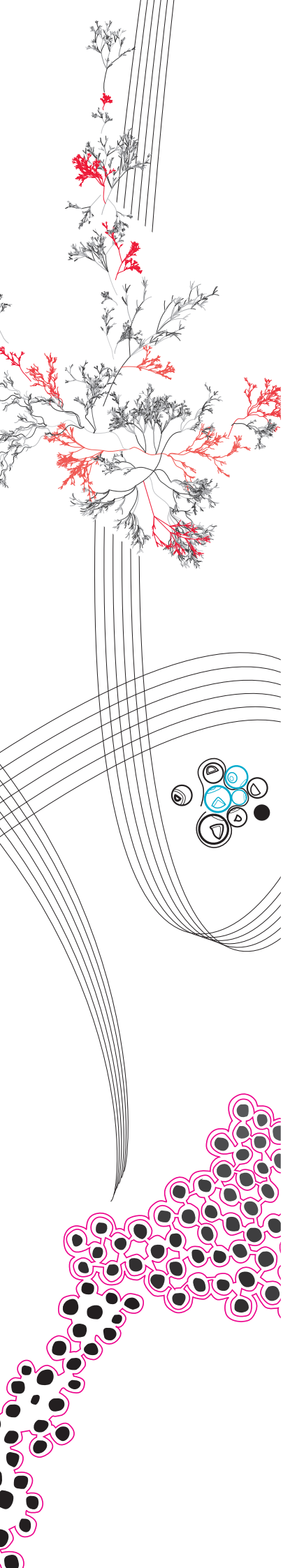


December 16, 2015

MASTER THESIS

IMPLEMENTATION AND ANALYSIS OF REAL-TIME OBJECT TRACKING ON THE STARBURST MPSOC

VIKTORIO S. EL HAKIM

Electrical Engineering, Mathematics and Computer Science (EEMCS)
Computer Architecture for Embedded Systems (CAES)

EXAMINATION COMMITTEE:
Prof. dr. ir. M.J.G. Bekooij
Ir. J. (Hans) Scholten
G. Kuiper, M.Sc.

Documentnumber
Centre for Telematics and Information Technology (CTIT) — 1

UNIVERSITY OF TWENTE.

University of Twente

MASTER THESIS

Implementation and Analysis of Real-Time Object Tracking on the Starburst MPSoC

Author: V.S.e. Hakim

Student number: s1415778

Committee: Prof. dr. ir. M.J.G. Bekooij

Ir. J. (Hans) Scholten

G. Kuiper, M.Sc.

Computer Architecture for Embedded Systems
(CAES), Department of EEMCS, University of Twente,
Enschede, The Netherlands
December 16, 2015

Abstract

Computer vision has experienced many advances over the recent years. As such, it started to rapidly find its way into many commercial and industrial applications, with the disciplines of visual object detection and tracking being most prominent. A good example of computer vision being applied in practice, is the social network and website Facebook, where advanced object detection algorithms are utilized to detect and recognize people and in particular – their faces. One of the main reasons for the rapid expansion of computer vision in practice, is the ever emerging new computing platforms, such as the cloud service, capable to handle the complex mathematics involved in analyzing and extracting information from images.

Despite its advances however, computer vision is still in its early stages of establishing a solid foothold in the world of embedded systems. Specifically, systems which are subjected to real-time constraints, with restricted computational resources, are struggling the most. Thus the usage of modern visual object detection and tracking algorithms in safety critical embedded systems is still more or less restricted. Fortunately, this is starting to change with newly developed embedded computer architectures, which employ application specific hardware to perform the task of computer vision more efficiently.

In this thesis, two state-of-the-art computer vision algorithms in the form of HOG-SVM detection and Particle filter tracking, are explored and evaluated on a real-time embedded MPSoC, called *Starburst*. Eventually, it can be shown that these two seemingly "difficult" algorithms, not only can satisfy certain real-time constraints, but also achieve a high throughput on an embedded system such as *Starburst*. To accomplish this, the thesis contributes with a powerful real-time hardware architecture of the HOG object detector, and a software based multiple processor object tracking framework, based on the Particle filter.

Both implementations are evaluated and tested on *Starburst*, to determine their respective real-time capabilities and whether imposed throughput constraints can be satisfied. Additionally, the functional behavior and accuracy of the implementations is also analyzed, but not to a full extent, since both algorithms are widely studied and documented in modern literature. The focus of this research is thus mainly on the temporal behavior.

Acknowledgements

First and foremost, I would like to thank Prof. Marco Bekooij, for giving me the opportunity to work under his supervision, on this interesting and challenging project. Additionally, I highly appreciate the amount of feedback and motivation he gave me, throughout the course of my master thesis assignment, as well as sparking my interest in real-time multiprocessor development.

Next, I would like to thank all of my colleges in the Pervasive Systems (PS) and CAES groups. Specifically I would like to thank Alex and Vignesh from the PS group, for the amount of time spent together, drinking coffee and discussing various scientific topics, and the additional feedback and suggestions provided about my thesis. Another special thank you goes to Oğuz, who kept me company during my research, and helped me on various occasions with the Xilinx toolchain.

Last but not least, I would like to thank all my family and friends, who supported me during my up and down moments. In particular, I would like to thank my dad, Semir, for encouraging and supporting me to do my master's study abroad, and my mom, Stella, for always being close to me when in trouble. Thank you both, you're the most wonderful parents in the world!

List of Abbreviations

ADC	Analog-to-Digital Converter
API	Application Program Interface
AR	Auto-Regressive
ARMA	Auto-Regressive Moving Average
ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Access Memory
CAES	Computer Architectures for Embedded Systems
CDF	Cumulative Distribution Function
CLB	Configurable Logic Block
CMOS	Complementary Metal–Oxide–Semiconductor
CORDIC	COordinate Rotation DIgital Computer
CPU	Central Processing Unit
CSDF	Cyclo-static Data-flow
DMA	Direct Memory Access
DSP	Digital Signal Processing
EKF	Extended Kalman Filter
ES	Embedded System
ET	Execution Time
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GPS	Global Positioning System
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
HOG	Histogram of Oriented Gradients
INRIA	Institute for Research in Computer Science and Automation
LIDAR	LIght Detection And Ranging
MB	Microblaze
MC	Monte Carlo
MPSoC	MultiProcessor System-on-Chip
MSB	Most Significant Bits
NoC	Network on Chip

PCA	Principle Component Analysis
PDF	Probability Density Function
PDP	Dot Product
PF	Particle Filter
PPF	Parallel Particle Filter
RGB	Red Green Blue
RMS	Root Mean Square
RMSE	Root Mean Squared Error
RNG	Random Number Generator
RT	Real-Time
RTOS	Real-Time Operating System
RTS	Real-Time System
SDF	Synchronous Data-flow
SIR	Sequential Importance Resampling
SMC	Sequential Monte Carlo
TDM	Time Division Multiplexing
TP	Throughput
UAV	Unmanned Aerial Vehicle
VGA	Video Graphics Array
WCET	Worst Case Execution Time

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Computer vision is an emerging discipline in computer science and electronic engineering, which strives at giving computers and machines the ability to perceive the environment in a similar way as humans do with the eyes. It covers a wide range of topics, such as machine learning, image and signal processing, video processing and control engineering. This invaluable ability allows machines to perform more intelligent tasks and thus provide a better service to their human operators. Computer vision has easily found its way into commercial applications, such as medical imaging, image search, medical robotics and others.

Nowadays, two of the major topics in computer vision are object recognition and tracking. Objects and features being extracted are used directly in the control systems of robots, smart vehicles, UAV's, smart traffic monitoring, etc, allowing intelligent control and decision making. Up until recently, the main means of object detection has mostly relied on other technologies such as LIDAR, since the incoming data is much easier to process in real-time. With the recent discoveries of new feature extractors and robust tracking algorithms, modern computer vision systems have shown perfomance comparable to that of traditional object detection systems, if not even better. One of the main reasons is the fact that vision also provides information about the appearance of the object(s) and the surroundings. This is particularly attractive in the area of personal assistance devices, where LIDAR systems for example are not appropriate due to their size. Nevertheless it is not uncommon to see combinations of traditional and vision based sensors, such as Google's self driving car project.

To complement this trend even further, recent decline in costs of CMOS image sensors made computer vision even more accessible to the world of embedded systems. Cheap camera modules can provide mobile platforms with a constant stream of rich visual information about the environment, at a relatively small cost and scale. This is particularly important in the area of portable electronics. To satisfy the growing need of long lasting and comfortable to carry electronics, portables rely heavily on embedded systems, due to their small factor, low-power usage and low-cost.

Unfortunately object recognition by means of computer vision is not yet optimal and efficient enough, when deployed on a conventional embedded system. The main reasons are that the on-board embedded computers do not possess sufficient hardware resources to ensure proper functioning and smooth execution

of the underlying algorithms. These embedded systems simply cannot guarantee that the needed information arrives on time and is indeed reliable. To make it more clear it is the vast amount of incoming raw data needed to be processed, that makes it challenging for an embedded system to extract useful information in real-time.

Standard personal computers equipped with powerful processors and GPU's can easily handle modern computer vision algorithms. But due to the power and cost restrictions, it is not feasible to install conventional computers on mobile platforms. Instead, a common technique is to perform vision on a remote computer or cluster, and receive control commands from there. But for mobile safety critical applications, the latency introduced is unacceptable.

Nowadays, there has been a huge surge of systems on chip incorporating multiple processing cores and hardware acceleration, dedicated for a specific task. The synergy between software and hardware allows the system to be both flexible and keep up with high throughput constraints. Indeed the intermediate usage of hardware, allows embedded systems to easily achieve throughputs, comparable to a pure software implementation on a high-end PC.

One such system is the *Starburst* MPSoC, actively developed at the chair of Computer Architectures for Embedded Systems (CAES), University of Twente, and the main development platform of this thesis. At the moment, the system is deployed on a Xilinx ML605 development board, featuring a high-capacity Virtex-6 FPGA, allowing plenty of room to study and evaluate embedded computer vision. It consists of multiple ring interconnected processors, which execute a RTOS known as Helix. Multi-processor software can be easily analyzed using real-time tools and eventually deployed on the cores, guaranteeing fair and deterministic behavior. Unfortunately it still lacks the hardware capabilities to handle computer vision effectively.

