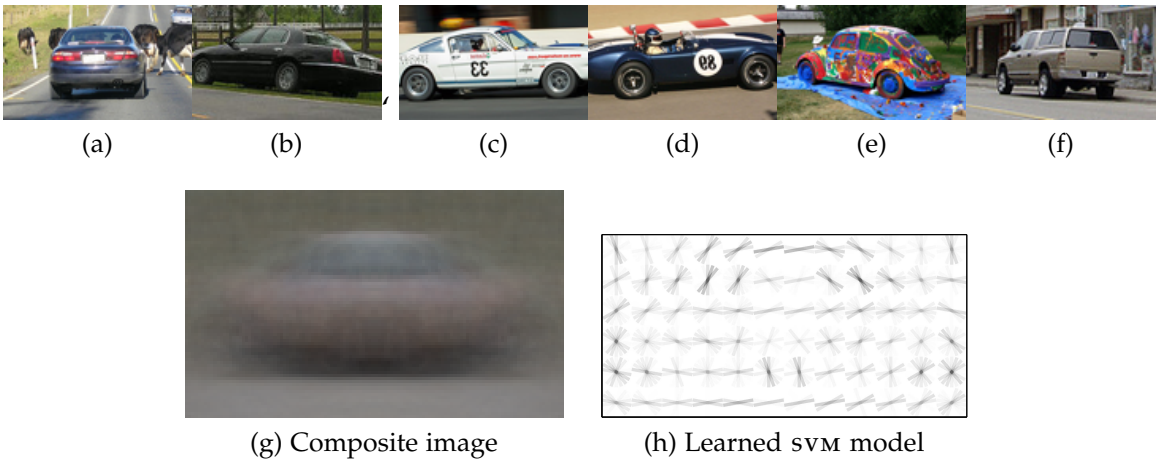
Figure 6.4: False positive (blue rectangle) from HOG pedestrian detector. Similar problems affected the car detector.

Table 6.2: Changes between HOG parameters and methods for pedestrian and car detectors, taken from [78].

Class	Window Size (pixels)	Average Height (pixels)	Angle Bins	Angle Range	Normalisation Method
Person	64×128	96	9	$(0 - 180°)$	L2-Hys
Car	104×56	48	18	$(0 - 360°)$	L1-Sqrt

These implementations were trained on data from the PASCAL Visual Object Classes Challenge [74], 2012 version[2]. Image patches annotated as containing cars were extracted from the training and testing sets and resized into a 104×56 window so that each car was around 48 pixels high. Cars were selected if they were not marked as "occluded", "truncated" or "difficult" in the ground truth, and were at least 40 pixels high in the original image. A positive training set was made up of 106 images of cars extracted from one half of the annotated set. These were flipped horizontally to double the available number of images, producing 212 positive training and 172 positive test image patches; examples are shown in Figure 6.5. A composite based on the average intensities of all training images is shown in Figure 6.5g. Negative examples (limited to 500 to reduce training time) from the dataset were also chosen from images which did not contain cars.

[2]Details at `http://pascallin.ecs.soton.ac.uk/challenges/VOC/`



(a) (b) (c) (d) (e) (f)

(g) Composite image (h) Learned SVM model

Figure 6.5: Example images for HOG car detector training. Composite of all training images shown in (g). Resulting detector model shown in (h).

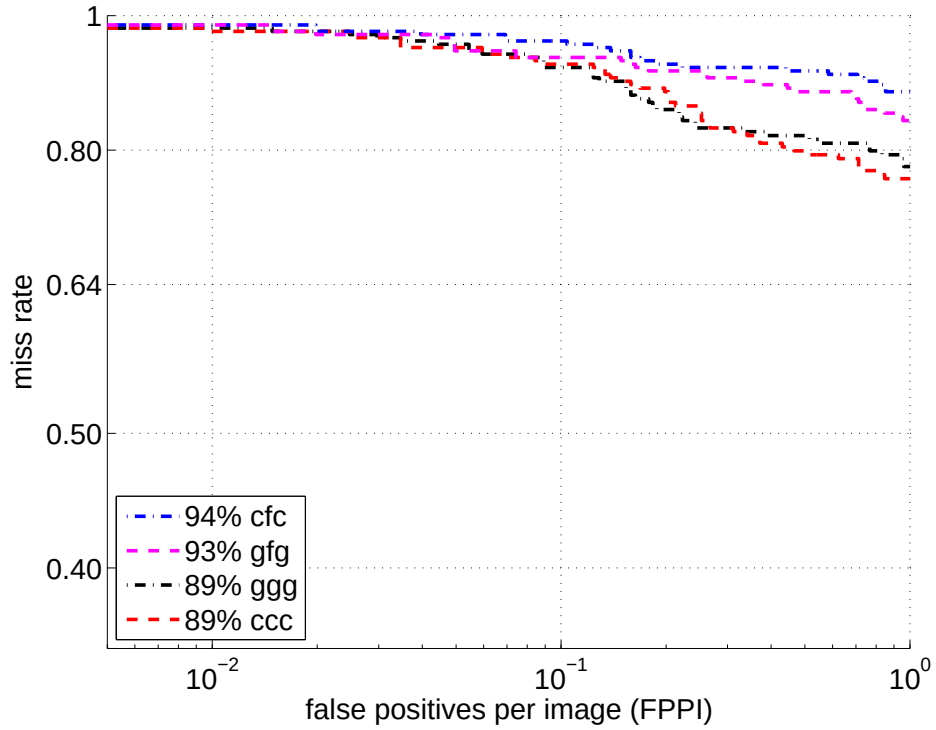


Figure 6.6: Detection Error Tradeoff curve for car detector implementations.

Table 6.3: Resource Utilisation for pedestrian (histogram and classification) and car (histogram only) HOG detectors with PCIe link logic on FPGA.

Resource	Percentage Used
Registers	27%
LUT	52%
Slice	75%
BlockRAMs	20%
Embedded Multipliers	14%

The SVM model was generated through training in a similar manner to that described in Chapter 5, (*i.e.* positive and negative training followed by retraining on all hard negative examples) and a visualisation is shown in Figure 6.5h. Following the discussion by Dollàr *et al.* [82], we show a FPPI curve in Figure 6.6, generated from all positive images in Pascal-VOC at scale factor 1.05. This detector is not as accurate as the original pedestrian version; this is probably due to the wider variation in training data and smaller dataset than INRIA. The latter has less variation in shape and size when seen from different angles; cars can be seen head-on, from the side and at various angles and elevations, and in some cases need to be wider than the window to maintain the 48-pixel height. The detector is only trained on cars, although during testing it was also possible to detect vans and trucks. The scaling factor was set to 1.05, giving a maximum of $n_{scales} = 31$. The grouping threshold was set to 1 and the hit threshold to 0.75 during use. Instead of using this hard threshold, methods for obtaining probabilistic output from a support vector classifier score (such as Platt's method involving sigmoids [93]) were investigated, but were not considered robust enough. For further justification see [94] or [95] §6.4.

All designs (pedestrian HOG with histogram and score outputs, and car HOG with histogram outputs) were implemented on the same FPGA, using Xilinx Planahead 13.4 as in Chapter 5, and with detectors running at $160MHz$. (While both car and pedestrian detectors were capable of running at $200MHz$, the larger feature vectors used in the car detector overflowed the output buffer at the higher clock rate.) Resource use for this joint design is given in Table 6.3. Performance characteristics are given — along with those for all other algorithm implementations — in Table 6.4.

(a) Motion detection by background subtraction in video.

(b) False regions of interest generated by background subtraction due to changes in illumination and camera shake.

Figure 6.7: Bounding box extraction from Mixture-of-Gaussians GPU implementation.

6.5.3 Background Subtraction

The Mixture of Gaussians algorithm was used to perform background subtraction and leave foreground objects. The OpenCV GPU version of this is based on Zivkovic's implementation [149] (which expands from [110]), and includes shadow detection, which was effective in the sequences with strong sunlight. The resulting binary image was morphologically opened and then transferred back to the host. Contour detection was then performed to generate bounding boxes around objects in the scene for further processing, as shown in Figure 6.7a. For each identified bounding box, the area was calculated and the box discarded if its area was below a threshold defined by its lower vertical bound y_0 within a frame:

$$T_{area} = C_a + \frac{C_b \times y_0}{h_{frame}},$$

where $C_a = 2000$ and $C_b = 400$. As the contents of every bounding box were then passed to one or more computationally intensive algorithms, early identification and removal of possible overlaps resulted in significant reductions in processing time. All bounding boxes were compared, and of those pairs with $\geq 90\%$ intersection with each other, the smaller one was discarded; *i.e.* we discard B_i if:

$$\frac{B_i \cap B_j}{area(B_j)} \geq 0.9 \;\&\; area(B_i) < area(B_j).$$

Occasionally heavy camera shake, fast adjustment of the camera gain, or fast changes in lighting conditions would cause large portions of the frame to be falsely indicated to contain motion, as shown in Figure 6.7b. When this occurred, all bounding boxes for that frame were dropped, and we waited until the next frame to reacquire bounding boxes.

6.5.4 Detection Combination

Detections of objects could be generated from two sources: a direct pass over the entire frame by the pedestrian or car detector, or by detections on a magnified region of interest triggered by areas of motion identified by the background subtractor. Regions with motion were extracted and magnified by 1.5× then passed to detectors on CPU or GPU. This allowed detection of pedestrians less than 96 pixels high, which would otherwise have been missed. The alternative to this step was to magnify the entire image by 2× then run both HOG detectors on the result. However, this would be prohibitively slow if done every frame. For both humans and cars, a minimum detection grouping threshold of 3 in the image patches was effective. Candidate detections from global and motion-cued sources were filtered using the 'overlap' criterion from the PASCAL VOC challenge [74]:

$$a_0 = \frac{area(B_i \cap B_j)}{area(B_i \cup B_j)},$$

and duplicates removed if $a_0(B_i, B_j) > 0.45$, and where the two object classes were compatible. Regions with unclassified motion were still passed to the tracker matcher to allow previously-classified detections to be updated or identification of new tracks.

6.5.5 Detection Matching and Object Tracking via Kalman Filtering

A constant-velocity Kalman filter [150, 102] was used to smooth detections from all sources before further processing. The filter equations are given in Appendix A. The detectors run every frame and do not take temporal information into account. This is done at a higher level by the tracker, which matches new detections to existing tracks. This approach is particularly useful with imperfect detectors which do not identify an object every frame, or only identify regions within an object. Unmatched predictions are carried over until the object is detected again, or deleted if an object is not seen for a long time. Due to the wide variation in depth of objects

as seen from each camera, all object detections were projected onto a ground plane before smoothing. This normalises inter-object distance and minimises perspective distortion — important for the clustering stage [101]. The equations for this are given in Section A.3.

A new ground plane detection D with bounding box centre (x_d, y_d) is matched to an existing track T_i if the Euclidean distance between T_i and D is less than a fixed threshold, and if T_i is the tracker with the smallest distance r. r is an elliptical distance measure between an ellipse centred at the origin, and a point (x, y):

$$r = \sqrt{x^2 + \frac{y^2}{1 - e^2}}, \tag{6.1}$$

where the ellipse has eccentricity $e = \sqrt{1 - b^2/a^2}$, defined using the radii a and b along the long and short axes respectively. The long axis of the ellipse in Equation 6.1 points along the x-axis and denotes the direction of motion of the tracker. (x_d, y_d) denotes the position of the detection as seen from the tracker, which is considered to be at the origin. $2r$ thus denotes the length of the long axis of an ellipse passing through (x_d, y_d) and centred with orientation θ_{T_i} at (x_{T_i}, y_{T_i}). If the tracker is stationary, r becomes the Euclidean distance measure. Once matched, the tracker is updated using the detection as the measurements in Equation A.7.

This approach takes into account deficiencies in both the object detection and background subtraction stages, in that detection bounding boxes may be located anywhere within an object, and may be significantly smaller than the object itself. At the same time, we especially wish to avoid matching a tracker to *e.g.* a nearby car travelling in the opposite direction, as this would generate a false positive for anomalous activity. Detections not matched to an existing tracker are assigned a new one, and trajectories not matched to a detection for a long time are deleted. Trajectories are merged if they are very close together, travelling at the same speed and in the same direction. At the end of this algorithm stage, a list of points denoting the centre of objects are passed to the trajectory clustering algorithm. Tracked points in the base plane are shown in Figure 6.8. As trajectory smoothing and detection matching operate on the level of abstraction of objects rather than pixels or features, the number of elements to process is low enough, and the computations for each one are simple enough that this step is not considered as a candidate for acceleration via parallelization.

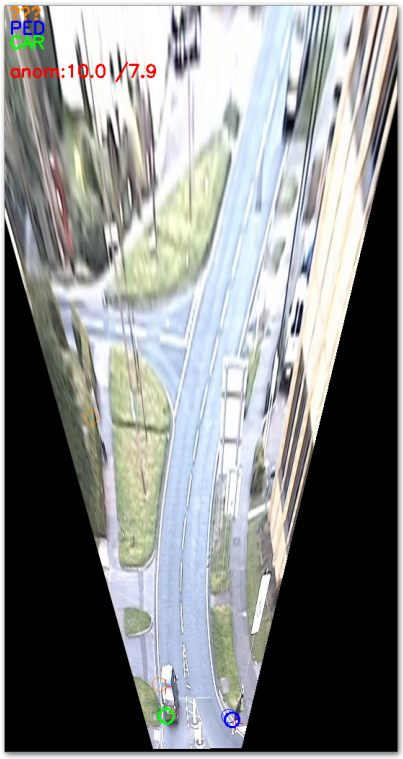


Figure 6.8: Object tracking on an image projected onto the ground plane.