Therefore the focus of this thesis is to implement and evaluate a state-of-the-art object recognition and a tracking system, both in hardware and software, on the *Starburst* MPSoC. This technical report summarizes and describes the whole design process and decisions involved during the development of the system. In this chapter, the reader is introduced to the research problem. Section one defines the problem in more detail and raises the appropriate research objectives. Section two states the contributions made by this thesis. Finally section three describes the outline of this report.

1.1 Problem definition

Why does computer vision prove to be so challenging for pure software-only embedded systems? The answer becomes obvious when one considers the vast amount of data to be processed and the complicated mathematical operations involved. Modern CMOS image sensors can provide a consistent stream of images with rates of up to 30 FPS, while the resolution can go beyond 720p. A huge amount pixels needs to be processed multiple times by the system, in order to extract useful features. Even then, the result is still hundreds of features that need to be matched, in order to determine whether the object(s) of interest is(are) present in the current scene. To be able to process all of that data, a modern robust object detection algorithm requires a lot of fast memory. Additionally in order to satisfy high throughput constraints, processing cores

need to work at a much higher frequency, resulting in higher power consumption. Although one can opt to incorporate a big amount of low-frequency cores in the system instead, this scheme brings a lot of other problems to the table, such as memory contention. To ensure fair sharing of resources between cores, a system usually assigns time slices to each of them for using a particular resource. This introduces be a big bottleneck, making parallel execution of computer vision algorithms that rely on big amounts of memory very difficult. These and other similar restrictions are very common to embedded devices, which try to maintain a low-power profile and small form factor. A big advantage of software implementations however are their flexibility and reconfigurability.

In contrast, a pure hardware implementation is not limited by the amount of processing available, but the amount of resources. In ASIC terms, this refers to the area on a chip and the chip process. In FPGA terms, hardware resources take form of BRAM, DSP slices and general purpose CLBs. Hardware gives a designer the power to more easily take advantage of inherent parallelism in an algorithm. Thus it is quite common that hardware implementations often exhibit per-pixel execution for each of the steps in an object detection and tracking processing pipeline. This results in drastic reduction of memory usage, as the need to store most of the data is eliminated, and lower power usage since the processing is done at a relatively low frequency. A big disadvantage however of purely hardware based solutions is the limited flexibility and configuration options. This is problematic because of the dynamic nature of most computer vision systems. The ability to change the parameters on the fly is very important in safety applications.

Taking both advantages and disadvantages of hardware and software into consideration, this thesis puts forward the following objectives:

1. Extend the *Starburst* MPSoC with hardware and software, that can handle generic video input from standard CMOS camera sensors.

2. Research and evaluate the state-of-the-art HOG-SVM object detection and Particle Filter tracking algorithms –

   (a) Determine the trade-offs between efficiency and robustness, of different implementations;

   (b) Identify their respective bottlenecks, and how to optimally mitigate their effect on an implementation, both in hardware and software;

   (c) Evaluate an optimal solution in software.

3. Realize a robust, visual object detection and tracking system based on *Starburst*, capable of achieving real-time performance with high throughput –

   (a) Develop an efficient and capable hardware design of the HOG-SVM object detection framework;

   (b) Implement a Particle Filter based object tracker in software, complemented with appropriate hardware accelerators;

   (c) Integrate the detector and tracker subsystems, while maintaining good balance between software and hardware.

4. Analyze and evaluate the system –

(a) Find and reduce any combinatorial paths in the hardware design;

(b) Estimate the resource usage in software and optimize appropriately.

(c) Make use of real-time analysis tools to determine the limits of the system and identify more bottlenecks

5. Test the system in a real-world setup.

The research question put forward is as follows:
Is the Starburst architecture suitable for computer vision algorithms? More precisely is it suitable for modern object detection and tracking algorithms that must satisfy real-time constraints? If not, then what should be incorporated in the architecture to make it suitable?

1.2 Contributions

There are four major contributions of this research. First, this thesis contributes with a unique and highly-configurable hardware implementation of the famous HOG feature person detector[1]. The algorithm has been presented during the 2013 Embedded Vision Summit, generating a lot of interest among computer vision practitioners and embedded system engineers. Its simplicity and effectiveness for the problem of general object class detection, and in particular - people, is the main motivation for its consideration in this thesis.

Second, a parallel particle filter based object tracking framework is designed to run on multiple cores, utilizing the *Helix* RTOS on *Starburst*. Since the PPF is implemented purely in software, it can be easily adapted for any tracking process. Additionally, it takes full advantage of the underlying *Starburst* architecture to achieve maximum potential.

Third, evaluation and analysis results are provided, to determine the temporal and functional performance of each algorithm. From a RTS perspective, this analysis gives insight into the reliability of the total system. From an ES perspective - insight about the efficiency.

Finally, a direct contribution is made for the *Starburst* MPSoC. By extending its hardware capabilities to handle embedded vision applications, this thesis provides means to further study algorithms in the area of computer vision on the MPSoC, and evaluate their real-time performance.

What this thesis does NOT contribute to is new ideas and algorithms in the field of computer vision. The focus is purely on evaluation and implementation of modern feature extraction and tracking techniques, therefore it heavily relies on available and proven research.

1.3 Thesis outline

So far, the first chapter introduced the reader to the topic of this thesis and the associated research questions. This section describes how the report is organized and a description of each chapter.

Chapter two describes the HOG object detection algorithm. It begins first with a brief history of the algorithm and introduces the reader to its contents. The chapter then goes on with a formal definition of the algorithm and each of

its components. It is then concluded with a small analysis and design considerations, which are used further in the actual hardware implementation.

Chapter three deals with object tracking. In particular, it focuses on the particle filter and its effective use in visual object tracking. First, important notions which are part of the PF framework, such as the SMC simulation, are introduced. Afterwards the notion of parallel particle filters is discussed. The focus is then directed on the usage of PFs for the problem of visual object tracking and how image features are used for particle weight calculation. Finally, the chapter is concluded again with a discussion on design considerations and particular bottlenecks associated with the particle filter, which will come in play later on.

Chapter four describes the implementation of the system. First an overview of the whole system is provided, while subsequent sections describe each part. Starting with hardware based feature extraction, to software implementation of tracking.

Chapter five presents the results. First, results related to the HOG detector are presented, such as hardware resources usage, simulation and parameter optimization.

The final chapter concludes the thesis, with a small discussion, future work and final thoughts.

Chapter 2

The HOG-SVM object detector

HOG features have proven on many occasions to be very effective for the task of object description and detection. Especially when combined with a linear SVM classifier. They were first discovered and used by researchers Dalal and Triggs, and described in their prominent paper[1], which has been considered by many as a state-of-the-art work. Subsequently, it has spawned many applications and extenstions, with one of the most prominent examples being the discriminative parts-based object detector by Felzenszwalb et. al.[2, 3]. A comprehensive study on the performance of HOG-SVM based detectors and reasoning behind their success can be found in [4].

HOG stands for "Histograms of Oriented Gradients" and as the name suggests, the descriptor consists of image gradient orientation histograms, extracted from image patches representing the object(s) of interest. Another descriptor based on a similar principle and also sharing a big amount of success in the field of object detection is Lowe's SIFT[5]. The work by Lowe certainly motivated the creation of the HOG descriptor. However it is important to note that SIFT was designed with a different purpose in mind. It is used to describe rotation and scaled invariant regions, local to the object(s) of interest. This usually results into a lot of descriptors extracted offline from an object template, which are subsequently stored in a data-base for matching purposes. It is therefore very suitable for image or video frame matching, but not for object classification. In contrast, HOG features are typically extracted from a dense grid of "cells", as part of a detection window. Hence they describe an object in its entirety and shape, and are not restricted to the appearance of the object. This makes them very suitable and effective for classification using SVMs. One major drawback is their inability to detect objects in different poses, thus prompting the use of multi-class SVMs, as in [3].

This chapter gives an overview of the HOG-SVM object detector, a computational resource analysis and potential bottlenecks one has to look out for, when implementing the algorithm in hardware.

2.1 HOG features

Histograms of Oriented Gradients are high-level features, which make use of the image gradient to describe an image patch. The constructed feature vector is subsequently used to detect objects of interest using a classifier. Feature vectors are also used to train the classifier. A general sequence of image processing steps are undertaken to extract the HOG features, summarized as follows:

1. Compute the image gradient approximations using horizontal and vertical kernels $[-1, 0, 1]$ and $[-1, 0, 1]^T$ respectively.
2. Compute the gradient magnitude and orientation images. This step transforms the gradient from rectangular to polar form.
3. Given a fixed window size, select a location on the gradient magnitude and orientation images, and extract an image patch within the window boundaries.
4. The patch is then divided into cells of equal size, along the width and height of the window. It is recommended to choose window size, such that it is a perfect multiple of the cell size.
5. For each cell, a histogram of some length L is constructed, where each bin represents quantized gradient orientations. Each gradient pixel from the cell casts a vote proportional to the magnitude into the bin where the pixel's orientation belongs.
6. To improve the descriptor's invariance to brightness and intensity fluctuations, cells are grouped into overlapping blocks. For each block, the cells' histograms are concatenated together to form a vector, which is then normalized. There is a wide choice of normalization functions.
7. Once all block vectors are generated, they are all concatenated to form the final descriptor.
8. The descriptor can now be fed into an SVM classifier, to determine whether the object of interest is present or not in the current window. Multiple descriptors are used to train the classifier.
9. Detections across all scales and positions are interpolated, to find the exact position and scale, based on the score of the SVM decision function.