6.5.6 Trajectory Clustering

The trajectory clustering algorithm used is based on a reimplementation of that described by Piciarelli and Foresti [106], used for detection of anomalies in traffic flow. Piciarelli *et al*. apply the algorithm to fast-moving traffic on a motorway, whereas the i-LIDS scenes have more discrete types of object (pedestrians and vehicles), numerous entrances and exits in the scene, greater opportunities for occlusion to occur, and long sequences where objects stop moving entirely and may start to be considered as part of the background. However, it allows online learning of cluster positions and analysis of tracks still in progress, so we describe the algorithm as we have used it in this application.

Starting with a short trajectory $T_i = (t_0, t_1, \ldots, t_4)$ consisting of several smoothed detections seen over several frames (five, in this case), the algorithm matches these to and subsequently updates a set of clusters C. Each cluster C_i contains a vector of elements $c_j = (x_j, y_j, \sigma_j)$, with a location x_j, y_j and a variance σ_j. Clusters are arranged in a tree structure, with each having zero or more children. One tree

(starting with a *root cluster*) thus describes a single point of entry to the scene and all observed paths which are taken through the scene from that point.

For a new or unmatched trajectory T_u, all root clusters and their children are searched to a given depth and T_u is assigned to the closest C if distance d is below a threshold. The calculation made for d is:

$$d(T_u, C_i) = min\left(\frac{dist(t_i, c_j)}{\sqrt{\sigma_j^2}}\right), j \in \{\lfloor(1-\delta)i\rfloor \ldots \lceil(1+\delta)i\rceil\}, \tag{6.2}$$

where *dist* is the Euclidean distance between point t_i of T_u and point c_j of C_i, and the lower and upper bounds of the elements within C_i to search over are controlled by $\delta = 0.4$. For new points in trajectories previously matched to a cluster, this δ factor allows points matched to longer clusters more possible matches, to take account of subsequent objects within one cluster not moving in exactly the same manner. If a point is matched to a cluster, the closest element match c_j is updated with t_i:

$$\begin{aligned} x_j &= (1-\alpha)x_j + \alpha x_i \\ y_j &= (1-\alpha)y_j + \alpha y_i \\ \sigma_j^2 &= (1-\alpha)\sigma_j^2 + \alpha[dist(T_i, c_j)]^2, \end{aligned}$$

where the learning rate $\alpha = 0.05$. If T_u is not a match to any point within the root search tree, a new root cluster is created and added to C.

Three actions are possible for new points appearing as the most recent point in a trajectory which has previously been matched:

- If a point is matched to the end of a cluster, a child of that cluster is spawned with a length of 1, matched to that point, and subsequently concatenated into the parent cluster. This is the most common behaviour for clusters generated to match specific trajectories and which keep pace with them.

- If a point is matched to within a cluster, that cluster point is updated as described above.

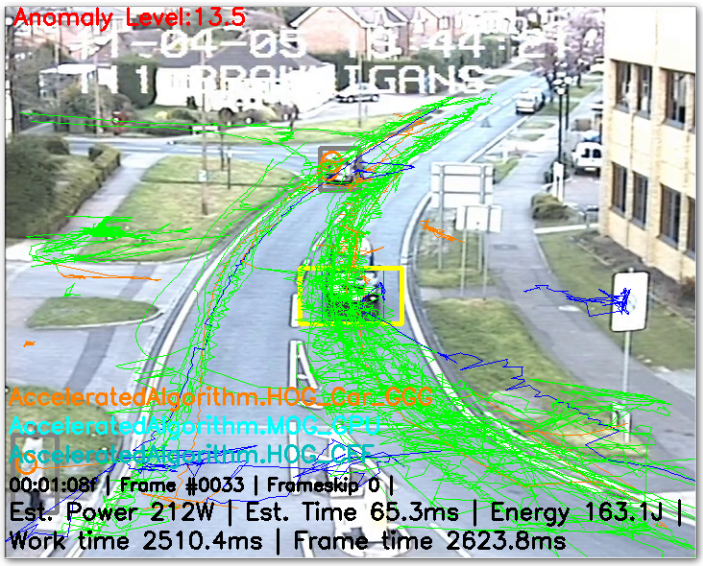


Figure 6.9: Learned object clusters, projected onto camera plane. Green, blue and orange tracks represent cars, pedestrians and undetermined (motion-only) objects respectively.

- If a point is matched to within a cluster and is diverging from that cluster, a child cluster is created and split from its parent. The parent cluster is truncated at the split point and subsequent points are stored in another child cluster.

Cluster maintenance (involving pruning clusters not seen in a long time, and concatenation of single-child clusters into their parents) is performed periodically. As with detection matching and tracking, clustering operates on a relatively small number of objects and is not computationally expensive, so was not considered as a candidate for acceleration. Learned class-specific object clusters are shown in Figure 6.9, projected back onto the camera plane.

6.5.7 Contextual Knowledge

Contextual knowledge in this case relies on known information about the normal or most common actions within the scene. Position and motion information can capture various examples of anomalous behaviour: for example stationary objects in an unusual location, or vehicles moving the wrong way down a street. Ground truth information giving object location, type and unique ID is provided for certain sequences within i-LIDS; however, due to discrepancies between the ground truth and detections generated by the object classifiers (caused by camera repositioning and differences in sampling frequency), unsupervised learning based on the output

of the object classifiers was used instead. This has an advantage over hidden Markov Model-based systems such as [105] in that we can still make use of the velocity data from (frequent) broken and reformed tracks away from the edges of the frame.

Type-specific information about object presence at different locations in the base plane was captured by recording the per-pixel location of each base-transformed bounding box, then downscaling this by two to decrease evaluation time. Figure 6.10 shows presence heatmaps for car and pedestrian classes for PV3, *i.e.* where these objects are expected to be found in the scene. Average per-pixel velocity $\bar{v}$ in $x-$ and $y-$directions was also recorded by sampling this data from the Kalman Filter, and downscaling in the same way. Thus for an object existing at base plane co-ordinates $(x \ldots x',\ y \ldots y')$ with $x-$velocity v_x and update rate $\alpha = 0.0002$:

$$\bar{v}_{(x \ldots x',\ y \ldots y')} = (1 - \alpha)\bar{v}_{(x \ldots x',\ y \ldots y')} + \alpha v\,.$$

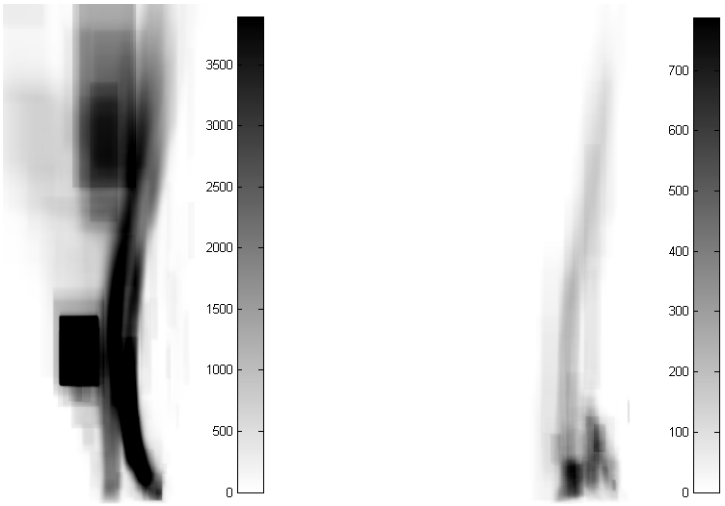
This is shown in Figure 6.11. For most conceivable traffic actions, presence and motion information is appropriate; however, this may fail to capture more complex interactions between people. In addition, errors from the object and motion detectors now affected the performance of the Bayesian context algorithm; Figure 6.11 also contains false-positive and object misclassifications. In certain areas these persisted over long periods of time, in effect removing the ability to identify stationary objects in the affected regions as anomalous.

6.5.8 Anomaly Detection

Based on the methods given in the two preceding sections, we can define an anomalous object as:

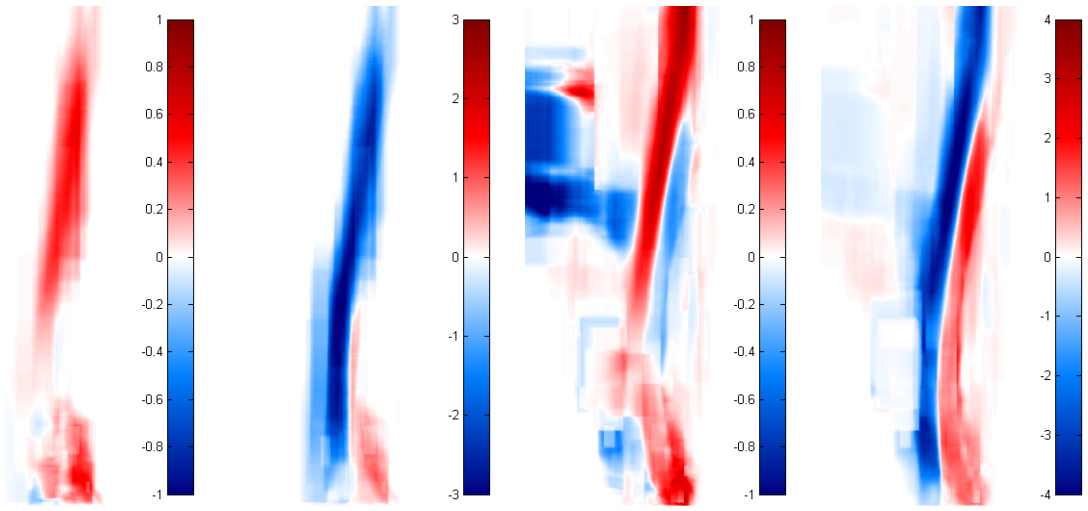
- one which is present in an unexpected area, or:
- one which is present in an expected area but which moves in an unexpected direction or at an unexpected speed.

A Bayesian method is used to determine if an object's velocity in the x and y directions should be considered anomalous, based on the difference between it and the known average velocity $\bar{v}$ in that region. Bayes' theorem [151] defines the probability of an anomaly at a given pixel $p(A|D)$, given detection of an object at



(a) Car presence (b) Pedestrian presence

Figure 6.10: Presence intensity maps for different object classes in PV3. The colour bar denotes objects passing through that pixel, and the x and y axes correspond to the ground plane image of the scene (shown in Figure 6.8).



(a) Person x-motion (b) Person y-motion (c) Car x-motion (d) Car y-motion

Figure 6.11: Ground-plane, per-pixel motion intensity maps of ground-truthed data for different object classes in PV3. The colour bar denotes average velocity at that pixel, and the x and y axes correspond to the ground plane image of the scene (shown in Figure 6.8).

that pixel:

$$p(A|D) = \frac{p(D|A)p(A)}{p(D|A)p(A) + p(D|\bar{A})p(\bar{A})}\,, \tag{6.3}$$

where the prior probability of an anomaly anywhere in the image, $p(A)$, is set to a constant value. $p(D|A)$, the likelihood of detecting an event at any pixel in the presence of an anomaly, is constant (*i.e.* we assume that an anomaly can occur with equal probability anywhere within the image, as we do not have any information about it), $p(\bar{A}) = 1 - p(A)$ and $p(D|\bar{A})$ is a measure based on the learned values for $\bar{v}_x$ or $\bar{v}_y$.

$p(D|\bar{A})$ returns values between $(0.01, 0.99)$ based on the distance between v and $\bar{v}$, assuming a linear relationship between distance to the mean and $p(\bar{A})$:

$$d_v = \begin{cases} sign(\bar{v}) \times \max(C|\bar{v}|, |\bar{v}| + C), & \text{if } sign(\bar{v}) = sign(v) \;\&\; |v| > |\bar{v}| \\ sign(\bar{v}) \times \min(-0.5|\bar{v}|, |\bar{v}| - c), & \text{otherwise.} \end{cases} \tag{6.4}$$

d_v is then used to obtain a linear equation for v, with a gradient of:

$$a = \frac{0.01 - 0.99}{d_v - \bar{v}}\,.$$

Finally, b is obtained in a similar manner, and v is projected onto this line to obtain a per-pixel likelihood that v is *not* anomalous,which is then clamped:

$$\begin{aligned} y &= av + b\,, \\ p(D|\bar{A}) &= \max(0.01, \min(0.99, y))\,. \end{aligned} \tag{6.5}$$

Profiling during development showed that evaluation of this step was expensive, so this function was vectorised using SSE on the CPU.

$p(D|\bar{A})$ is substituted into Equation 6.3 to obtain an object anomaly measure UO_x. This is repeated for $y-$velocity data to obtain UO_y.

This measure is combined with information about the abnormality of the current cluster associated with the object. When a trajectory moves from one cluster to one of its children, leaves the field of view, or is lost, the number of transits through that cluster is incremented. In addition, for any trajectory T matched to a cluster

C_p with children C_{c1} and C_{c2}, the number of trajectory transits between C_p and all C_c is logged, thus building up a trajectory frequency distribution between C_{c1} and C_{c2}. These two metrics (cluster transits and frequency distributions of parent–child trajectory transitions) allow anomalous trajectories to be identified:

$$UC(C_i) = \begin{cases} \dfrac{1}{1 + transits(C_i)}, & \text{if } C_i \text{ is a root node,} \\ 1 - \dfrac{transitions(C_p \rightarrow C_i)}{\Sigma(transitions(C_p \rightarrow \text{all children of } C_p))}, & \text{if } C_i \text{ is a child node of } C_p\,. \end{cases} \tag{6.6}$$

An overall anomaly measure U_i for an object i is then obtained, and U_{max} is updated:

$$U_i = w_o \frac{\Sigma_1^{\tau_i} UO_x}{\tau_i} + w_o \frac{\Sigma_1^{\tau_i} UO_y}{\tau_i} + w_c UC(C_i)\,. \tag{6.7}$$

$$U_{max} = \max(U_i)\,. \tag{6.8}$$

The $UO_{x,y}$ measures are running totals which are normalised using the object age τ.

After computing Equation 6.8, we have now moved from an input to the system consisting of a frame of pixels, to a single scalar representing the level of anomaly present in the scene for that frame. Two detectors are required to flag an object as behaving unusually before the system treats it as an anomaly. w_0 and w_c are thus set to 10, with the overall anomaly detection threshold set at 15.

Once an anomaly is detected for the minimum threshold time t_A, its location and object class is logged and a snapshot of the annotated video frame is saved. An example snapshot is in Figure 6.12. In the next Section we discuss how the U_{max} measure is used to choose which platforms to map the algorithms onto.

6.6 Dynamic Mapping

The mapping function selects algorithm implementations used to process the next frame. This code is re-run every time a frame is processed, allowing for selection

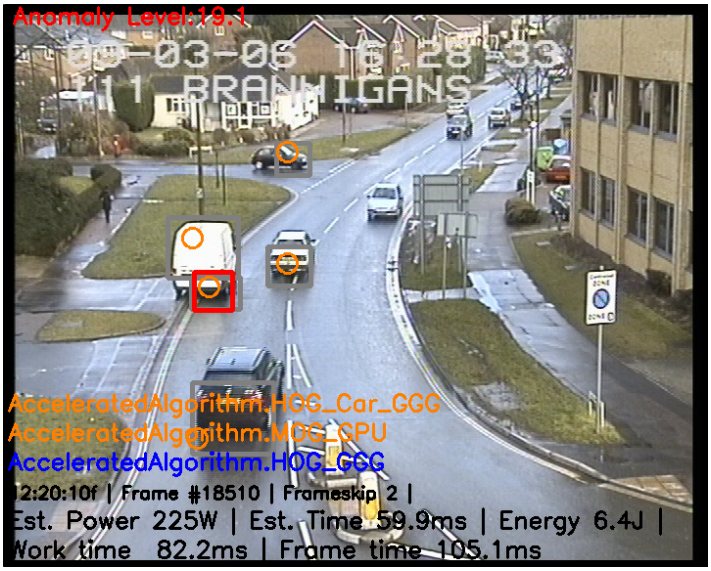


Figure 6.12: An anomaly detection is logged. The red box indicates a van parked in a forbidden location.

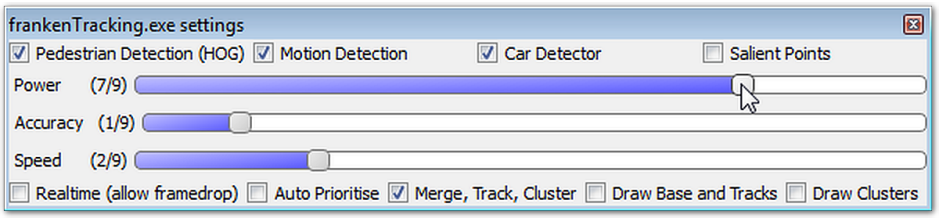


Figure 6.13: Dashboard for user-driven priority selection. Moving a slider to the right represents increased weight given to that performance characteristic. Other sliders are automatically moved to the left to compensate.

of new implementations in response to changing scene characteristics. After every processed frame, if $process\ time > frame\ time$, then $\lceil process\ time / frame\ time \rceil$ frames are skipped to regain a realtime rate. The algorithms used for all tests were fixed as HOG-PED, HOG-CAR and MOG.

6.6.1 Priority Recalculation

A mechanism for displaying and updating the current system priorities P is shown in Figure 6.13. This allocated ten credits between the three priority sliders or characteristics in P: power consumption w_p, processing time w_t and detection accuracy w_ϵ. Moving a slider — either programmatically or by user intervention — caused the other sliders to move in the other direction accordingly. Here we work on

Table 6.4: Performance characteristics for various algorithm implementations on 770×578 video. Where the algorithm required scaling, n_{scales} was tuned to the dataset used, and was 8 for pedestrians and 11 for cars in this case.

Algorithm	**Platform**	**Processing Time (ms)**	**Power Consumption (W)**	**Accuracy (log-average miss rate (%))**
HOG-PED	*ggg*	17.6	229	52
	cff	23.0	190	62
	gff	27.5	186	61
	gfg	39.0	200	59
	cfc	117.3	187	59
	ccc	282.0	191	53
HOG-CAR	*ggg*	34.3	229	89
	cfc	175.6	189	94
	gfg	60.0	200	92
	ccc	318.0	194	89
MOG	GPU	8.1	202	N/A

the assumption that a higher level of anomaly in the scene should be responded to with increased processing resources in order to obtain more information about the scene in a timely fashion, with fewer dropped frames, and at the expense of lower power consumption. Conversely, frames with low or zero anomaly levels caused power consumption to be scaled back, at the expense of accuracy and processing time. Realtime processing is maintained by dropping a greater number of frames. When automatic prioritisation was used, (*i.e.* the system was allowed to respond to scene events by changing its mapping) the *speed* priority was increased to maximum when $U_{max} \geq 15$. A level of hysteresis was built in by maximising the *power* priority when $U_{max} < 12$.

6.6.2 Implementation Mapping

Once a list of candidate algorithms and a set of performance priorities is generated, implementation mapping is run. This uses performance data for each algorithm implementation, shown individually for all algorithms in Table 6.4.

This data is summarised for all possible solutions in Figure 6.14, where dots closer to bright green, red or blue signify the majority of algorithm stages being mapped to FPGA, GPU and CPU respectively. A more detailed version with individually

annotated solutions is given in Figure 6.15. A Pareto curve can be seen in this Figure, stretching from $\{ped - ggg, car - ggg, mog - gpu\}$ on the top left to $\{ped - cfc, car - cfc, mog - gpu\}$ in the lower centre, with all other points shadowed.

An exhaustive search function is used to generate and evaluate all possible mappings, using a cost or fitness function:

$$\mathcal{C}_i = w_p P_i + w_t t_i + w_\epsilon \epsilon_i \,, \tag{6.9}$$

where all w are controlled by P. Estimated power, runtime and accuracy is also generated at this point. As established in §5.6.3, there is no cost associated with changing the mapping between frames, so this is not considered as part of $\mathcal{C}$. The mapped algorithms are then run sequentially to process the next frame and generate detections.

Having now described all the steps in Algorithm 1, in the next Section we evaluate its performance.

6.7 Evaluation Methodology

The clusters and heatmaps were first trained by initialising on training video sequences containing varying levels of traffic. Each of the test sequences was then processed in approximately real time, using these learned clusters and heatmaps. The same set of learned clusters and heatmaps were used for each cluster, and were not updated (*i.e.* changes were not carried over) between test videos. To account for camera repositioning between sequences (some of which were filmed months apart), the homography matrices required to register each video clip onto the base plane were obtained manually. Tests were run three times and referred to as follows, using the prioritisation settings described in §6.6.1:

speed speed prioritised and anomaly-controlled automatic prioritisation off;

power power prioritised and anomaly-controlled automatic prioritisation off;

auto anomaly-controlled automatic prioritisation on.

For each configuration, all anomalous events (defined as objects having $U > 15$ for more than time limit of either $t_A = 10$ or 15 seconds) were logged and compared with ground-truth data. Total processing time and ratio of dropped to processed frames were logged. As the power measurement device did not have a method for recording average power over a period of time, power consumption for one frame was estimated by averaging the energy used to process each implementation over runtime for that frame. Average power consumption over a sequence of frames was obtained in the same manner.

6.8 Results

We present performance results on two datasets; an initial evaluation on *BankSt* and a complete evaluation on i-LIDS.

6.8.1 Detection Performance on BankSt videos

An initial evaluation was done on videos from *BankSt*. As noted in Section 6.2, this was done to evaluate system and algorithm performance on higher-quality videos than i-LIDS. A set of clusters and heatmaps was trained on a 20-minute clip from the *BankSt* dataset. As before, registration was done manually. Evaluation was done using a three-minute test clip, involving normal traffic patterns and including a blue car driving round the corner then parking illegally on the yellow lines, as shown in Figure 6.16. This one event was detected (see Figure 6.16b) along with one false positive event caused by slow-moving pedestrians. Both these events were detected using all three prioritisations.

As Figure 6.17 shows, video quality also affected detection performance; in Figure 6.17a magnifying allows additional classification of objects in the middle distance — shown via blue (pedestrians) and green (car) circles, while in i-LIDS data, magnifying and reclassifying the area around a moving object often failed to improve on existing detections.

6.8.2 Detection Performance on i-LIDS videos

For the i-LIDS data, clusters and heatmaps were first trained by initialising on two twenty-minute daylight sequences containing varying levels of traffic from the i-LIDS training set.

A modified version of the i-LIDS criteria [152] was used to evaluate detection performance. Events are counted as true positives if they are registered within 10 seconds of the start of an event. For parked vehicles, i-LIDS considers the start of the event to be one full minute *after* the vehicle is parked, whereas we consider the start of the event to be when the vehicle parks, thus we may flag events *before* the timestamp given in the ground truth data. Such events are counted as true positives. The time window for matching detections is thus either 70 or 75 seconds long. We also do not ignore events in the first five minutes of a video; they are present in the ground truth and we take much less than 5 minutes to initialise the system. The i-LIDS criteria only require a binary alarm signal in the presence of an anomaly; however, we require anomalous tracks to be localised to the object causing the anomaly.

Precision p and recall r are calculated in the usual manner from the number of true positives, false positives and false negatives (TP, FP and FN respectively):

$$p = \frac{TP}{TP + FP}, \tag{6.10}$$

$$r = \frac{TP}{TP + FN}. \tag{6.11}$$

i-LIDS also uses a F_1 score, allowing detection performance to be expressed as a single number. It is obtained from Van Rijsbergen [153] and defined as:

$$F_1 = \frac{(\alpha + 1)rp}{r + \alpha p}, \tag{6.12}$$

where α is a "recall bias" measure, set at 0.55 for real-time operational awareness (to reduce false-alarm rate), or 60 for event logging (to allow logging of almost everything with low precision). α is obtained from $\alpha = 1/(\beta^2 + 1)$, and "measures the effectiveness of retrieval with respect to a user who attaches β times as much importance to recall as precision" [153].

Table 6.5: Detection performance for parked vehicle events on all prioritisation modes on i-LIDS sequence PV3.

Prioritisation	**True positives**	**False positives**	**False negatives**	p(%)	r(%)
for $t_A = 10$ *seconds*					
power	4	44	26	8.3	13.3
speed	6	51	25	10.5	19.4
auto	6	53	25	10.2	19.4
for $t_A = 15$ *seconds*					
power	2	15	29	11.8	6.5
speed	8	13	23	38.1	25.8
auto	4	14	26	22.2	13.3

Table 6.6: Detection performance for parked vehicle events on all prioritisation modes on *daylight sequences only* in i-LIDS sequence PV3.

Prioritisation	**True positives**	**False positives**	**False negatives**	p(%)	r(%)
for $t_A = 10$ *seconds*					
power	4	29	23	12.1	14.8
speed	6	40	22	13.0	21.4
auto	6	42	22	12.5	21.4
for $t_A = 15$ *seconds*					
power	2	10	29	16.7	6.5
speed	8	8	23	50.0	25.8
auto	4	10	26	28.6	13.3

Table 6.5 shows precision and recall for all "parked vehicle" events in the i-LIDS PV3 sequence. As the night-time sequences have no events but still generate a large proportion of false positives, we also show results for daylight-only ("day" and "dusk" clips) in Table 6.6.

Table 6.7 shows F_1 scores for all videos and daylight-only scenes, for $t_A = 10$ and 15 seconds.