2.1.1 Gradient calculation

It is important to note that contrary to popular belief, applying a Gaussian filter or similar to the image before computing the gradient, is actually not recommended, as found by Dalal and Triggs. They also found that any other kernel used to compute the gradients, like the Sobel filter, also reduces performance. Conveniently, this makes the task of gradient calculation quite easy, especially in hardware. Generally, the gradients are computed from an image's intensity, but it is possible to use all three RGB channels for more discriminative detection. The gradient is computed using the following equation:

$$\nabla I(x, y) = \begin{pmatrix} I_x(x, y) \\ I_y(x, y) \end{pmatrix} \approx \begin{pmatrix} I(x+1, y) - I(x-1, y) \\ I(x, y+1) - I(x, y-1) \end{pmatrix}, \tag{2.1}$$

where $I(x, y)$ is the image intensity at discrete coordinates x and y. The gradient is then converted into polar form with the relations[1]

$$\|\nabla I(x,y)\|_2 = \sqrt{{I_x}^2 + {I_y}^2} \tag{2.2}$$

$$\theta_{\nabla I(x,y)} = \text{atan2}(I_y,\ I_x) \ \in \ (-\pi, \pi] \tag{2.3}$$

2.1.2 Window and cell formation

After the gradient is computed and transformed into polar form, the resulting magnitude and orientation images are "scanned" using a sliding window for detection purposes. For training purposes, specific window patches containing the object of interest are extracted by hand from training images. The window size used for people detection is usually 64×128. The window is then divided into cells of some size. Dalal and Triggs found that the algorithm performs best with cell sizes of 6×6 pixels and 8×8 pixels. Due to the obvious mathematical properties however, a 8×8 cell is used throughout this thesis. The hiearchy of the divisions (including blocks) is demonstrated in Fig. 2.1.

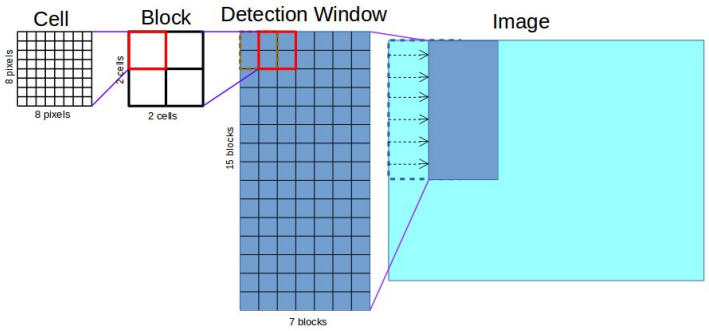


Figure 2.1: Division hierarchy from image to window, to blocks and finally - cells.

Next, gradient orientation histograms can be computed for each cell. The choice of histogram length is important and it depends on the orientation range. It is shown, that 9 and 18 bins is the optimal amount to achieve good detection rate for unsigned and signed orientation respectively. Lower amounts tend to reduce the performance. Furthermore the use of unsigned gradient orientation, i.e. $|\text{atan2}(I_y,\ I_x)| \ \in \ [0, \pi]$, leads to better detection rate. Typically, the bin index is computed by quantizing the histogram orientation, given the histogram length and the orientation range, but this can cause magnitude votes to be casted in the "furthest" bin, even though they are much closer to its left or right neighbor (depending on the quantization method). To reduce this ambiguity, the

[1]The definition of the atan2 function is described in Appendix A.

vote casted by each magnitude pixel can be split between the two neighboring bins, proportionally to the differences of the corresponding orientation and the orientations represented by the bins. Nevertheless the histogram in its simplest form is defined by the equation

$$H(i) = \sum_{x,y} M(x,y,i) \qquad 0 \leq i < L\,;\, i, L \in \mathbb{N} \tag{2.4}$$

where

$$M(x,y,i) = \begin{cases} \|\nabla I(x,y)\|_2, & \text{if } h\left(\text{atan2}(I_y,\, I_x)\right) = i \\ 0, & \text{otherwise} \end{cases}$$

and h is a quantization function, typically represented as

$$h(x) = \left\lfloor \frac{x}{2\pi} + \frac{1}{2} \right\rfloor \times L, \tag{2.5}$$

if the orientation is signed and

$$h(x) = \left\lfloor \frac{x}{\pi} \right\rfloor \times L \tag{2.6}$$

otherwise.

2.1.3 Block formation and histogram normalization

Once histograms are computed, cells are grouped into overlapping blocks, such that each block shares the majority with its neighbors, but excludes at least one row or column of cells. The order of cells doesn't matter, as long as it is the same during training and detection. This process not only introduces redundancy, but also allows the histograms to be normalized around a whole region, reducing the influence of intensity and brightness fluctuations significantly and therefore - improving the detection rate. Dalal and Triggs found block sizes of 2×2 and 3×3 cells to be the most optimal with respect to cell size, with the latter giving the best results. In this thesis, a block size of 2×2 is used.

During the grouping process, histograms of the corresponding cells are concatenated together to form a row vector of features. The vector is then normalized using one of the following relations

$$\text{L2-norm:} \quad \mathbf{x} = \frac{\mathbf{v}}{\sqrt{\|\mathbf{v}\|_2^2 + e^2}} \tag{2.7}$$

$$\text{L1-norm:} \quad \mathbf{x} = \frac{\mathbf{v}}{\|\mathbf{v}\|_1 + e} \tag{2.8}$$

$$\text{L1-sqrt:} \quad \mathbf{x} = \sqrt{\frac{\mathbf{v}}{\|\mathbf{v}\|_1 + e}} \tag{2.9}$$

where $\boldsymbol{v}$ is the block histogram vector and $\boldsymbol{x}$ is the resulting normalized vector. The constant e should be small, but its exact value is not specified.

An additional L2-hyst norm can also be used, which is just the L2-norm with its final value being clipped. Both share equivalent detection results.

2.1.4 Feature vector

Finally, the normalized block histogram vectors are all concatenated to form the feature vector. This vector has a very high dimensionality. For instance, a 64×128 window, divided by 8×8 cells, results into 105 2×2 blocks. If the histogram length per cell $L = 8$, the dimensionality of the feature vector is 3,360. This is perhaps the biggest bottleneck of the algorithm, resulting in a lot of memory usage and complex mathematical operations. Of course for smaller objects(which don't require a big detection window) the dimensionality would reduce, but not substantially.

2.1.5 Classification using linear SVMs

Once the feature vector is acquired by the steps described above, it can be directly fed into a linear SVM classifier. The linear SVM is a binary decision classifier, governed by the score function

$$y(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b \tag{2.10}$$

where $\boldsymbol{x}$ is the input feature vector, and $\boldsymbol{w}$ and b are a vector of weights and a bias constant respectively, estimated during the training process. The decision function used to determine whether a window k contains the object of interest or not is defined as

$$f(\mathbf{x}_k) = \begin{cases} 1, & \text{if } y(\mathbf{x}_k) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.11}$$

2.2 Algorithm analysis and bottlenecks

By now, the reader should be able to understand the steps undertaken to perform object detection, by means of HOG features and a linear SVM classifier. However it hasn't yet been clarified, what is the efficiency of the method, with respect to different implementations and parameters. This section tries to summarize various bottlenecks of the algorithm and the expected resource usage and computational complexity on standard sequential machines. It does so with hopes to give a clear picture of the inner workings of the method and simplify the design process further in this thesis.

2.2.1 Gradient calculation

The gradient component images are computed using kernels $[-1, 0, 1]$ and $[-1, 0, 1]^T$. This is perhaps the least computationally intensive method, involving only two subtractions per pixel. If the convolution is implemented to directly process the image in a raster scan-like fashion, it requires only two memory locations for the horizontal kernel and two line buffers for the vertical. Conveniently, these kernels give the best detection results, as pointed out by Dalal and Triggs.

The obvious bottleneck is the subsequent conversion of the gradient into polar form, involving six (two squares,square root,division and inverse tangent) non-linear operations. It becomes even more severe if floating point arithmetic is used. Typically operations such as the square root function are implemented

using an iterative method, with Newton-Raphson being the most prominent. This method can either be implemented in software or in hardware, as part of a floating-point accelerator, such as the ARM NEON™[6] engine or Intel®'s SSE[7]. Regardless, each of these operations require a lot of instruction cycles per pixel, especially if no floating-point accelerator is present on the underlying architecture. The exact number is largely architecture and method dependent. This bottleneck becomes a big problem on embedded architectures and therefore cannot be remedied, without sacrificing accuracy, such as using fixed-point arithmetic instead of floating point. In some methods, ROM memory is also used for look-up tables, such as linear interpolation. However, this bottleneck affects only the throughput of the algorithm and not its potential real-time performance, as most approximation methods are deterministic.

2.2.2 Window extraction and cell formation

During the detection process, a patch of the image encompassed by a window is extracted at each possible position(defined by the top-left pixel coordinate of the patch), where the window can fit. This act of "sliding" the window and extracting the pixels is followed by forming cells. The whole process involves a lot of memory reads and writes, thus resulting in yet another bottleneck.

Why is it a bottleneck? A typical implementation of the extraction process would result into $(W - W_w + 1) \times (H - H_w + 1)$ windows being extracted, where W and W_w are the image and window widths respectively, and H and H_w - the heights. Then per each window, a total of $W \times H_w$ pixels are accessed from the frame buffers, storing the image gradient magnitude and orientation, for cell formation and histogram binning. This involves a lot of read and write operations and can significantly reduce the through-put of the detection system. Also the amount of memory used is unacceptable for embedded operations. The effect of data transportation is not evident in high-end processing systems, but becomes in embedded systems. Caching and/or sharing the data also introduces non-determinism in the system - highly undesirable for real-time operation. Keep in mind however, that one does not need to store the whole image in the frame buffers, but only enough to hold a window.

There isn't a direct solution to reduce the memory usage and data transportation. The only solution is to exploit how the raw pixel data arrives and compute the histograms as pixels arrive from the input video device, after being processed by the steps earlier. This solution will further be explored in this thesis, when discussing the hardware implementation.

2.2.3 Cell HOG binning

Histogram binning involves accumulation of gradient magnitude pixel values into bins, based on the orientation value at the same pixel locations. Besides the memory bottleneck introduced earlier, there are others during the summation and quantization processes themselves.

First, if a direct method defined by e.g. equations 2.5 and 2.6 is used to compute the bin index, it would introduce an expensive division and multiplication. The only way to avoid these operations then is to use a series of comparisons of the orientation pixel value to a discrete orientation interval table, as is done in e.g [8]. In any case more than one instruction is needed to compute the index.

That is why, the orientation should first be quantized, before feeding each pixel to the orientation frame buffer. This avoids the need to compute the bin index during the binning process.