Table 6.8 shows performance details for all three priority modes on i-LIDS PV3. The runtime column shows overall program execution time relative to the total length of all source videos. This shows that, once overheads from decoding input frames

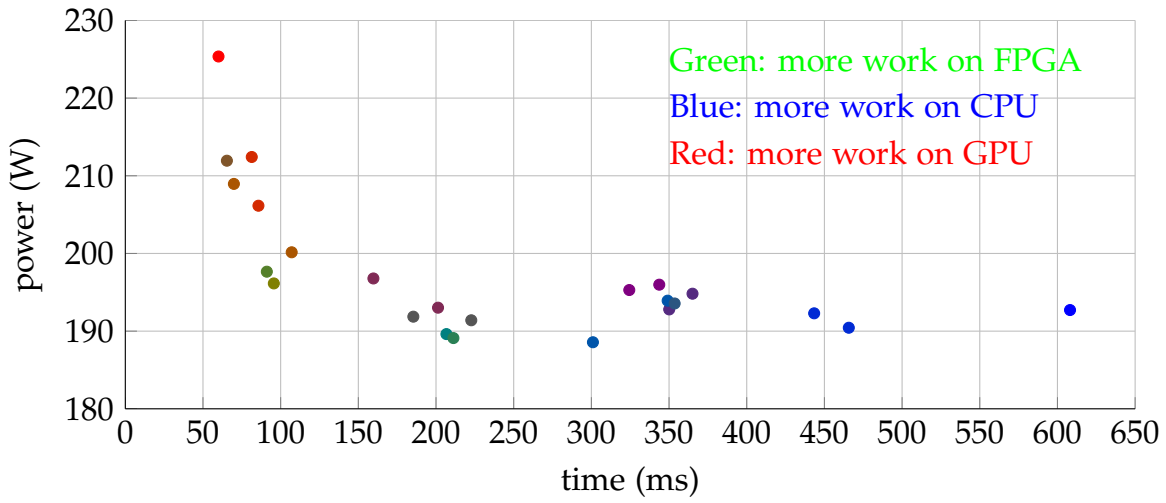


Figure 6.14: Power and time plots of all possible solutions, where each solution consists of one car HOG, one pedestrian HOG and the MOG detector, as described in Table 6.4. A greener dot represents a solution with most operations mapped to FPGA, while bluer and redder dots represent those with most operations mapped to CPU or GPU respectively.

Table 6.7: F_1-scores for all prioritisation modes on i-LIDS sequence PV3.

Prioritisation	**PV3**		**PV3 daylight only**	
	$F_1(\alpha = 0.55)$	$F_1(\alpha = 60)$	$F_1(\alpha = 0.55)$	$F_1(\alpha = 60)$
for $t_A = 10$ *seconds*				
power	0.0957	0.1317	0.1294	0.1475
speed	0.1254	0.1913	0.1510	0.2118
auto	0.1226	0.1912	0.1466	0.2115
for $t_A = 15$ *seconds*				
power	0.0910	0.0652	0.1067	0.0652
speed	0.3259	0.2594	0.3752	0.2601
auto	0.1797	0.1342	0.2030	0.1345

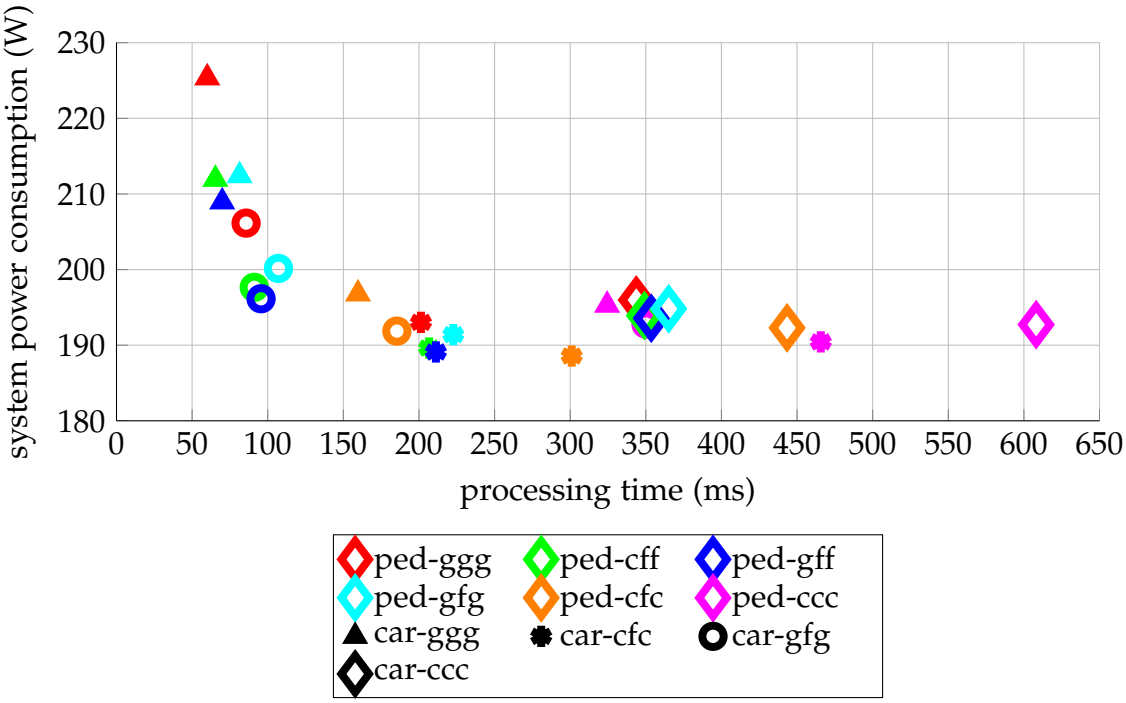
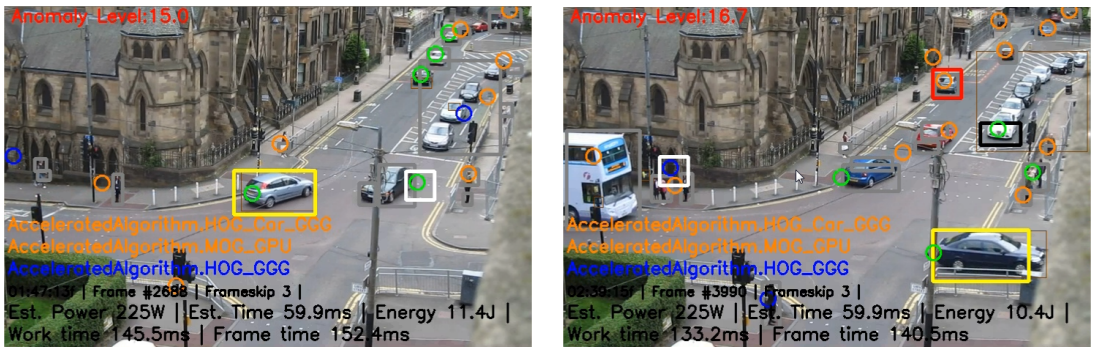


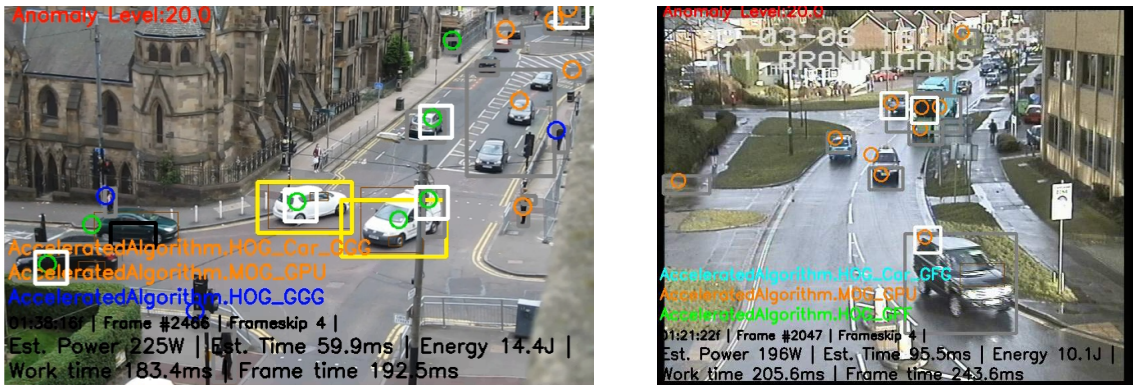
Figure 6.15: Power and time plots of all possible solutions, where each solution consists of one implementation of car HOG, one for pedestrian HOG and the MOG detector, as described in Table 6.4. Pedestrian implementations are labelled by colour, and car implementations by shape; thus the dark blue hollow circle at $t = 95ms$ means a solution with pedestrian HOG mapped to *gff*, car HOG mapped to *gfg* and MOG, as always, mapped to GPU.



(a) Blue car (centre)

(b) Stopped vehicle

Figure 6.16: Detection on the BankSt dataset. In (a), the blue car (centre) is behaving normally, while in (b) it has stopped by the roadside and is eventually flagged as anomalous.



(a) *BankSt* video (b) i-LIDS video

Figure 6.17: Dataset quality impacts quality of detections: in a *BankSt* video (a), video quality allows classification of most objects as either pedestrian (blue circle) or car (green circle). In i-LIDS, (b), detections often remain as uncategorised motion (orange circle).

Table 6.8: Processing performance for all prioritisation modes on i-LIDS sequence PV3. Idle/baseline power is 147*W*.

Prioritisation	**Frames skipped (%)**	**Runtime as % of source time**	**Processing time as % of source time**	**Mean estimated power above baseline while processing (W)**
power	75.8	125.4	87.9	49.1
speed	59.5	124.3	81.7	72.8
auto	66.5	127.8	83.4	61.9

from a video file on disk, marking up output images for display, and displaying the resulting images are included, total runtime is slightly slower than real (or source video) time. The processing time column does not include overheads. It assumes that a raw video frame is already present in memory, and frame-by-frame marked-up video output is not required (*i.e.* only events are logged). Under these circumstances, the system runs faster than realtime, with the percentage of skipped frames shown in the second column. This is as expected; the slower power-optimised priority setting causes more frames to be skipped. Table 6.9 repeats this for daylight-only sequences, with similar results.

Figure 6.18 shows examples of true positive, false negative and false positive detections logged at t_A seconds. While some true positives are detected (up to 50% in the best case), many false negatives and false positives are present, and have various causes. False positives are often caused by the background subtractor

(a) TP (b) TP (c) TP

(d) TP (e) FP (f) FP

(g) FP (h) FN (i) FN, FP

(j) FN, FP

Figure 6.18: True detections and failure modes of anomaly detector on i-LIDS PV3. Anomalies are marked by red squares. (a)–(d): true positives of different objects in varying locations and with varying numbers of other vehicles. (e)–(g): false positives, caused by slow-moving traffic, an object moving offscreen and incorrect background subtraction respectively. (h): false negative caused by occlusion. (A car is hidden behind the road sign on the right-hand side). (i) is treated as a false negative and a false positive: the detector identifies the car on the left instead of the van parked beside it. (j) is an anomaly identified outside the allowed time window, so counts as a false negative and false positive.

Table 6.9: Processing performance for all prioritisation modes on *daylight sequences only* in i-LIDS sequence PV3. Idle/baseline power is 147*W*.

Prioritisation	Frames skipped (%)	Runtime as % of source time	Processing time as % of source time	Mean estimated power above baseline while processing (W)
power	75.5	125.4	87.9	49.1
speed	60.1	124.2	82.0	78.4
auto	66.0	122.0	83.4	61.8

erroneously identifying patches of road as foreground, caused by the need to acquire slow-moving or waiting traffic in the same region. The anomaly detectors then flag an object on the road as either stationary for long periods or moving in the wrong direction, and generate a false alarm.

False negatives are, in general, caused by poor performance of the object classifiers or background subtractor. Directly failing to detect partially occluded objects (as in Figure 6.18h), or stationary, repeated false detections in regions overlapping the roadside which are generated during training can both cause anomalies to be missed. To reduce processing of a large number of regions which are erroneously designated as foreground by the background subtractor during a large, fast change in camera gain, all regions are discarded immediately after such a change and the background is re-learnt once the image stabilises. In at least two cases, camera gain is changed immediately after a vehicle parks at the roadside. The parked vehicle is then treated as part of the new background and not detected.

The false positive shown in Figure 6.18i is a result of the criteria for i-LIDS detection; the white van is stopped and should be registered as an anomaly, but the car on the left is flagged instead. This situation counts as one false-negative and one false-positive event.

6.9 Analysis

In this Section we draw attention to significant results from this experiment and place it in the context of other work. We then comment on possible improvements to

this system in ascending order of abstraction, from improvements to the underlying architecture to algorithms to the task itself.

Unlike in Chapter 5 when we considered the tradeoffs between time, power and accuracy, here the system must run in real-time, so we drop frames to ensure a close-to-realtime rate. The main tradeoffs which we can make here are therefore accuracy and power; optimising for time allows more frames to be processed, which in combination with the natural increased detection accuracy of the *ggg* detectors, increases p and r. This is borne out by the data in Table 6.5. This twin improvement of accuracy and time is partly because all of the detectors run slightly slower than the frame rate. A faster detector will usually improve accuracy, as it sees more frames. For a hypothetical less accurate detector which was faster than the frame rate, such an observation would no longer apply.

The data of interest in Table 6.9 are mean estimated power consumption above baseline; there is 29W range in average power consumption between highest-power and lowest-power priority settings. When the system was run with speed prioritised with the FPGA turned off, power consumption was 208W, or 62W above baseline. The average power for auto-prioritised mode with the FPGA on was 61.9W, but this is dependent on the dataset used; datasets with fewer moving objects would have a lower average power consumption than this. This is also due to the lack of *cff* and *gff* implementations for the car HOG detector; these would reduce average power consumption below this number.

As Figure 6.19 shows, running the system in automatic prioritisation mode allows an increase in accuracy of 10% over the lowest-power option for a cost of 12W in average power consumption. A further 17% gain in F_1-score (from *auto* to *speed*) costs an extra 17W above baseline. These results show a clear relationship between power consumption and overall detection accuracy.

Figure 6.20 presents these results in relative and absolute terms, showing the relative decreases in accuracy from *speed* when moving to lower-power prioritisations, and using $F_1(speed)$ as a baseline. The 12W power reduction at *auto* reduces accuracy by 45 % of baseline, while the fully-optimised *power* option loses 72% accuracy for 32% in power savings from best-case.

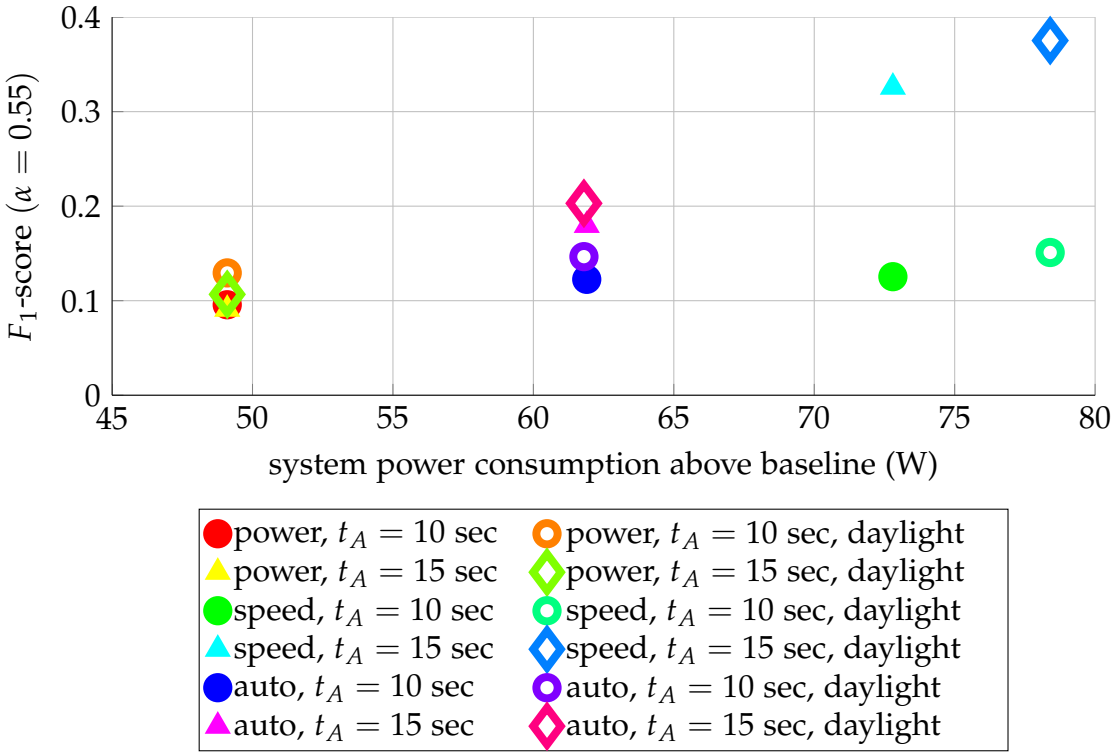
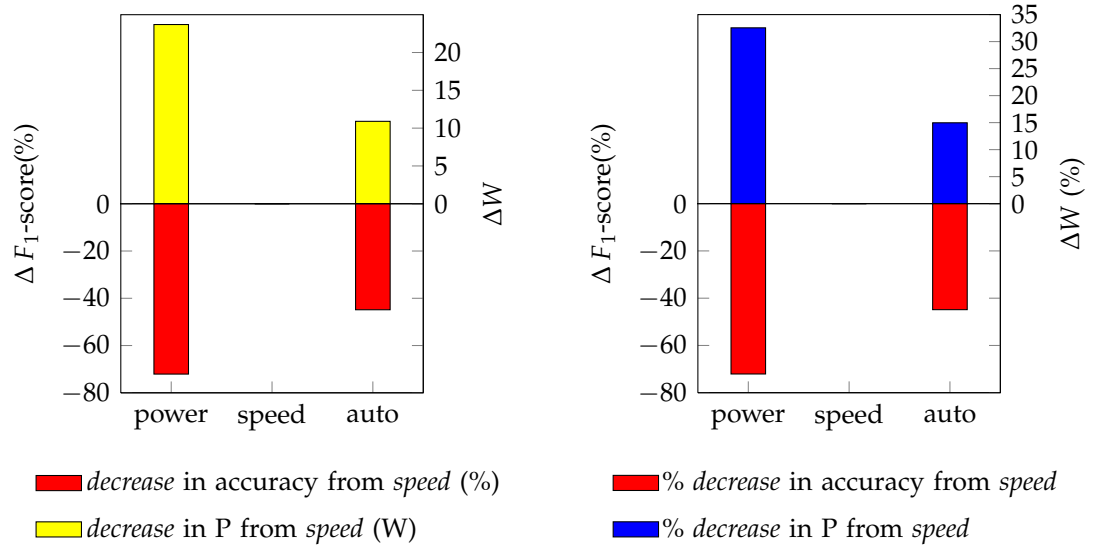


Figure 6.19: F_1-scores for operational awareness ($\alpha = 0.55$) against power consumption, for various anomaly time periods on i-LIDS *PV*3.

6.9.1 Comparison to State-of-the-Art

Some previous work has considered four clips made publicly available from i-LIDS, known as AVSS and containing four parked-vehicle events, classed as easy, medium, hard and night. The only work we are aware of which evaluates the complete i-LIDS dataset is by Albiol *et al.* [117], as discussed in Section 6.3. Using spatiotemporal maps and manually-applied lane masks per-clip to denote areas in which detections are allowed, Albiol *et al.* are able to improve significantly on the precision and recall figures given above, reaching $p-$ and $r-$values of 1.0 for several clips in PV3. They do not provide performance information but note that they downscale the images to 320×240 to decrease evaluation time. In their work they discuss the applicability and limitations of background subtractors for detecting slow-moving and stopped objects, as well as discussing the same difficulties with the i-LIDS data previously seen in Figure 6.17.

However, we also consider real-time implementation and power consumption, with an end goal of a fully automatic system. Unlike Albiol *et al.*'s requirement for



(a) *Absolute* decrease in power consumption and relative decrease in F_1-score of each configuration compared to *speed* prioritisation.

(b) *Relative* decrease in power consumption and relative decrease in F_1-score of each configuration compared to *speed* prioritisation.

Figure 6.20: Relative tradeoffs between system power consumption above baseline and F_1-score. In each case, a bar further from zero in either direction indicates *decreased* accuracy or power consumption, and $F_1(speed) = 100\%$.

manual operator intervention, we only need to re-register videos between clips to overcome camera movement, which could be done automatically.

If we only consider our detector accuracy compared to those of other researchers working on this data, we cannot improve upon existing results. The most obvious way to do so would be to use the currently best-performing object classifiers, but implementing multiple heterogeneous versions of these would take a prohibitively long time to develop. Any accuracy measurement must be traded against other characteristics, as Figure 6.19 shows.

We set out, however, to tackle the novel and different problem of power-aware anomaly detection rather than offline lane masking. To the best of our knowledge, no other work has been presented in this area.

6.9.2 System Architecture Improvements

An exhaustive search method was used for exploring the search space, and this was suitable given the relatively low number of 24 combinations (1 background subtraction implementation $\times$ 6 pedestrian detectors $\times$ 4 car detectors, as in Figure 6.15) in use as candidates to be mapped. (Note that this design space is *much* smaller than the work referred to in Section 2.4). However, this does not adequately capture any interactions or commonalities between detectors. Generic, detector-independent operations such as colour-to-grayscale conversion and transferring of frames between CPU and GPU are cached, *i.e.* are not repeated for separate detectors run in a single frame. Additional common steps are not removed; for example, the "resize" step is repeated for both pedestrian and car detectors for all n scales. FPGA-specific steps are also not identified and pipelined: the source image at each scale is run through the FPGA separately for each detector. This could be streamlined to generate window scores in parallel for multiple detectors (as described in Komorkiewicz *et al*. [154]) for pedestrians and cyclists, at the cost of additional memory or DMA controller logic. Allowing these common steps to be identified and cached would decrease processing time considerably, but at the expense of complexity, both of the image pipeline processing system and the mapping algorithm. The behaviour and performance characteristics of one detector would now depend on its own performance, plus whichever other algorithms are running during that frame. This would also require additional complexity during the design of each detector implementation. One interesting extension of this is to treat each image transport, conversion or resizing step as a separate algorithm stage, define prerequisites based on these for each processing stage and allow the allocation algorithm to generate a connected, optimised pipeline, in a similar manner to Quinn *et al*.'s firmware and software selection mechanism [60].

Such interaction between a potentially much larger permutation of algorithm implementations may then require a more optimised search algorithm, such as tabu search [126], a genetic algorithm or dynamic programming techniques [60], although the runtime of such an allocation algorithm over large search spaces may start to become significant compared to the runtime of the detectors themselves.

Running separate algorithms in parallel simultaneously on different accelerators is another plausible improvement which would reduce processing time, possibly at the expense of power consumption, although we already multithread the *cfc*

detectors at different scales. Currently, multiple implementations of different object detectors running on GPU must share the same constant memory, (*e.g.* detectors having different descriptor feature lengths) which means that detectors for different objects cannot be run simultaneously.

6.9.3 Algorithm-Specific Improvements

A higher-level improvement which could be applied is to swap one or more of the detectors for state-of-the art versions, for example the `VeryFast` pedestrian detector [83], which shows both improved detection accuracy and faster runtime, at the cost of flexibility (the current algorithm is GPU-only). Similarly, the car classifier could be swapped in a similar manner, or further detectors trained on more object classes could be added. (We currently do not aim to detect buses, trucks, vans and cyclists, instead relying on motion detection or misclassifications from other object detectors). The most significant limiting factor here is the time required to write an implementation of any algorithm, on one or more target platforms. The current system is written to support this, and object detection algorithms can be easily added provided performance metadata is known.

6.9.4 Task-Level Improvements

As discussed in this chapter, contextual information can be used to detect objects behaving anomalously. Letham *et al*. [130] also show that it can be used to increase the quality of existing detections, by removing false detections in unusual areas; this could possibly be combined with the score or confidence-measure associated with each detected object.

Finally, specifically considering this anomaly detection scenario, improved tracking, contextual awareness and anomaly detection mechanisms could be considered. (Recall that we described this as an example of a real-world computer vision problem which we have applied a heterogeneous system to, rather than having set out to produce a state-of-the-art anomaly detector). Compromises have been made at multiple levels of abstraction to attempt to mitigate the relatively poor performance of the object classifiers and background subtraction mechanism. The latter is severely affected by camera shake and changes in camera gain. The relatively high false positive rate of all detectors resulted in compromises being made to the tracker; with more accurate detectors the tracker could be made more sensitive,

allowing stationary or occluded objects to be tracked for much longer with a lower risk of false alarms from false detections believed to be anomalous. The Bayesian motion-based anomaly detection is also heavily affected by the false positives from the classifiers, and the trajectory clustering mechanism does not properly handle lost tracks in the middle of a scene (properly tracking the trajectory of objects is as task for the object tracker rather than the clustering mechanism).

In the i-LIDS scenarios described above, detecting parked vehicles is one part of the challenge; the other is to look for pedestrians subsequently entering or leaving those vehicles. An interesting addition would be to expand the system to look for such events, although this would probably require some of the baseline improvements described above, and possibly require careful prioritisation of tasks per-frame, possibly dropping some requested algorithms to better fit the available processing time.

6.10 Conclusion

This chapter has described a system of heterogeneous processors used for performing a real-world computer vision task: real-time anomaly detection in video. The system adjusts the platforms it runs on and the power it uses in response to scene content. While there are various problems with the implementation, as described in the previous section, we were able to detect some of the "parked vehicle" anomalous events as categorised by the UK Home Office in the i-LIDS dataset, and show the effect of using the presence or absence of these events in the video to dynamically adjust system performance characteristics. We compared this to a single-accelerator model and showed that power consumption for the automatically-prioritised system, was approximately equal to the single-accelerator model for the dataset used. We were also able to show a clear link between overall detection accuracy and system power consumption. To our knowledge, other work in this area has not been presented.

This chapter has thus answered the second question posed in the Introduction chapter: *"does the optimal mapping of a set of algorithms to a heterogeneous set of processors change over time, and does such a system offer any advantage in a real-world image processing task?"*

The answers to this are "yes — if dynamically-adjusted priorities are used to weight the mapping function," and "performance does not improve on that of a single-accelerator system *on this dataset*, but this is dataset-dependent". This answer is affected by the algorithm implementations available, such as the lack of *cff* and *gff* car detectors, each of which we would expect to be faster, lower-power and less accurate than their *cfc* and *gfg* equivalents. This leads to a broader discussion next chapter about design implementation times, something which has not hitherto been considered here.

This forms the conclusion of the experimental work in this thesis. In the following, final chapter, we discuss outcomes from this work and directions for future research.

7. Conclusion

7.1 Summary

The overall aim of this thesis was to answer, in detail, two sets of related questions: *"how does the performance of an algorithm when partitioned temporally across a heterogeneous array of processors compare to the performance of the same algorithm in a singly-accelerated system, when considering a real-world image processing problem?"* and *"what is the optimal mapping of a set of algorithms to a heterogeneous set of processors? Does this change over time, and does a system with this architecture offer any advantage in a real-world image processing task?"*.