Second, a total amount of 64 additions are required to fill each bin. If interpolation is used, the computational complexity is even bigger. However keeping track of the histograms, by adding only "newly" arrived pixels and throwing away "old" pixels in a raster-scan fashion can significantly improve bin accumulation in that regard. To elaborate further, suppose that $\boldsymbol{A}$ is a matrix of gradient magnitude pixels representing a cell, as part of the frame buffer. Assuming that pixels arrive along the width of an image, the histogram can be updated by subtracting the values from the last column of $\boldsymbol{A}$ and adding the column of pixels next to the cell from the frame buffer. $\boldsymbol{A}$ is updated accordingly as well, since new data is shifted in the frame buffer. This results into 17 additions and one subtraction per cell - a great improvement! This method is also explored later in this thesis.

Another very powerful approach is to use the so called integral histogram[9], based on integral images. An example, where HOG-like features are extracted using this approach for PF based tracking is described by Yang et al[10]. Instead of performing accumulation "on-the-fly", integral images are first computed for each bin. Computing the histograms for each cell then amounts to two additions and one subtraction per bin/integral image. In total however, the amount of additions stays relatively the same as the previously described method. However it makes histogram extraction quite easy on a sequential machine. The obvious disadvantage is the additional frame buffers required for each bin, added to the fact that this method doesn't map well in hardware.

2.2.4 Block formation and normalization

To form blocks, one has to first compute the cell histograms. Subsequently, histograms are concatenated together to form a block vector and eventually normalized using of the norms, defined by Eq. 2.7,2.8 or 2.9. The act of concatenation itself involves copying the histograms of the cells involved into a new memory locations and then normalizing.

The most prominent computation hurdle is the amount divisions involved during normalization. It can be reduced to one, by first computing the reciprocal of the norm and then multiplying each of the vector components. The amount of multiplications can be also reduced to one, by exploiting the dot product in the SVM score function. This is discussed later. As for the actual norm function, the complexity depends on the design choice. The L2-norm and L1-sqrt are the most computationally expensive, but the miss rate of the detector is significantly reduced. L1-norm increases the miss rate slightly, compared to the other norms, but it is computationally light since it involves only additions. One must thus accept these trade-offs and decide if accuracy is important, compared to algorithmic efficiency.

2.2.5 SVM dot product

The dimensionality of the final feature vector is the biggest bottleneck here. The big amount of multiplications and additions involved in the dot product (as seen in Eq. 2.10), makes this step the most computationally intensive. The

number of multiplications is governed by the following relation, assuming an 8×8 cell size and window size $W_w \times H_w$:

$$M = \left(\frac{H_w}{8} - 1\right) \cdot \left(\frac{W_w}{8} - 1\right) \cdot p^2 \cdot L \quad \mod(H_w, 8) = 0 \ \wedge \ \mod(W_w, 8) = 0 \tag{2.12}$$

where M is the total multiplications amount, p - block width and height in cells, L - histogram length and $\mod(a, b)$ is the modulus function of two integers[1]. The number of additions is the same, including the SVM bias constant.

This step can only be accelerated by splitting the dot product into smaller dot products of block vectors and performing multiplications in parallel. The results of these products are gradually accumulated, until the final block of the window is "filled" in. A simple comparison operation is then needed to determine whether an object is detected or not. Afterwards, the memory location which holds the sum can be reused for a new window. Most of these small dot products can be calculated efficiently on a processor with DSP capabilities, possibly reaching single clock cycle execution. This is the exact approach used by Hahnle et al.[11] and the implementation described by this report.

2.2.6 Summary

The HOG-SVM detector is a very complex and computationally heavy algorithm. Memory storage, data transportation and non-linearity of operations are the most dominant sources of bottlenecks. Yet a lot of its features can be exploited, such as parameter tweaking and the way data is acquired, to increase the efficiency of the algorithm. A lot of the non-linear operations can be approximated and efficiently implemented in hardware to achieve single clock cycle performance, which is the major advantage of hardware vs. software implementations. Most memory locations can be reused for newer data, with the old one safely thrown away.

2.3 Design considerations and trade-offs

This section summarizes some parameter considerations of the algorithm and their associated trade-offs. It provides a small discussion for each step, to explain the reasoning behind these design decisions. Since the associated design choices affected the out-come of the final implementation, it is beneficial in case that reader is interested in the though process behind those decisions.

2.3.1 Cell and block size

In [1], the authors experimented with various cell and block sizes and found that any combination of 8×8 or 6×6 cells with 2×2 or 3×3 blocks give the most optimal results. A cell size of 6 and block size of 3 give the best result. Cells sizes of 6 and 8 share the same performance for block size of 2.

The soon to be discussed hardware implementation however uses a cell size of 8 and block size of 2, since both integers are a power of two. This allows easy multiplication and division by means of simple shifts, which becomes useful

[1] A definition of the function is described in Appendix A.

in counting processes later on. Additionally as it will be seen later, this allows design of "perfect" binary adder trees, which is more than welcome in the design. All of this will come at the cost of slight performance drop.

2.3.2 Choice of histogram length

In their research, Dalal and Triggs found that increasing the length of the gradient orientation histogram increases performance significantly, by minimizing miss rate. However, they also show that using unsigned orientation as opposed to signed, cuts the length by a factor of two, while giving similar results. This means for example, that choosing a length of 18 for the full range of 0°to 360°, gives equivalent results for a length of 9.

In this thesis, a histogram length of 8 is used. Again the main reason is the fact that 8 is a power of two and therefore maps very well in hardware. Combined with a block size of 2, the resulting block vector will have a length of 32. Any multiplications or divisions by 8 or 32 can be substituted with a simple shift operation. Signed orientation is used to accommodate the small length. As seen in the results section in[1], this configuration gives close to optimal performance.

2.3.3 Block normalization

Normalization is a very important step during the HOG extraction process and therefore cannot be skipped. It is shown that any normalization function significantly improves the detection accuracy. However with greater accuracy, comes higher computational cost which translates into more hardware resources when implementing the detector on an FPGA.

This implementation utilizes the L1-norm, because of its simplicity. In hardware, it can be directly implemented by a simple binary adder tree. The only operation that needs to be performed then is a division. As it will be seen, this division done in fixed-point arithmetic using a linear interpolation circuit. After the inverse norm has been computed, it is then passed to the SVM dot-product, to be multiplied. The miss rate will increase, but not by much. The authors of [8] report the using this norm still allows for robust performance, compared to any of the other norms.

Chapter 3

Tracking using Particle Filters

The Particle Filter(PF) is a general name for a class of approximate Bayesian state estimation techniques, which rely on successive Monte Carlo simulations to solve the filtering problem. One of its earliest versions is the Sequential Importance Resampling(SIR) filter, or "the bootstrap algorithm" as introduced by Gordon et al.[12]. As demonstrated in the paper, the Particle Filter proves to be an extremely effective state estimation technique for general non-linear problems, as it greatly outperforms EKF at determining the state of a highly non-linear system through a highly non-linear observation function.

However the accuracy and effectiveness of the filter is mostly dependent on the amount of particles generated by each Monte-Carlo simulation. To be effective, the particle filter requires large number of particles. This number rises exponentially with the dimensionality of the system and is also affected by the non-linearity of the system. Thus, the main drawback of the PF is its computational cost.

This chapter introduces the reader to the notion of Particle Filtering, its use in visual tracking, its bottlenecks and their respective remedies. It begins with a short definition of the filtering problem from a Bayesian perspective – the very foundation of the Particle filter algorithm – and then proceeds with a description of the PF and in particular - the SIR algorithm. The description is rather general, but it is backed with a simple one dimensional example, to help the reader understand the algorithm more intuitively. Next, the use of the particle filter to solve the visual object tracking problem is covered. The focus is shifted primarily on standard motion models of a tracked object, and how standard image processing techniques are used to observe and determine its existence in successive frames. Then the chapter goes on with a discussion about the main bottlenecks encountered in each step of the algorithm, which prevent its fluent execution on a machine with limited computational resources. This is followed with more discussions from the author's point of view about acceleration approaches to reduce the effects of these bottlenecks, and ensure high-throughput operation. The chapter is concluded with a summary of design considerations behind the implementation, developed as a result of this research.

3.1 Algorithm definition

3.1.1 Bayesian state estimation

Given is a general stochastic system, governed by the following discrete-time equations[13, 14]

$$\mathbf{x}_k = \mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{w}_{k-1}) \tag{3.1}$$

$$\mathbf{y}_k = \mathbf{h}_k(\mathbf{x}_k, \mathbf{v}_k) \qquad k \in \mathbb{N}, \tag{3.2}$$

where $\mathbf{x}_k$ and $\mathbf{x}_{k-1}$ are the current and old state vectors of the system at some time stamp t_k and t_{k-1} respectively, while $\mathbf{y}_k$ is a state observation vector. Additionally $\{\mathbf{w}_{k-1}\}_{k\in\mathbb{N}_{>0}}$ and $\{\mathbf{v}_k\}_{k\in\mathbb{N}}$ are independent process and observation noise sequences. $\mathbf{f}_k : \mathbb{R}^{n_x} \times \mathbb{R}^{n_w} \to \mathbb{R}^{n_x}$ is the state-transition function, based on an analytical or purely statistical system model (possibly non-linear), which is used to derive the current state from the old state, where n_x and n_w are the dimensions of the state and process noise vectors, respectively. The state vectors are usually hidden or not directly observable. The only means to observe the current state is through the observation (or measurement) function $\mathbf{h}_k : \mathbb{R}^{n_x} \times \mathbb{R}^{n_v} \to \mathbb{R}^{n_y}$, which transforms the current state vector into the observation vector $\mathbf{y}_k$, where n_y and n_v are the dimensions of the observation and measurement noise vectors, respectively.

The objective of optimal filtering is to derive an estimate $\hat{\mathbf{x}}_k$ of the current state vector, using past and recent observations. From a Bayesian stand point, this amounts to building up sufficient confidence in the state $\mathbf{x}_k$ at time t_k, by recursively deriving the conditional posterior PDF $p(\mathbf{x}_k|\mathbf{y}_{1:k})$, where $\mathbf{y}_{1:k}$ is a set of all observation vectors from time t_1 up until t_k. It is assumed that the initial PDF $p(\mathbf{x}_0|\mathbf{y}_0) = p(\mathbf{x}_0)$ is known, with $\mathbf{y}_0$ being an empty set of measurements. It is then possible to construct $p(\mathbf{x}_k|\mathbf{y}_{1:k})$ recursively in two steps: prediction and update; which is then used to estimate $\hat{\mathbf{x}}_k$.