We provide the justification for asking these questions in the Introduction (1), considering this from both an academic and industrial angle; here we noted the commercial benefits to Thales of having a detailed technical study of performance on heterogeneous architectures available. As this is research within an engineering discipline and with an industrial component, it is done with a clear application in mind: that of improving scene understanding in the context of scene surveillance and situational awareness from mobile vehicles, operating under time and power constraints.

Advancing research in this area is a priority for the UK Ministry of Defence, as shown by a 2013 award of grants totalling £16m to UK universities, including Heriot-Watt and Edinburgh, to investigate signal processing in the networked battlespace. A portion of this will focus on anomaly detection in video and other modalities and on reducing the size, weight and power of sensing devices in the field – topics which have been addressed by this work.

In the Background, we place this work in an academic context. First, using an existing processor taxonomy, we provide details necessary to understand the basic

architecture of the platforms we used, and discuss alternative architectures which we considered. We go on to consider image processing algorithms relevant to the problem of scene understanding.

Concentrating specifically on object detection, we chart the progress of the state-of-the-art algorithms in human detection, focusing mainly on Histogram of Oriented Gradients, its derivatives and implementations. We conclude Chapter 2 by bringing these two disciplines together, by discussing work on the automatic mapping of algorithms to hardware, both at design and runtime. After analysing previous work on performance comparisons between FPGA and GPU along with design space exploration work, we argue that there is as yet no clear overall roadmap for selecting processing architectures for an application based on performance alone, and other factors such as power consumption must be considered. Under such constraints, dynamic mapping within a heterogeneous processing architecture becomes more attractive.

In Chapter 3 we discussed sensor inputs and explored possibilities for indirectly improving performance by confining object detection work to specific image regions through segmentation. We described an alternative architecture, Ter@pix, before choosing FPGA and GPU. We also considered the feasibility of simulating the heterogeneous processing system we hoped to build, before ultimately choosing a hardware-focused approach. We described design space exploration methods operating on simple segmentation algorithms. These early explorations then influenced the design of our PC-based system, which we documented in Chapter 4. Here we also note the architectural mechanism and issues involved when transferring data between accelerators.

Chapter 5 explores in depth the HOG algorithm and the possibilities available when implementing it across multiple platforms. We describe the multiple methods for partitioning algorithm stages across heterogeneous processing architectures, and the tradeoffs inherent in each one. The outcome of this, as well as a real-time pedestrian detector, was a quantised set of tradeoffs of power, speed and accuracy for a particular algorithm, with the understanding that such tradeoffs would generalise over other detection algorithms. This, then, answered our first question: *a heterogeneous processor array with tasks partitioned temporally does not outperform a single-processor array when only processing time is considered, but offers lower power consumption while maintaining competitive performance, thus moving along a*

Pareto-optimal curve. The *when partitioned temporally* qualification here is important; other approaches such as processing larger images on FPGA and higher scales on GPU are not considered in this work, as we concentrated on the placement of discrete pipeline stages. In addition, this conclusion depends to an extent on implementation details, the most significant of which is the FPGA PCIE interface. This underscores the importance of efficient methods of data transfer between processing platforms: while systems with more optimised data transfer interfaces might perform better then the one we describe, any improvement in power or time would come at the cost of increased design and implementation time.

Finally, in Chapter 6, we turned to the problem of anomaly detection in a scene, using detection of parked vehicles as an example. With the stated goal of reducing power consumption during idle video sequences while still retaining the capability to generate fast, accurate detections in the presence of anomalies, we used our heterogeneous system to detect multiple object classes in real time. We defined a scene anomaly level generated by trajectory clustering and motion-based object context, and were able to use that to recalculate the optimal mapping between algorithms and architecture. In the process, we addressed our second question: *the optimal mapping between architectures and algorithms changes over time, and is affected by system performance priorities. On the video sequences used, performance was comparable to that of a single-accelerator system, but this was heavily dependent on the dataset.* We are unaware of any other such framework for dynamically reallocating tasks between heterogeneous processors when power consumption is constrained and real-time operation must be maintained.

7.2 Contributions

The original contributions of this work are, in summary:

- We give a comprehensive analysis of the performance of a complex signal processing algorithm when applied to a platform with multiple heterogeneous accelerators (FPGA and GPU). Having considered processing time, power consumption and accuracy, we show the cost (in percentage change from best measurement for that characteristic) of trading one of these against the other. This work ([16]) was published in the Journal of Emerging and Selected Topics

in Circuits and Systems in 2013 and a talk and paper was presented at the "Smart Cameras for Robotic Applications" workshop, part of IROS, in 2012 [15].

- We construct and describe a real-time image processing system for anomaly detection which dynamically modifies the processing elements it runs on and hence its power consumption characteristics in response to events within a scene. To the best of our knowledge, there is no other literature on dynamic switching between heterogeneous accelerators under real-time constraints. Work describing this was accepted for presentation at VISAPP 2014 [17] as a full paper, with a 2013 acceptance rate of 9%.

7.2.1 Outcomes

In addition to the publications from the original contributions above, we also produced two other outcomes:

- A real-time technology demonstrator was produced and exhibited at a Thales research day.

- An invited talk describing the work done in chapter 5 was given at a BMVA Symposium on "Vision in an Increasingly Mobile World".

7.3 Future Research Directions and Improvements

More work, funded by significant Networked Battlespace grants from DSTL, (the UK MoD research department), and EPSRC, is now being undertaken to improve the confidence and real-time performance of classifiers such as HOG; this is a continuation of some more widely applicable parts of this current work. However, there are several interesting possibilities for extending the work documented in this thesis in different directions. These are listed below and should be considered in conjunction with the improvements given in Section 6.9.

One interesting direction would be to greatly improve the monitoring of power consumption of the system to one which has much more granularity (*i.e.* FPGAs have built in power measurement tools, as do later generations of GPU). Alternatively,

focusing on measurement of energy rather than power may be instructive: is higher power consumption for a shorter duration an acceptable tradeoff, and does this hold for all system sizes?

The most significant architectural bottleneck discussed in Chapter 5 was transfer times between architectures. Current developments allow closer integration between CPU and accelerator, with both elements being available on one chip. Some platforms now include a fast CPU, reconfigurable logic and a simple manycore array on one chip; the application and constraints described in this thesis would be rewarding to investigate on such a platform. More broadly, any such base platform should be sufficiently mature that algorithms can be implemented on to it without encountering limitations due to memory interfaces such as DMA or external RAM. This implies using a system based on Bittner's work [14], or at a minimum, a robust, high-performance DMA controller and SDRAM interface on FPGA.

This point naturally leads to a discussion about tools. Current model-based design tools on FPGA lack automatic or minimally-customisable mechanisms for incorporating an interface to external subsystems; this should be addressed. Alternatively, the ability to exploit parallelism within an algorithm and write a single implementation in a high-level language such as OpenCL and have that automatically compile to multiple architectures such as FPGA and GPU is a much broader general research problem, one with lots of potential both academically and industrially. Current-generation autocoding tools for FPGA are not a complete solution. For simple streaming pixel-in, pixel-out processing with no need for recall of previous frames, autocoding an algorithm in Simulink might be a trivial task. However, for involved algorithms requiring any level of short- or long-term data storage (and thus instantiation of memories), fine control over finite state machines, or manual pipelining, autocoding *reduces the time to capture design elements of, but does not remove the need for an understanding of* the underlying hardware.

Finally, a natural continuation of the current work would be to port it to an embedded system with a massively reduced power consumption, particularly when idle. Nvidia's forthcoming Logan chipset has a stated power consumption of 2W, while being able to run standard CUDA code. Implementation of the closed-loop surveillance system described in Chapter 6 on such a platform would provide a compelling validation of the work described in this thesis.

A. Mathematical Formulæ

A.1 Vector Norms

For a vector $\mathbf{v}$, normalised vectors $\mathbf{nv}_{L_1Sqrt}$ and $\mathbf{nv}_{L_2Hys}$ are defined as follows. ϵ is added to prevent division by zero.

$$\mathbf{nv}_{L_1Sqrt} \equiv \sqrt{\frac{\mathbf{v}}{\| \mathbf{v} \|_1 + \epsilon}} \tag{A.1}$$

For $\mathbf{nv}_{L_2Hys}$, we first find

$$\mathbf{v}' \equiv \frac{\mathbf{v}}{\sqrt{\sum \| \mathbf{v} \|_2^2 + \epsilon}}, \tag{A.2}$$

then cap $\mathbf{v}'$ at a constant V_{max}, then

$$\mathbf{nv}_{L_2Hys} \equiv \frac{\mathbf{v}'}{\sqrt{\sum \| \mathbf{v}' \|_2^2 + \epsilon}}. \tag{A.3}$$

A.2 Kalman Filter

Given knowledge of the state of an object x at time t_{k-1} (in this case, position and velocity in x and y in $2D$ space), its state at time t_k can be *predicted* given some knowledge or assumptions about the process causing x to move:

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \tag{A.4}$$

$$P_k^- = AP_{k-1}A^T + Q. \tag{A.5}$$

Here, $\hat{x}_k^-$ is the predicted state of x at t_k, $\hat{x}_{k-1}$ is the *a posteriori* state at t_{k-1}, A is the state transition matrix, B is the coupling between process noise and state, and u_{k-1} is the control input to the system (zero in this case). P_k^- is the estimated *a priori* error covariance and Q is the process noise covariance matrix.

Once the prediction step is completed, a detection position can be matched to the position-components of $\hat{x}_k^-$. If a match is made, the state is then *corrected* with the updated measurement $z_k = (x_d, y_d)$ to produce the *a posteriori* state estimate $\hat{x}_k$:

$$K_k = P_k^- H^T / (H P_k^- H^T + R) \tag{A.6}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \tag{A.7}$$

$$P_k = P_k^- - K_k H\,. \tag{A.8}$$

Where K_k is the Kalman gain, H relates measurements taken to the state and R is measurement noise covariance.

A.3 Planar Homography

Bradski and Kaelher define planar homography as "a projective mapping from one plane to another" [155]. This is used to project points from a camera plane to a ground plane to obtain a 'birds-eye view'. To move to a point in a plane $p_{dst} = [x_{dst}, y_{dst}]$ from a point in a plane $p_{src} = [x_{src}, y_{src}]$, the following equation is used:

$$p_{dst} = H p_{src} \tag{A.9}$$

where

$$\begin{bmatrix} x_{dst} \\ y_{dst} \\ 1 \end{bmatrix} = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{bmatrix} \begin{bmatrix} x_{src} \\ y_{src} \\ 1 \end{bmatrix} \tag{A.10}$$

and the 9-entry homography matrix H contains homography data learned from control points.

Bibliography

[1] J. S. Warm, R. Parasuraman, and G. Matthews, "Vigilance Requires Hard Mental Work and Is Stressful," *Hum. Factors J. Hum. Factors Ergon. Soc.*, vol. 50, no. 3, pp. 433–441, Jun. 2008.

[2] N. H. Mackworth, "The breakdown of vigilance durning prolonged visual search," *Q. J. Exp. Psychol.*, vol. 1, no. 1, pp. 6–21, Apr. 1948.

[3] F. M. Donald, "The classification of vigilance tasks in the real world." *Ergonomics*, vol. 51, no. 11, pp. 1643–55, Nov. 2008.

[4] Volvo Car Group, "Press Release: Pedestrian Detection with full auto brake - unique technology in the all-new Volvo S60," 2010. [Online]. Available: https://www.media.volvocars.com/global/en-gb/media/pressreleases/31773

[5] M. Mathias, R. Timofte, R. Benenson, and L. Van Gool, "Traffic sign recognition - How far are we from the solution?" in *Int. Jt. Conf. Neural Networks*, Dallas, Aug. 2013.

[6] DSTL, "UDRC Technical Challenges," 2013. [Online]. Available: http://www.see.ed.ac.uk/drupal/udrc/technical-challenges/

[7] Euro NCAP, "Autonomous Emergency Braking," 2013. [Online]. Available: http://www.euroncap.com/results/aeb.aspx

[8] Royal Canadian Mounted Police., "Single Vehicle Rollover – Saskatoon RCMP Search for Injured Driver with Unmanned Aerial Vehicle," 2013. [Online]. Available: http://www.rcmp-grc.gc.ca/sk/news-nouvelle/video-gallery/video-pages/search-rescue-eng.htm

[9] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, p. 54, Sep. 2005.

[10] W.-M. W. Hwu and D. B. Kirk, *Programming Massively Parallel Processors*. Chicago: Morgan Kauffman, 2010.

[11] H. Meuer, E. Strohmaier, H. Simon, and J. Dongarra, "Top 500 Supercomputers: Highlights - November 2013," 2013. [Online]. Available: http://www.top500.org/lists/2013/11/highlights/

[12] Ministry of Defence, "Defence Standard 00-82 Vetronics Infrastructure for Video Over Ethernet Part 0 : Guidance," Ministry of Defence, Tech. Rep. 2, 2012.

[13] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *2008 SC - Int. Conf. High Perform. Comput. Networking, Storage Anal.* IEEE, Nov. 2008, pp. 1–11.

[14] R. Bittner, E. Ruf, and A. Forin, "Direct GPU/FPGA communication Via PCI express," *Cluster Comput.*, no. February, Jun. 2013.

[15] C. Blair, N. M. Robertson, and D. Hume, "Characterising Pedestrian Detection on a Heterogeneous Platform," in *Work. Smart Cameras Robot. Appl. (Scabot '12), IROS 2012*, 2012.

[16] ——, "Characterising a Heterogeneous System for Person Detection in Video using Histograms of Oriented Gradients: Power vs. Speed vs. Accuracy," *IEEE J. Emerg. Sel. Top. Circuits Syst.*, vol. 3, no. 2, pp. 236–247, 2013.

[17] C. G. Blair and N. M. Robertson, "Event-Driven Dynamic Platform Selection for Power-Aware Real-Time Anomaly Detection in Video," in *Int. Conf. Comput. Vis. Theory Appl. (VISAPP 2014)*, Lisbon, Jan. 2014.

[18] G. J. Awcock and R. Thomas, *Applied Image Processing*. Basingstoke: Macmillan New Electronics, 1995.

[19] P. Viola and M. Jones, "Robust real-time object detection," *Int. J. Comput. Vis.*, vol. 57, no. 2, pp. 137–154, 2002.

[20] M. Rahman, J. Ren, and N. Kehtarnavaz, "Real-time implementation of robust face detection on mobile platforms," in *Proc. 2009 IEEE Int. Conf. Acoust. Speech Signal Process.* IEEE Computer Society, 2009, pp. 1353–1356.

[21] M. Zuluaga and N. Topham, "Design-space exploration of resource-sharing solutions for custom instruction set extensions," *Trans. Comput. Des. Integr. Circuits Syst.*, vol. 28, no. 12, pp. 1788–1801, 2009.

[22] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research : A View from Berkeley," University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[23] D. B. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," *Int. Symp. F. Program. Gate Arrays*, pp. 63–72, 2009.

[24] A. R. Brodtkorb, C. Dyken, T. R. Hagen, and J. M. Hjelmervik, "State-of-the-art in heterogeneous computing," *Sci. Program.*, vol. 18, pp. 1–33, 2010.

[25] C. Moler, "Matrix computation on distributed memory multiprocessors," in *Hypercube Multiprocessors*, 1986.

[26] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. April 18–20 1967 spring Jt. Comput. Conf.*, vol. 30, no. 3, pp. 483–485, 1967.

[27] C. Johnston, K. Gribbon, and D. Bailey, "Implementing image processing algorithms on FPGAs," in *Proc. Elev. Electron. New Zeal. Conf. ENZCon'04*, 2004, pp. 118–123.

[28] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.

[29] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," in *Comput. Graph. Forum*, vol. 26, no. 1. Citeseer, 2007, pp. 80–113.

[30] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[31] G. Hendeby, J. Hol, R. Karlsson, and F. Gustafsson, "A graphics processing unit implementation of the particle filter," in *Proc. Eusipco 2007*, vol. 1, no. Eusipco, 2007, pp. 1639–1643.

[32] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs," 2011.

[33] V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," *2008 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.*, pp. 1–8, Jun. 2008.

[34] V. Pham, P. Vo, and V. Hung, "GPU implementation of Extended Gaussian mixture model for Background subtraction," in *Comput. Commun. Technol. Res. Innov. Vis. Futur. (RIVF), 2010 IEEE RIVF Int. Conf.* IEEE, 2008, pp. 1–4.

[35] B. Sharma, R. Thota, N. Vydyanathan, and A. Kale, "Towards a robust, real-time face processing system using CUDA-enabled GPUs," *2009 Int. Conf. High Perform. Comput.*, pp. 368–377, Dec. 2009.

[36] L. Polok, A. Herout, P. Zemčík, M. Hradiš, R. Juránek, and R. Jošth, "'Local Rank Differences' Image Feature Implemented on GPU," in *Adv. Concepts Intell. Vis. Syst.* Springer, 2008, pp. 170–181.

[37] M. Boyer and D. Tarjan, "Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors," in *Parallel & Distrib. Comput. Symp.*, no. May, 2009.

[38] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Realtime Computer Vision with OpenCV," *Queue*, vol. 10, no. 4, p. 40, Apr. 2012.

[39] G. Wang, B. Rister, and J. Cavallaro, "Workload Analysis and Efficient OpenCL-based Implementation of SIFT Algorithm on a Smartphone," in *IEEE Glob. Conf. Signal Inf. Process.*, no. December, 2013.

[40] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel Processing in a Restructurable Computer System," *IEEE Trans. Electron. Comput.*, vol. EC-12, no. 6, pp. 747–755, Dec. 1963.

[41] Xilinx, "MicroBlaze Processor Reference Guide," 2012.

[42] S. Lu, P. Yiannacouras, T. Suh, R. Kassa, and M. Konow, "A Desktop Computer with a Reconfigurable Pentium," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 1, pp. 1–15, 2008.

[43] D. Bacon, R. Rabbah, and S. Shukla, "FPGA Programming for the Masses," *Queue*, vol. 11, no. 2, p. 40, Feb. 2013.

[44] D. G. Bailey and C. T. Johnston, "Algorithm Transformation for FPGA Implementation," in *2010 Fifth IEEE Int. Symp. Electron. Des. Test & Appl.* IEEE, 2010, pp. 77–81.

[45] R. Zoss, A. Habegger, V. Bandi, J. Goette, and M. Jacomet, "Comparing signal processing hardware-synthesis methods based on the Matlab tool-chain," in *2011 Sixth IEEE Int. Symp. Electron. Des. Test Appl.* IEEE, 2011, pp. 281–286.

[46] A. Papakonstantinou, K. Gururaj, J. a. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," *2009 IEEE 7th Symp. Appl. Specif. Process.*, pp. 35–42, Jul. 2009.

[47] S. Edwards, "The Challenges of Hardware Synthesis from C-Like Languages," in *Des. Autom. Test Eur.* IEEE, 2005, pp. 66–67.

[48] ——, "The challenges of synthesizing hardware from C-like languages," *Des. & Test Comput. IEEE*, vol. 23, no. 5, pp. 375–386, 2006.

[49] Xilinx, "Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries," Xilinx, Tech. Rep. 1167, 2013.

[50] K. Underwood, "From Silicon to Science: The Long Road to Production Reconfigurable Supercomputing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 4, 2008.

[51] D. H. Jones, A. Powell, C.-S. Bouganis, and P. Y. Cheung, "GPU Versus FPGA for High Productivity Computing," *2010 Int. Conf. F. Program. Log. Appl.*, pp. 119–124, Aug. 2010.

[52] J. Svab, T. Krajnik, J. Faigl, and L. Preucil, "FPGA based Speeded Up Robust Features," in *2009 IEEE Int. Conf. Technol. Pract. Robot Appl.* IEEE, Nov. 2009, pp. 35–41.

[53] G. N. Gaydadjiev, N. T. Quach, and B. Zafarifar, "Real-time FPGA-implementation for blue-sky Detection," in *IEEE 18th Int. Conf. Appl. Syst. Archit. Process.*, 2007, pp. 76–82.

[54] M. Hiromoto, K. Nakahara, H. Sugano, Y. Nakamura, and R. Miyamoto, "A Specialized Processor Suitable for AdaBoost-Based Detection with Haar-like Features," in *2007 IEEE Conf. Comput. Vis. Pattern Recognit.* IEEE, Jun. 2007, pp. 1–8.

[55] Altera Corporation, "White Paper FPGA Run-Time Reconfiguration : Two Approaches," Altera, Tech. Rep. March, 2008.

[56] Xilinx, "Partial Reconfiguration User Guide," Xilinx, Tech. Rep. 702, 2012.

[57] I. Colwill, "Multi Agent System Platform in Programmable Logic," DPhil Thesis, Sussex, 2008.

[58] F. Fons, M. Fons, E. Cantó, and M. López, "Real-time embedded systems powered by FPGA dynamic partial self-reconfiguration: a case study oriented to biometric recognition applications," *J. Real-Time Image Process.*, pp. 1–23, 2011.

[59] L. Gantel, S. Layouni, M. E. a. Benkhelifa, F. Verdier, and S. Chauvet, "Multiprocessor Task Migration Implementation in a Reconfigurable Platform," *2009 Int. Conf. Reconfigurable Comput. FPGAs*, pp. 362–367, Dec. 2009.

[60] H. Quinn, M. Leeser, and L. Smith King, "Dynamo: a runtime partitioning system for FPGA-based HW/SW image processing systems," *J. Real-Time Image Process.*, vol. 2, no. 4, pp. 179–190, 2007.

[61] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *F. Program. Log. Appl. 2006. FPL'06. Int. Conf.*, Xilinx. IEEE, 2006, pp. 1–6.

[62] M. Happe, E. Lübbers, and M. Platzner, "A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking," *J. Real-Time Image Process.*, vol. 8, no. 1, pp. 95–110, Jul. 2011.

[63] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D.-J. Lee, "Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study," in *16th Int. Symp. Field-Programmable Cust. Comput. Mach.* IEEE, 2008, pp. 173–182.

[64] B. Cope, P. Y. Cheung, W. Luk, and L. Howes, "Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study," *IEEE Trans. Comput.*, vol. 59, no. 4, pp. 433–448, Apr. 2010.

[65] M. Papadonikolakis, C.-S. Bouganis, and G. Constantinides, "Performance comparison of GPU and FPGA architectures for the SVM training problem," in *2009 Int. Conf. Field-Programmable Technol.* IEEE, 2009, pp. 388–391.

[66] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *2009 Int. Conf. F. Program. Log. Appl.* IEEE, 2009, pp. 126–131.

[67] B. Cope, "Video processing acceleration using reconfigurable logic and graphics processors," PhD Thesis, imperial college, 2008.

[68] B. Cope, P. Cheung, W. Luk, and S. Witt, "Have GPUs made FPGAs redundant in the field of video processing?" in *Proceedings. 2005 IEEE Int. Conf. Field-Programmable Technol.* IEEE, 2005, pp. 111–118.

[69] C. Grozea, Z. Bankovic, and P. Laskov, "FPGA vs. Multi-core CPUs vs. GPUs: Hands-On Experience with a Sorting Application," *LNCS Facing Multicore-Challenge*, pp. 105–117, 2011.

[70] S. Bauer, U. Brunsmann, and S. Schlotterbeck-macht, "FPGA Implementation of a HOG-based Pedestrian Recognition System," in *MPC Work. Karlsruhe*, no. July, 2009.

[71] S. Bauer and S. Kohler, "FPGA-GPU architecture for kernel SVM pedestrian detection," in *Comput. Vis. Pattern Recognit. Work. (CVPRW), 2010 IEEE Conf.* IEEE, 2010, pp. 61–68.

[72] R. Bittner, E. Ruf, and A. Forin, "Direct GPU/FPGA Communication via PCI Express," *2012 41st Int. Conf. Parallel Process. Work.*, pp. 135–139, Sep. 2012.

[73] Intel Corporation, "History of Many-Core leading to Intel Xeon Phi," 2012.

[74] M. Everingham, L. Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) Challenge," *Int. J. Comput. Vis.*, vol. 88, no. 2, pp. 303–338, Sep. 2009.

[75] C. Cheng and C.-S. Bouganis, "An FPGA-based object detector with dynamic workload balancing," in *2011 Int. Conf. Field-Programmable Technol.* IEEE, Dec. 2011, pp. 1–4.

[76] D. Hefenbrock, J. Oberg, N. T. N. Thanh, R. Kastner, and S. B. Baden, "Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs," *2010 18th IEEE Annu. Int. Symp. Field-Programmable Cust. Comput. Mach.*, pp. 11–18, 2010.

[77] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Comput. Vis. Pattern Recognition, 2005.* IEEE Computer Society, 2005, pp. 886– 893.

[78] N. Dalal, "Finding People in Images and Videos," PhD Thesis, Institut National Polytechnique de Grenoble / INRIA Grenoble, 2006.

[79] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models." *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 9, pp. 1627–45, Sep. 2010.

[80] P. Dollár, Z. Tu, P. Perona, and S. Belongie, "Integral Channel Features." in *BMVC*, 2009, pp. 1–11.

[81] P. Dollar, S. Belongie, and P. Perona, "The Fastest Pedestrian Detector in the West," in *Procedings Br. Mach. Vis. Conf. BMVC 2010*, 2010, pp. 68.1–68.11.

[82] P. Dollar, C. Wojek, B. Schiele, and P. Perona, "Pedestrian Detection: An Evaluation of the State of the Art," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 4, pp. 743–762, Jul. 2011.

[83] R. Benenson and M. Mathias, "Pedestrian detection at 100 frames per second," in *Comput. Vis. Pattern Recognit. (CVPR), 2012 IEEE Conf.*, 2012, pp. 2903–2910.

[84] R. Benenson, M. Mathias, T. Tuytelaars, and L. Van Gool, "Seeking the strongest rigid detector," in *Proc. IEEE CVPR*, 2013.

[85] A. Martin, G. Doddington, and T. Kamm, "The DET curve in assessment of detection task performance," DTIC, Tech. Rep., 1997.

[86] R. Kadota and H. Sugano, "Hardware architecture for HOG feature extraction," in *Intell. Inf. Hiding Multimed. Signal Process. 2009. IIH-MSP'09. Fifth Int. Conf.* IEEE, 2009, pp. 1330–1333.