The prediction step involves solving the Chapman-Kolmogorov equation to obtain the prior PDF

$$p(\mathbf{x}_k|\mathbf{y}_{1:k-1}) = \int p(\mathbf{x}_k|\mathbf{x}_{k-1})p(\mathbf{x}_{k-1}|\mathbf{y}_{1:k-1})d\mathbf{x}_{k-1}, \tag{3.3}$$

where $p(\mathbf{x}_k|\mathbf{x}_{k-1})$ is defined by the system model equation 3.1 and the known statistics of $\mathbf{w}_{k-1}$, assuming that the model is a first order Markov process[14]. The idea of the prediction step is to gain some "trustfully" prior knowledge of the state, based on the system dynamics and control input (if present).

During the update step, the posterior PDF of the state is obtained using Bayes' rule and the newly acquired measurement $\mathbf{y}_k$, such that

$$p(\mathbf{x}_k|\mathbf{y}_{1:k}) = \frac{p(\mathbf{y}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{y}_{1:k-1})}{p(\mathbf{y}_k|\mathbf{y}_{1:k-1})} = \frac{p(\mathbf{y}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{y}_{1:k-1})}{\int p(\mathbf{y}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{y}_{1:k-1})d\mathbf{x}_k}, \tag{3.4}$$

where $p(\mathbf{y}_k|\mathbf{x}_k)$ is the likelihood function, defined by the system observation equation 3.2 and the known statistics of $\mathbf{v}_k$. The likelihood PDF is based on a similarity measure between the most recent observation and the prior, after "propagating" it through the observation model.

The Bayesian filter can thus be summarized in the following steps:

1. Start with an idea of what the initial state might be, by defining the initial PDF $p(\mathbf{x}_0)$.

2. Predict a future state using 3.3 and deriving the prior distribution $p(\mathbf{x}_k|\mathbf{y}_{1:k-1})$.

3. Obtain a new measurement $\mathbf{y}_k$, from a sensor or other sources.

4. Use the prediction and new measurement in the update step, to obtain the posterior distribution $p(\mathbf{x}_k|\mathbf{y}_{1:k})$.

5. Use the posterior to acquire an estimate $\hat{\mathbf{x}}_k$ of the current state.

6. Repeat steps 2 to 5, until system shut down or failure.

3.1.2 Sequential Importance Sampling

The above definition forms the basis for the so-called optimal Bayesian filter solution. In general however, it cannot be obtained analytically for non-linear and/or non-Gaussian noise systems. The only analytical solution that exists, is the so called Kalman filter and is restricted to linear, Gaussian noise systems. For the non-linear and/or non-Gaussian case, approximate methods are used such as PFs.

Particle filtering comes in a lot of flavors, but in its most general form, it is known as the Sequential Importance Sampling algorithm. It is a Monte Carlo (MC) based method, which implements the Bayesian filter by approximating the prior and posterior density functions, with a set of discrete samples. An estimate of the state and/or its variance is subsequently obtained from the approximated posterior. As the number points increases, the SIS filter approaches the optimal Bayesian filter. In addition, a resampling step is introduced at the end of each PF iteration, resulting in the so-called Sequential Importance Resampling algorithm.

Without going in too much detail, a general iteration of the algorithm begins by generating a set of random samples, $\{\mathbf{x}_i\}_{i=1,...,N}$, from a proposal importance density distribution $q(\mathbf{x}_k|\mathbf{y}_k)$, such that $\mathbf{x}_i \sim q(\mathbf{x}_k|\mathbf{y}_k)$. Here N denotes the total number of samples used by the algorithm, while the $\sim$ symbol indicates the probabilities of the samples on the left side, are proportional to the PDF on the right side. Then these samples are weighted, based on the likelihood PDF, $p(\mathbf{y}_k|\mathbf{x}_k)$, and the importance sampling principle. Together these samples and weights form a set of tuples, called "particles". The importance density, $q(\mathbf{x}_k|\mathbf{y}_k)$, is chosen, so that it has the same support as the posterior distribution $p(\mathbf{x}_k|\mathbf{y}_k)$. There is a wide choice of candidate distributions, but most commonly it is selected to be the distribution $p(\mathbf{x}_k|\mathbf{x}_{k-1})$, because of its simplicity and ease of implementation. This choice is also adopted throughout this thesis, and the particular implementation as a result of this choice is detailed below.

Just like the analytical Bayesian filter, the PF has its own prediction and update steps. The prediction step involves the relations

$$\begin{aligned} &\mathbf{x}_{0,i} \sim p(\mathbf{x}_0) \\ &\mathbf{x}_{k,i} = \mathbf{f}_{k-1}(\mathbf{x}_{k-1,i}, \mathbf{w}_{k-1}) \quad i = 1, ..., N \end{aligned} \tag{3.5}$$

where $\mathbf{w}_{k-1}$ is randomly generated noise sample from the process PDF, and $\{\mathbf{x}_{k,i}\}_{i=1,...,N}$ are proposal samples of the future state. The procedure is similar to the Bayesian filter. The main difference is that instead of analytically deriving the prior state PDF (by evaluating the integral in Eq. 3.3), it is iteratively approximated, by randomly sampling the process noise PDF and using the state-transition relations. Hence why generating more random samples increases the similarity between the analytical solution and the discrete version.

Once a set of sample predictions are generated, each is assigned a weight $w_{k,i}$ during the update step, such that

$$\begin{aligned} w_{0,i} &= \frac{1}{N} \quad i = 1, ..., N \\ w_{k,i} &= w_{k-1,i}\ p(\mathbf{y}_k|\mathbf{x}_{k,i}). \end{aligned} \tag{3.6}$$

Similarly to the Bayesian case, the update step of the PF makes use of a newly acquired measurement, $\mathbf{y}_k$, to calculate the likelihoods of the prediction samples generated earlier, through the likelihood PDF, $p(\mathbf{y}_k|\mathbf{x}_{k,i})_{i=1,...,N}$. This is done by propagating the samples through the observation model defined in Eq. 3.2 (without the noise factor), using a distance measure to calculate the measurement mismatch error, and finally calculating the likelihoods using the PDF of the measurement noise. If the measurement noise, $\mathbf{v}_k$, is additive and its PDF is Gaussian with zero mean and covariance matrix $\mathbf{\Sigma}$, then 3.6 can be simplified to

$$w_{k,i} = w_{k-1,i}\ \frac{1}{\sqrt{2\pi|\mathbf{\Sigma}|}} \exp\left(-\|\mathbf{y}_k - \mathbf{h}_k(\mathbf{x}_{k,i}, 0)\|^2_\Sigma\right) \quad i = 1, ..., N, \tag{3.7}$$

[15, Chapter 15], where $\|\mathbf{v}\|^2_A = \mathbf{v}^\top \mathbf{A}\mathbf{v}$ denotes a weighted norm of some vector $\mathbf{v}$, and $|\mathbf{A}|$ – the determinant of a matrix $\mathbf{A}$.

Note that the above-defined relations describe the special case, when the proposal importance density function, $q(\mathbf{x}_k|\mathbf{y}_k)$, is chosen to be the state PDF $p(\mathbf{x}_k|\mathbf{x}_{k-1})$. For a definition of the more general case and a derivation for this particular case, please refer to [14].

The calculated tuples $\{\mathbf{x}_{k,i}, w_{k,i}\}_{i=1,...,N}$ form the resulting particle set, which can now be used to approximate the posterior PDF using a sequence of delta functions, such that

$$p(\mathbf{x}_k|\mathbf{y}_k) \approx \hat{p}(\mathbf{x}_k|\mathbf{y}_k) = A_k^{-1} \sum_{i=1}^{N} w_{k,i}\delta(\mathbf{x}_k - \mathbf{x}_{k,i}) \tag{3.8}$$

where $A_k = \sum_{i=1}^{N} w_k^i$ is a weight normalization constant. An estimate, $\hat{\mathbf{x}}_k$, can be calculated using

$$E(\mathbf{x}_k|\mathbf{y}_k) \approx \hat{\mathbf{x}}_k = A_k^{-1} \sum_{i=1}^{N} w_{k,i}\mathbf{x}_{k,i}, \tag{3.9}$$

which is just a weighted average of the discrete posterior; or using

$$\hat{\mathbf{x}}_k = \mathbf{x}_{k,j} \qquad j = \arg\max_i\ w_{k,i}, \tag{3.10}$$

which amounts to just picking the sample with highest weight.

The overall operation of the SIS Particle filter is compactly described in Alg. 1. This description assumes that the algorithm is executed on a sequential machine, which periodically acquires sensor measurements (as part of some sort of control system or filtering process), such as GPS location, ADC voltage measurements, radar or laser scanner range measurements, etc.

Algorithm 1 SIS Particle Filter Algorithm

for $i = 1 : N$ **do**
    Initialize $\{\mathbf{x}_{0,i}, w_{0,i}\}$, such that $\mathbf{x}_{0,i} \sim p(\mathbf{x}_0)$ and $w_{0,i} = \frac{1}{N}$.
end for
for each system iteration $k \in \mathbb{N}_{>0}$ **do**
    Acquire new measurement $\mathbf{y}_k$ from sensors (GPS, Camera, ADC, etc..)
    for $i = 1 : N$ **do**
        Draw a sample $\mathbf{x}_{k,i} \sim p(\mathbf{x}_k|\mathbf{x}_{k-1})$ using Eq. 3.5
        Assign a particle weight, $w_{k,i}$, using Eq. 3.7
    end for
    Approximate $\hat{\mathbf{x}}_k$ using 3.9 or 3.10
end for

3.1.3 The Resampling Step

The SIS approach suffers from a fundamental issue, where the weights of all but one particle become negligible after a few iterations. This is the so-called *sample degeneracy* problem, and it proves to be very problematic for the operation of the PF. There are ways to counteract this issue, such as choosing a very large amount of particles, N, or a better importance density PDF, $q(\mathbf{x}_k|\mathbf{y}_k)$, but the most common is the introduction of the Resampling step at the end of each PF iteration.

During this step, particles are resampled from the discrete approximated posterior distribution, $\hat{p}(\mathbf{x}_k|\mathbf{y}_k)$, so that their respective weights are set back to $w_{k,i} = \frac{1}{N}, i = 1, ..., N$, and the state samples, $\{\mathbf{x}_{k,i}\}_{i=1,...,N}$, are "relocated" to state space regions with high-valued weights. The new set of particles now represents the approximated posterior as a discrete uniform distribution. There are many genetic algorithms which accomplish this task, but the most common *naive* method is described in Alg. 2 (also known as the Roulette Wheel Sampling algorithm). Note that the weights are assumed to be normalized before resampling, such that $\sum_{i=1}^{N} w_{k,i} = 1$.

Since at the end of each iteration (and initially) all of the weights have a value N^{-1}, Eq. 3.7 simplifies to

$$w_{k,i} = \frac{1}{N\sqrt{2\pi|\mathbf{\Sigma}|}} \exp\left(-\|\mathbf{y}_k - \mathbf{h}_k(\mathbf{x}_{k,i}, 0)\|_{\Sigma}^2\right) \quad i = 1, ..., N, \tag{3.11}$$

while Eq. 3.9 to

$$\hat{\mathbf{x}}_k = N^{-1} \sum_{i=1}^{N} \mathbf{x}_{k,i}. \tag{3.12}$$

The modified SIS algorithm, by including the resampling step, is called Sequential Importance Resampling algorithm, and is described in Alg. 3.

Algorithm 2 Roulette Wheel Resampling

function $\{\mathbf{x}^*_{k,i}, w^*_{k,i}\}_{i=1,...,N} =$ Resample$(\{\mathbf{x}_{k,i}, w_{k,i}\}_{i=1,...,N})$
Input : Old particle set of the current iteration
Output : New set of particles

Initialize cumulative sum of weights: $c_1 = w_{k,1}$
for $j = 2 : N$ **do**
    Build the rest of the sum: $c_j = c_{j-1} + w_{k,j}$
end for
for $i = 1 : N$ **do**
    Generate a uniformly distributed random number: $u \sim \mathbb{U}[0, 1]$
    Initialize index: $j = 1$
    while $c_j < u$ **do**
        $j = j + 1$
    end while
    Assign new particle: $\{\mathbf{x}^*_{k,i}, w^*_{k,i}\} = \{\mathbf{x}_{k,j}, \frac{1}{N}\}$
end for

Algorithm 3 SIR Particle Filter Algorithm

for $i = 1 : N$ **do**
    Initialize $\{\mathbf{x}_{0,i}, w_{0,i}\}$, such that $\mathbf{x}^i_0 \sim p(\mathbf{x}_0)$ and $w_{0,i} = \frac{1}{N}$.
end for
for each system iteration $k > 0$; $k \in \mathbb{N}$ **do**
    Acquire new measurement $\mathbf{y}_k$ from sensors (GPS, Camera, ADC, etc..)
    for $i = 1 : N$ **do**
        Draw a sample $\mathbf{x}_{k,i} \sim p(\mathbf{x}_k|\mathbf{x}_{k-1})$ using Eq. 3.5
        Assign a particle weight, $w_{k,i}$, using Eq. 3.11
    end for
    Resample particles: $\{\mathbf{x}_{k,i}, w_{k,i}\} =$ Resample$(\{\mathbf{x}_{k,i}, w_{k,i}\})$ $\quad i = 1, ..., N$
    Approximate $\hat{\mathbf{x}}_k$ using 3.12
end for

Resampling is a typical genetic approach, where parent particles produce a new "evolved" off-spring. It thus increases the chances of particles with high-valued weights to grow even bigger and produce more accurate results. But it also introduces a problem, where particles with high weights are statistically selected multiple times, leading to a loss of diversity [14]. This issue is known as *sample impoverishment* and is particularly dominant, when the process noise has small variance. Therefore particles are usually "roughened [15, 12], by either increasing the variance of the system model's process noise, or just adding artificial noise to the resampled particles. More roughening methods are discussed in [16].

3.1.4 SIR example

To demonstrate the operation of the Particle filter and help its understanding, an example is presented of a non-linear system and a non-linear sensor. The modeled system is an object falling through the atmosphere of Earth. This system is quite common, but the particular example used to demonstrate the

PF, is a modified version of the one provided in [17, 15].

The state of the system consists of three variables – the altitude of the object, its velocity and constant ballistic coefficient. The continuous state-space equations of the system are defined as

$$\begin{aligned} \dot{x_1}(t) &= x_2(t) + \mathrm{w}_1 \\ \dot{x_2}(t) &= \frac{\rho_0}{2} \exp\left(-\frac{x_1(t)}{\alpha}\right) x_2^2(t) x_3(t) - g + \mathrm{w}_2 \\ \dot{x_3}(t) &= \mathrm{w}_3, \end{aligned} \tag{3.13}$$

where $g \approx 9.832\,[m/s^2]$ is the gravitational acceleration constant, ρ_0 – the air density at sea level and α defines the relationship between air density and altitude. $\mathbf{w} = (\mathrm{w}_1\ \mathrm{w}_2\ \mathrm{w}_3)^\top$ is white process noise with zero mean and covariance matrix $\boldsymbol{S}$, such that $\mathbf{w} \sim \mathcal{N}(0, \boldsymbol{S})$.

The object is being observed from a sensor, which measures the range y between the falling object and the device. This can be a radar system, a "smart" camera sensor, a laser range finder, or others. The sensor is located at an altitude a, while the distance between it and the object's vertical line of fall is M. Measurements are taken periodically at discrete time stamps t_k, such that

$$\begin{aligned} y_k &= h_k(\mathbf{x}_k, v_k) \\ &= \sqrt{M^2 + (x_1(t_k) - a)^2} + v_k, \end{aligned} \tag{3.14}$$

where v_k is measurement white noise with zero mean and variance R, such that $v_k \sim \mathcal{N}(0, R)$. This setup is illustrated in Fig. 3.1.

Since the state-space equations are continuous, a classical 4th order Runge–Kutta method is used to derive a discrete state-transition model. Let T_s be the time between successive measurements, such that $t_k = kT_s,\ k \in \mathbb{N}$, while the step-size $\tau = T_s/L$, with L being the number of Runge–Kutta iterations per measurement. If $\dot{\mathbf{x}}(t) = (\dot{x}_1\ \dot{x}_2\ \dot{x}_3)^\top = \mathbf{f}(t, \mathbf{x})$ as defined by Eq. 3.13, then a Runge–Kutta iteration is

$$\begin{aligned} &\boldsymbol{z}_0(k) = \mathbf{x}_k \\ &\boldsymbol{z}_i(k) = \boldsymbol{z}_{i-1}(k) + \frac{\tau}{6}\, \boldsymbol{g}\left(t_k + (i-1)\tau, \boldsymbol{z}_{i-1}(k)\right) + \mathbf{w}_i \tau; \quad 1 \le i \le L \in \mathbb{N}_{>0}, \end{aligned}$$

with $\mathbf{w}_i \sim \mathcal{N}(0, \tau \boldsymbol{S})$ and

$$\begin{aligned} \boldsymbol{g}(t, \mathbf{x}) &= \boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4 \\ \boldsymbol{k}_1 &= \mathbf{f}(t, \mathbf{x}) \\ \boldsymbol{k}_2 &= \mathbf{f}(t + \frac{\tau}{2}, \mathbf{x} + \frac{\tau}{2}\boldsymbol{k}_1) \\ \boldsymbol{k}_2 &= \mathbf{f}(t + \frac{\tau}{2}, \mathbf{x} + \frac{\tau}{2}\boldsymbol{k}_2) \\ \boldsymbol{k}_2 &= \mathbf{f}(t + \tau, \mathbf{x} + \tau\boldsymbol{k}_3). \end{aligned}$$

Now one can define the discrete state-transition equation as

$$\begin{aligned} \mathbf{x}_k &= \mathbf{f}_{k-1}(\mathbf{x}_{k-1}, \mathbf{w}_{k-1}) \\ &= \boldsymbol{z}_L(k-1) \quad k \in \mathbb{N}_{>0}. \end{aligned} \tag{3.15}$$

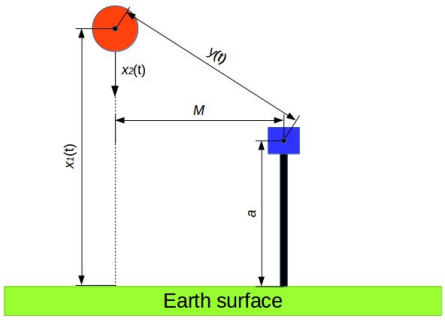
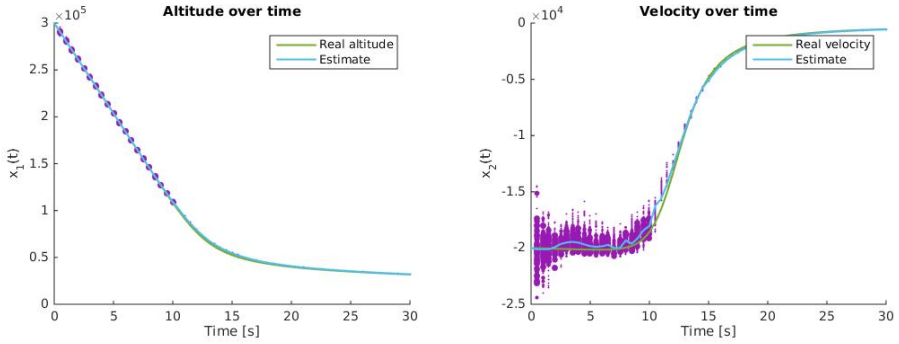
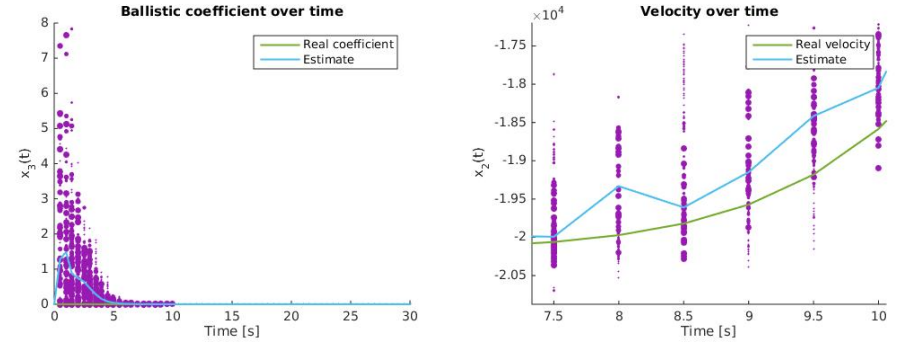