[87] S. Martelli, D. Tosato, M. Cristani, and V. Murino, "FPGA-based pedestrian detection using array of covariance features," in *Distrib. Smart Cameras (ICDSC), 2011 Fifth ACM/IEEE Int. Conf.*, 2011, pp. 1–6.

[88] P. E. P. Rybski, D. Huber, D. D. Morris, and R. Hoffman, "Visual classification of coarse vehicle orientation using Histogram of Oriented Gradients features," in *2010 IEEE Intell. Veh. Symp.* IEEE, Jun. 2010, pp. 921–928.

[89] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[90] C. Burges, "A tutorial on support vector machines for pattern recognition," *Data Min. Knowl. Discov.*, vol. 2, no. 2, pp. 121–167, 1998.

[91] C. Burges and e. Schölkopf, "Improving the accuracy and speed of support vector machines," in *Adv. Neural Inf. Process. Syst.*, 1997.

[92] S. Romdhani, P. Torr, B. Scholkopf, and A. Blake, "Efficient face detection by a cascaded support-vector machine expansion," *Proc. R. Soc. A Math. Phys. Eng. Sci.*, vol. 460, no. 2051, pp. 3283–3297, Nov. 2004.

[93] J. Platt, "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods," in *Adv. large margin Classif.*, 1999.

[94] M. Tipping, "Sparse Bayesian learning and the relevance vector machine," *J. Mach. Learn. Res.*, vol. 1, pp. 211—-244, 2001.

[95] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. University Press Group Limited, 2006.

[96] V. Prisacariu and I. Reid, "fastHOG-a real-time GPU implementation of HOG," Department of Engineering Science, Oxford University, Tech. Rep. 2310, 2009.

[97] P. Dollár, "Piotr's Image and Video Matlab Toolbox (PMT)," http://vision.ucsd.edu/~pdollar/toolbox/doc/index.html.

[98] M. Hiromoto and R. Miyamoto, "Hardware architecture for high-accuracy real-time pedestrian detection with CoHOG features," in *Comput. Vis. Work. (ICCV Work. 2009 IEEE 12th Int. Conf.* IEEE, Sep. 2009, pp. 894—-899.

[99] T. P. Cao and G. Deng, "Real-Time Vision-Based Stop Sign Detection System on FPGA," in *Digit. Image Comput. Tech. Appl.* IEEE, 2008, pp. 465–471.

[100] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "FPGA-Based Real-Time Pedestrian Detection on High-Resolution Images," *2013 IEEE Conf. Comput. Vis. Pattern Recognit. Work.*, pp. 629–635, Jun. 2013.

[101] B. Morris and M. Trivedi, "A Survey of Vision-Based Trajectory Learning and Analysis for Surveillance," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 8, pp. 1114–1127, Aug. 2008.

[102] M. S. Grewal and A. P. Andrews, *Kalman Filtering: Theory and Practice Using MATLAB*, 2nd ed. John Wiley & Sons, 2001.

[103] S. Atev, O. Masoud, and N. Papanikolopoulos, "Learning Traffic Patterns at Intersections by Spectral Clustering of Motion Trajectories," *2006 IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, pp. 4851–4856, Oct. 2006.

[104] X. Li, W. Hu, and W. Hu, "A Coarse-to-Fine Strategy for Vehicle Motion Trajectory Clustering," *18th Int. Conf. Pattern Recognit.*, pp. 591–594, 2006.

[105] B. Morris and M. Trivedi, "Trajectory learning for activity understanding: Unsupervised, multilevel, and long-term adaptive approach," *Pattern Anal. Mach. Intell.*, vol. 33, no. 11, pp. 2287–2301, 2011.

[106] C. Piciarelli and G. Foresti, "On-line trajectory clustering for anomalous events detection," *Pattern Recognit. Lett.*, vol. 27, no. 15, pp. 1835–1842, Nov. 2006.

[107] B. Morris and M. Trivedi, "Learning, modeling, and classification of vehicle track patterns from live video," *Intell. Transp. Syst. . . .*, vol. 9, no. 3, pp. 425–437, 2008.

[108] R. Khoshabeh, T. Gandhi, and M. M. Trivedi, "Multi-camera Based Traffic Flow Characterization & Classification," *2007 IEEE Intell. Transp. Syst. Conf.*, pp. 259–264, Sep. 2007.

[109] C. C. Loy, T. Xiang, and S. Gong, "Detecting and discriminating behavioural anomalies," *Pattern Recognit.*, vol. 44, no. 1, pp. 117–132, Jan. 2011.

[110] C. Stauffer and W. Grimson, "Adaptive background mixture models for real-time tracking," in *Proceedings. 1999 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 99, 1999, pp. 246–252.

[111] M. J. Roshtkhari and M. D. Levine, "Online Dominant and Anomalous Behavior Detection in Videos," in *IEEE Conf. Comput. Vis. Pattern Recognit.* Ieee, Jun. 2013, pp. 2611–2618.

[112] S. Sivaraman and M. M. Trivedi, "Looking at Vehicles on the Road: A Survey of Vision-Based Vehicle Detection, Tracking, and Behavior Analysis," *IEEE Trans. Intell. Transp. Syst.*, vol. 14, no. 4, pp. 1773–1795, Dec. 2013.

[113] T. Machida and T. Naito, "GPU & CPU cooperative accelerated pedestrian and vehicle detection," in *2011 IEEE Int. Conf. Comput. Vis. Work. (ICCV Work.* IEEE, Nov. 2011, pp. 506–513.

[114] M. S. Ryoo, M. Riley, and J. K. Aggarwal, "Real-time detection of illegally parked vehicles using 1-D transformation," in *2007 IEEE Conf. Adv. Video Signal Based Surveill.* Ieee, Sep. 2007, pp. 254–259.

[115] S. Boragno, B. Boghossian, J. Black, D. Makris, and S. Velastin, "A DSP-based system for the detection of vehicles parked in prohibited areas," *2007 IEEE Conf. Adv. Video Signal Based Surveill.*, no. 1, pp. 260–265, Sep. 2007.

[116] A. Bevilacqua and S. Vaccari, "Real time detection of stopped vehicles in traffic scenes," in *Adv. Video Signal Based Surveillance, 2007. AVSS 2007. IEEE Conf.* IEEE, 2007, pp. 1–5.

[117] A. Albiol, L. Sanchis, A. Albiol, and J. M. Mossi, "Detection of Parked Vehicles Using Spatiotemporal Maps," *IEEE Trans. Intell. Transp. Syst.*, vol. 12, no. 4, pp. 1277–1291, Dec. 2011.

[118] C. Galuzzi and K. Bertels, "The Instruction-Set Extension Problem: A Survey," *Lect. Notes Comput. Sci.*, vol. 4943, pp. 209–220, 2008.

[119] R. Niemann and P. Marwedel, "Hardware/software partitioning using integer programming," *Proc. ED&TC Eur. Des. Test Conf.*, pp. 473–479, 1996.

[120] M. Dhodhi, F. Hielscher, R. Storer, and J. Bhasker, "Datapath synthesis using a problem-space genetic algorithm," *Trans. Comput. Des. Integr. Circuits Syst.*, vol. 14, no. 8, pp. 934–944, 1995.

[121] M. Zuluaga and N. Topham, "Resource Sharing in Custom Instruction Set Extensions," in *2008 Symp. Appl. Specif. Process.* IEEE, 2008, pp. 7–13.

[122] O. Almer, M. Gould, N. Topham, and B. Franke, "Selecting the Optimal System : Automated Design of Application-Specific Systems-on-Chip," in *Proc. 4th Int. Work. Netw. Chip Archit.*, 2011.

[123] K. Kuchcinski, "Constraints-driven design space exploration for distributed embedded systems," *J. Syst. Archit.*, vol. 47, no. 3-4, pp. 241–261, Apr. 2001.

[124] C.-S. Bouganis, S.-B. Park, G. A. Constantinides, and P. Y. K. Cheung, "Synthesis and Optimization of 2D Filter Designs for Heterogeneous FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 4, pp. 1–28, 2009.

[125] H. Quinn, "Runtime tools for hardware/software systems with reconfigurable hardware," PhD Thesis, Northeastern University, Boston, 2004.

[126] U. Schneider, "A tabu search tutorial based on a real-world scheduling problem," *Cent. Eur. J. Oper. Res.*, Mar. 2010.

[127] K. Deb, "Multi-objective genetic algorithms: problem difficulties and construction of test problems." *Evol. Comput.*, vol. 7, no. 3, pp. 205–30, Jan. 1999.

[128] A. Thompson, P. Layzell, and R. Zebulum, "Explorations in design space: Unconventional electronics design through artificial evolution," *IEEE Trans. Evol. Comput.*, vol. 3, no. 3, pp. 167–196, 1999.

[129] B. Connor, I. Carrie, R. Craig, and J. Parsons, "Discriminative imaging using a LWIR polarimeter," in *Proc. SPIE*, vol. 7113, 2008.

[130] J. Letham, N. M. Robertson, and B. Connor, "Contextual smoothing of image segmentation," in *Int. Conf. Comput. Vis. Pattern Recognit. Work. CVPRW 2010*, San Francisco, 2010, pp. 7–12.

[131] J. Leskela, J. Nikula, and M. Salmela, "OpenCL embedded profile prototype in mobile device," in *Signal Process. Syst. 2009. SiPS 2009. IEEE Work.*, 2009, pp. 279–284.

[132] P. Bonnot, F. Lemonnier, G. Edelin, G. Gaillat, O. Ruch, and P. Gauget, "Definition and SIMD implementation of a multi-processing architecture approach on FPGA," *Des. Autom. Test Eur.*, pp. 610–615, 2008.

[133] NVidia Corporation, *CUDA Toolkit Reference Manual v5.0*. NVidia Corporation, 2012.

[134] J. A. Letham, "Context Based Image Segmentation," MRes Thesis, Heriot-Watt University, 2011.

[135] C. J. Tucker, "Red and Photographic Infrared linear Combinations for Monitoring Vegetation," *Remote Sens. Environ.*, vol. 150, pp. 127–150, 1979.

[136] K. Kluge and S. Lakshmanan, "A deformable-template approach to lane detection," in *Intell. Veh. 95 Symp. Proc.*, 1995, pp. 54–59.

[137] Y. Wang, N. Dahnoun, and A. Achim, "A Novel Lane Feature Extraction Algorithm Based on Digital Interpolation," in *Proc. Eusipco 2009*, vol. 1, no. Eusipco, 2009, pp. 480–484.

[138] J. Lee and C. Crane, "Road Following in an Unstructured Desert Environment Based on the EM(Expectation-Maximization) Algorithm," in *2006 SICE-ICASE Int. Jt. Conf.* Ieee, 2006, pp. 2969–2974.

[139] B. Zafarifar and P. de With, "Blue Sky Detection for Picture Quality Enhancement," in *Adv. Concepts Intell. Vis. Syst.* Springer, 2006, pp. 522–532.

[140] Intel Corporation, "Intel X58 Express Chipset Datasheet," 2009.

[141] PCI-SIG, "PCI Express 2.0 Base Specification," 2006.

[142] K. Lund, D. Naylor, and S. Trynosky, "Application Note 859: Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs: DDR2 SDRAM DMA Initiator Demonstration Platform," Xilinx Corporation, Tech. Rep. 859, 2008.

[143] T. Wilson, M. Glatz, and M. Hodlmoser, "Pedestrian detection implemented on a fixed-point parallel architecture," in *Consum. Electron. 2009. ISCE'09. IEEE 13th Int. Symp.* IEEE, 2009.

[144] NVidia Corporation, "Developing a Linux kernel module using RDMA for GPUDirect," Nvidia Corporation, Tech. Rep., 2012.

[145] K. Irick, M. DeBole, V. Narayanan, and A. Gayasen, "A Hardware Efficient Support Vector Machine Architecture for FPGA," in *16th Int. Symp. Field-Programmable Cust. Comput. Mach.* IEEE, Apr. 2008, pp. 304–305.

[146] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines," in *17th IEEE Symp. F. Program. Cust. Comput. Mach.* IEEE, 2009, pp. 115–122.

[147] NVidia Corporation, *CUDA C Best Practices Guide*. NVidia Corporation, 2012.

[148] UK Home Office, "Imagery Library for Intelligent Detection Systems - Detailed guidance," 2013. [Online]. Available: https://www.gov.uk/imagery-library-for-intelligent-detection-systems

[149] Z. Zivkovic, "Improved adaptive Gaussian mixture model for background subtraction," in *Proc. 17th Int. Conf. Pattern Recognition, 2004. ICPR 2004.*, vol. 2. IEEE, 2004, pp. 28–31.

[150] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," University of North Carolina at Chapel Hill, Tech. Rep. 1, 2006.

[151] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006, vol. 4.

[152] Home Office Centre for Applied Science and Technology, *Imagery Library for Intelligent Detection Systems (I-LIDS) User Guide*, 4th ed. UK Home Office, 2011.

[153] C. J. Van Rijsbergen, *Information Retrieval, 2nd edition*. Butterworth-Heinemann, 1979.

[154] M. Komorkiewicz, M. Kluczewski, and M. Gorgon, "Floating point HOG implementation for real-time multiple object detection," in *22nd Int. Conf. F. Program. Log. Appl.*, Aug. 2012, pp. 711–714.

[155] G. Bradski and A. Kaelher, *Learning OpenCV*. O'Reilly, 2008.