Figure 3.1: Sketch of the system; the blue box represents the distance sensor, while the orange sphere represents the falling object



(a) Altitude (b) Velocity



(c) Ballistic coefficient (d) Velocity (zoomed in)

Figure 3.2: System simulation plots; the purple dots represent the particles set for each state variable before resampling, with their respective sizes proportional to the weights

Using the process and observation models, one can now track the state of the falling object using Alg. 3. The parameters of the system and the PF are summarized in Table 3.1, while plots of the simulated and estimated system state can be seen in figures 3.2a–3.2c. Figure 3.2d shows a zoomed-in region

Parameter	Value
N	100
L	12
T_s	0.5 s
α	6096 [m]
ρ_0	0.411492076 $[kg^2/m^4]$
a	30,480 [m]
M	30,480 [m]
$\mathbf{x}_0$	$[91440[m] \quad -6096[m^2/s] \quad 3.048 \cdot 10^{-4}[kg/m^2]]^\top$

Table 3.1: Parameters values for falling body PF tracking example

on the velocity plot, where the particles per iteration are more clearly visible. It is worthwhile noticing, that the particles further from the real state have much smaller weights, while the closer ones have bigger weights. One can also see that the estimated velocity follows a trajectory, close to the particles with high weights. This is because after resampling, the new set of particles for the current iteration are more densely concentrated around these regions.

3.2 Visual Tracking

Visual tracking is a specific case of the filtering problem, where the location of a target object is estimated in each successive frame of a periodic video sequence. In this sense, system observations are usually acquired in the form of raw, discrete images, which potentially contain the object. The objective is to filter out all of the background "clutter" and **other** objects, by using past estimations and specific features that represent the appearance of the target. Optionally, one can also estimate other state variables, such as scale and pose. In fact, it is possible to estimate an object's affine transformation, as is done in [18]. Although one may argue, that the same effect is achieved via a detection algorithm, there is one subtle difference. A detector by itself, doesn't necessarily guarantee that a recognized object, similar to the target, is indeed the same object. By relying on the state history, a tracking algorithm has the potential to "lock" on the target, therefore increasing the probability of locating it again in future frames, provided that it is still observable. Even if the object of interest is lost, due to occlusions or other interferences, a tracking algorithm still "remembers" its past state, preserving the potential of tracking. This gives the system an ability to record an object's trajectory, and make decisions based on how it evolves over time. The situation becomes even more interesting, when multiple targets are tracked simultaneously.

However, since any tracking algorithm relies on the basic principles of system state estimation, it must be properly initialized with a good idea of the starting location of the object in the very first video frame, before the tracking process can begin. Otherwise it may take a long time, until the system locks onto the desired target (assuming it even does so). In a static setup, where for example the camera sensor doesn't move and the object is always present in the video, the initial location of the target can be easily "guessed" at system startup, by manually selecting a particular region of the image. On the other hand, in

a setup where the targets are dynamically "cherry picked" from the incoming frames, based on their class and/or desired visual signatures, an object detection and recognition algorithm is used to initialize the tracker for each new target. The outputs of the detector can also be used as an extra prediction of the state, to further improve the estimation process, provided that they do not represent a new target [19]. This becomes particularly useful, if only one object is being tracked.

From a PF perspective, the particles represent 2-D locations on the frame image, which potentially belong to the object of interest, as well as additional state parameters, such as scale, rotation, etc. In the context of visual tracking, these "predicted" locations are generated using an object motion model, which is just a system model based on the motion behavior of the object. However, their likelihoods are not calculated in the conventional way defined by e.g. Eq. 3.11, using a generalized observation model. Such a model that incorporates the complete process of transforming observed light reflected from the object into an image, as well as the quantization and distortion effects of the camera sensor, would be too difficult and computationally intensive to implement. Instead, a more simple, approximate method is to sample regions from the current frame image, centered around the particles' locations. These extracted image "patches" are then compared to a template (which contains the target's visual signatures), using some sort of a matching technique. The score of the matching process is then used as a distance measure, which is "fed" in the function defined by e.g. Eq. 3.11 to compute the weights. The boundaries of the regions encompassing the patches, depend on the expected scale of the object, which can be static (plus some variance) if it is not incorporated in the state space, or dynamic otherwise. Their contours may also have different shapes, but most commonly a rectangular shape is assumed. A visual representation of this process can be seen in Fig. 3.3.

In the text to follow, the motion model is discussed in detail, as well as some visual observation models, encountered in modern Particle filter tracking systems. However the reader is encouraged to check [20] for a more detailed overview of various techniques for visual PF based tracking.

3.2.1 Motion Model

As mentioned earlier, the motion model is a specific formulation of the system model (as defined in a generic system state estimation framework), where the state-space of a tracked object incorporates its position in space, its velocity, acceleration and possibly other variables related to the object's appearance. Here, the state-transition equations typically describe the dynamical behavior of the position, as it evolves over time.

There are generally two cases of an object's motion behavior: a) when the motion is well defined and b) – when it is stochastic. The first case applies to problems, where the object(s) being tracked follow a deterministic pattern. In this case, it is possible to define the state-transition equations, based on the principles of classical mechanics. An example of this case is vehicle tracking, where cars usually exhibit approximate linear motion on roads, or angular motion when making a turn. Long-range ballistics is another example, where the motion of a projectile is well defined.

Yet objects that don't have a well defined motion (such as humans), exhibit

a dynamic behavior of the position, governed by a stochastic process. Therefore, the models used to describe the motion of "randomly" moving objects, are statistical, such as Hidden Markov Models (HMMs), Bayesian networks, Auto-Regressive Moving Average (ARMA), etc. Since one of the objectives put forward by this research, is to implement a generic object tracking system, the focus of this thesis will be primarily on such models, and in particular – the ARMA process, because of its simplicity in implementation.

The most commonly used and easy to implement stochastic motion model (also the model assumed throughout this thesis), is a special case of the ARMA process, or namely a p-order linear AR system model, defined by the relation

$$\mathbf{x}_k = \mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{w}_{k-1}) = \tag{3.16}$$

$$= \sum_{i=1}^{p} \boldsymbol{A}_i \mathbf{x}_{k-i} + \mathbf{w}_{k-1} = \tag{3.17}$$

$$= \mathbf{x}'_k + \mathbf{w}_{k-1} \tag{3.18}$$

where $\{\mathbf{w}_{k-1}\}_{k\in\mathbb{N}_{>0}}$ is an artificially generated, independent noise sequence, $\boldsymbol{A}_{i=1,\dots,p}$ are coefficient matrices and $\mathbf{x}_k$ is an object's state at time t_k. The state variable is usually defined as the discrete 2-D position in an image, such that $\mathbf{x}_k = \begin{pmatrix} x_k & y_k & \Delta x_k & \Delta y_k \end{pmatrix}^\top$, where x_k and y_k are discrete coordinates on an image, and Δx_k and Δy_k – discrete velocities. If the noise is zero-mean Gaussian with a co-variance matrix $\boldsymbol{\Sigma}$, then Eq. 3.16 can be also formulated as

$$\mathbf{x}_k \sim \mathcal{N}(\mathbf{x}'_k, \boldsymbol{\Sigma}) . \tag{3.19}$$

which amounts to generating random samples from normal distributions, centered around the sum of "old" states. As seen in the relation above, the AR process is quite simple and straightforward to implement, but it has its limitations in terms of accuracy.

First, because it is purely statistical, it relies a lot on the random number generation procedure. It is therefore important that the Random Number Generator (RNG) is statistically diverse and non-biased. Fortunately, there are a lot of statistically diverse pseudo-random number generators, such as the recently developed PCG[21].

Second, the accuracy and computational complexity of the model depends largely on the amount of particles. If the variance of an object's movement is high, then the amount of particles needed to produce accurate results increases significantly. On the other hand, if the variance is small, using too much particles can result into repeated generation of the same samples and a waste of computational resources. Since the variance is largely dependent on the velocity and acceleration of the target, there isn't a specific amount of particles suitable for all situations.

A third factor affecting the accuracy, is the right choice of coefficient and covariance matrices. Here, the situation is similar to the previous case, and there isn't a "best" choice of these parameters. Some implementations try to learn the optimal values of these matrices from training sequences off-line, and thus greatly improve the accuracy in that regard. However, the best option is to adaptively estimate the coefficients, as is done for example in [18]. Nevertheless, if the target(s) to be tracked are not expected to behave too sporadicly, then the issues discussed so far are not very severe and the AR model is sufficient for the tracking process.

3.2.2 Measurement Model

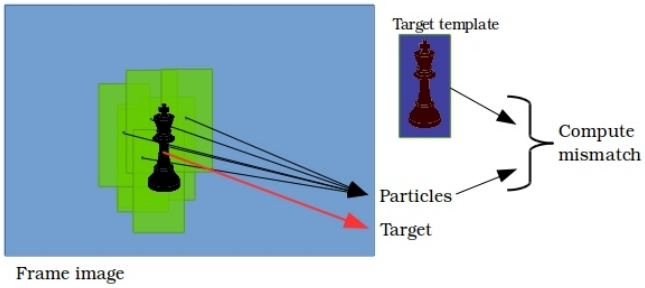


Figure 3.3: Graphical illustration of the particle weighting process for the Visual PF

The measurement model of a typical visual PF tracker, is not directly used in the form of a observation function, as defined earlier in the beginning of this chapter. One can recall, that the purpose of this measurement model is to eventually compute the likelihoods for each of the generated state samples during the prediction step, thus forming the particle set of the current iteration. However, as noted earlier, incorporating a complete visual measurement model is nearly impossible, so the general method to compute the likelihoods, is to extract regions of pixels from the current frame, centered around the discrete locations of the particles on the image, as seen in Fig. 3.3. Then these "patch" images are matched with an object signature template, which contains all relevant information about the appearance of the tracked target. The matching scores, which represent the similarity between the object and the patches, are then used to compute the likelihoods for each particle. The means of computing these scores, is usually through the process of feature extraction and matching, a common procedure in the field of computer vision and image processing. Features like HOG, Haar wavelet responses, edges, pixels' color, etc, are extracted from the sample patches and compared to the template, using some sort of a distance measure. The template is also generated in a similar way, by extracting the same features off-line from an image that fully encompasses the interest object in a still pose. If the pixels themselves are used as signatures during the matching process, then the template represents an image, capturing the target's appearance.

A very common technique to calculate the matching scores of each patch, is by using cross-correlation, as is done in e.g. [18, 22]. Matching using cross-correlation can lead to very good results, even if the number of particles is not very big. As a trade-off, it is a very computationally "heavy" approach, because of the amount of non-linear operations involved per pixel, for each particle. A standard normalized cross-correlation filter is defined by

$$\gamma(f_i, T) = M^{-1} \frac{\sum_{x,y} \left(f_i(x,y) - \bar{f}_i\right)\left(T(x,y) - \bar{T}\right)}{\sqrt{\sum_{x,y} \left(f_i(x,y) - \bar{f}_i\right)^2 \left(T(x,y) - \bar{T}\right)^2}}; \quad 0 < i \leq N, \tag{3.20}$$

where N is the number of image patches (or particles equivalently), $\gamma(f_i, T)$ is

the similarity score, $f_i(x, y)$ and $T(x, y)$ are patch and template images, respectively, with fixed amount of pixels M, and $\bar{f}_i$ and $\bar{T}$ – their means, respectively [23]. If the template and patch images, minus their respective means, are "flattened" into vectors, such that the order of pixels is preserved, then the score function can be represented as a normalized dot-product of these vectors. The weights of each particle are ultimately computed using e.g. Eq. 3.11, by substituting the vector distance with the score function.

Most commonly, the pixels represent the intensity of the current frame, either in one or all three RGB channels. The image frame can also be preprocessed, prior to extracting the image patches, by means of e.g. edge detection and/or normalized distance transform [22]. Such preprocessing allows an object to be tracked, based on its shape and structure, therefore reducing the influence of contrast and illumination changes in the image. One must keep in mind, that the template image needs to undergo the same process, prior to being matched with a patch.

Cross-correlation matching has an additional flaw, besides its computational cost – the template usually represents the object in one pose. So for example, if the object is indeed present in an image patch, but its rotation with respect to the center is different, then the matching score might be quite low, especially if the shape of the object is uneven. Thus, the template is usually "transformed" and matched several times per particle, leading to extremely high computing overhead. There exist powerful approaches, which simplify and accelerate this process, such as the one described in [24], but they still prove too computationally intensive to use, as part of a PF based tracking framework.

Another more common method of matching is by means of histograms, whose aim is to reduce the dimensionality of the template significantly. Also, since a histogram represents an image in terms of the occurance of similar pixel values, no structural or spatial information is stored or used in the matching process. Although patches might undergo certain geometric transformations, if the state-space incorporates scale and/or rotation, no explicit use of these transformations is needed. This method has therefore, a significant advantage over direct pixel matching. Usually, histograms can quantify various types of features in an image, such as colors, intensity, gradients (HOG features), etc, in a similar, compact format, making them a very versatile tool. Matching scores of different types of histograms, can be combined to achieve more robust tracking.

A typical way of matching histograms, is by treating them as vectors and using one of the many distance functions available to compute the mismatch score. However, a common metric used in Particle filtering is the Bhattacharyya distance, defined as

$$D(\mathbf{H}_T, \mathbf{H}_{P_i}) = \sqrt{1 - \sqrt{\mathbf{H}_T^\top} \sqrt{\mathbf{H}_{P_i}}} \quad ; \quad 0 < i \leq N, \tag{3.21}$$

where $\mathbf{H}_T^\top$ and $\mathbf{H}_{P_i}$, are the template and image patch histograms, treated as row and column vectors of length L, respectively.

Although generally much easier to construct and match, a big problem with histograms is that they oversimplify the representation of an object, in terms of its features (especially if their length is small). This often results in image patch histograms, with similar modalities and skewness as the template histogram, even though the "objects" underneath the associated patch regions of

the current frame have nothing in common with the target. Thus, the chances of loosing track of the target increases significantly, because of false mismatches from background clutter. To reduce the effects of this problem, one can incorporate several types of histograms, based on different features. This essentially introduces more signatures, to discriminate background clutter from the template.

To further decrease the chances of mismatching, one can also incorporate some spatial information. An easy way to accomplish this, is by dividing each image patch and the template image into regions. Histograms are then constructed and matched for each corresponding local patch. The size of these local regions must not be "small", compared to the global patch, otherwise the same issues associated with cross-correlation based matching will start surfacing up. The "local" histograms can be independently matched, or after being concatenated together to form a vector (similarly to the HOG descriptor). A good example of the above mentioned approaches is described in [10], where the authors use Haar-like features in combination with gradient orientation histograms.

3.3 Computational complexity and bottlenecks

The PF algorithm is a very powerful, robust solution to the generic non-linear filtering problem, demonstrating exceptional tracking performance even for the more challenging systems. Yet its biggest drawback is its computational cost, preventing it from being largely deployed on embedded systems and high-end computers alike.

One of the main reasons, is the number of particles. A consequence of incorporating one or more state variables in the state-space, is that the amount of particles needed to support a statistically meaningful and accurate estimation, rises exponentially with the dimensionality of the state vector. Thus, the negative effect of various bottlenecks on the computation time of a typical PF iteration, also grows exponentially. This phenomenon is known as the "curse of dimensionality", and is of particular significance in any PF implementation. Therefore, for systems with a large state space, dimensionality reduction techniques are usually applied, such as Principle Component Analysis (PCA), or combinations with other filters, such as the EKF.

The other reason is the nature of the numerical operations involved in each step per iteration, ranging from costly non-linear mathematical functions, to elaborated sampling algorithms. All of these operations pose as a computational bottleneck to the flow of the algorithm, because of their complexity, and limit its throughput significantly on embedded architectures. This results in limited usability of the filter in safety critical systems, which demand fast tracking that can keep up with the throughput of modern sensors.

Here, the bottlenecks associated with each processing step of the Particle Filter are identified and discussed, with hopes to provide background information to the reader about their severity and effects on the overall execution time. Understanding these bottlenecks and the associated trade-offs between algorithmic complexity and estimation accuracy, was of great help in making the appropriate design decisions during implementation. The focus is on the most common bottlenecks, encountered in various implementations of the SIR

algorithm, with additional attention given to the visual object tracking case. Also, it is assumed that the PF is executed on a sequential, single processor machine.

3.3.1 Prediction step analysis

The prediction step, as described earlier in this chapter, is the first part of the SIR algorithm, where state prediction samples are generated from the PDF, $p(\mathbf{x}_k|\mathbf{x}_{k-1})$, as defined by the system (or motion) model, forming the first "half" of the particle set (with weights forming the other). Regardless of the model and the associated mathematical operations, every state prediction involves a RNG operation. If the process noise is modeled as Gaussian (as is often the case), samples are generated from the normal distribution, resulting in several non-linear operations executed per particle. As an example, the off-the-shelf C implementation of the commonly used Marsaglia method [25] described in Alg. 4, to sample from a normal distribution, involves an nondeterministic amount of uniform pseudo-random number generations and multiplications, a logarithm, a division and a square root. Additionally, uniformly generated random numbers are of integer type, and must be normalized to the range $[0, 1]$, by means of divisions (or multiplications if the normalization factor is pre-calculated). In case of sampling from multivariate normal distributions, the complexity increases with the dimensionality of the noise sequence. Another implementation of the normally distributed RNG, is the Box-Muller transform, which involves trigonometric operations. Other approaches exist, which involve a reduced amount of non-linear operations (such as the Ziggurat method developed by Marsaglia et al. [26]), but they still require a couple to effectively sample from the whole distribution. In case of sampling from both sides of the normal distribution, an expensive logarithmic operation is guaranteed to be executed. In the end, there is always the option of incorporating uniformly distributed process noise into the model, instead of Gaussian, at the cost of reduced accuracy.

Algorithm 4 Marsaglia's Method for 1-D Normal Distribution Sampling

function $x =$ randn(μ, σ^2)
Input : Mean and variance of the normal distribution
Output : A normally distributed random sample

1: **do**
2: Draw uniformly distributed variate $x_1 \sim \mathbb{U}[0, 1]$
3: Draw uniformly distributed variate $x_2 \sim \mathbb{U}[0, 1]$
4: Set $w = (2x_1 - 1)^2 + (2x_2 - 1)^2$
5: **while** $w \geq 1 \vee w = 0$
6: $x = \mu + \sigma^2 x_1 \sqrt{-2w^{-1} \log(w)}$

The rest of the numerical operations involved in the prediction step, depend mostly on the system model. If it is a simple AR process for example, the only expensive operation is the random variable generation itself, plus a couple of additions. But for non-linear models, the prediction step becomes more complex.

3.3.2 Update step analysis

The second part of a standard SIR PF iteration, is the update step, used to compute likelihoods (or weights) of the samples generated in the prediction step. Similarly to the prior step, the update step's computational complexity depends on the measurement model, usually described by the set of equations, relating the state vector to the observation vector. The numerical operations involved can be highly non-linear, and must be executed for each particle, resulting in huge computational overhead. Fortunately, there is no random number generation involved. But regardless of the intermediate numerical operations involved, there is always a non-linear similarity function involved to ultimately compute the likelihoods, that compares the current observation with "predicted" observations (as a result of propagating the particles' state vectors through the observation model). The most commonly used function is defined by Eq. 3.7 or ultimately – Eq. 3.11. As such, it contains a distance and exponentiation functions, with the rest of the operations involving constants being computed off-line. Thus, even when disregarding the measurement model, the update step is already quite computationally expensive, and a big bottleneck to the operation of the PF.

For the visual case, a model isn't even used, as was discussed earlier in this text. Instead, the likelihoods are directly computed, based on a similarity measure between visual cues of a template, that captures the appearance of the estimated object's position in a specific time frame, and image patches, centered around the predicted object locations in the current frame. In this case, the situation becomes even more severe, since computing the weights is accomplished through a series of image processing steps. In the best case, the complexity of the update step is roughly $O(NM)$, where M is the amount of pixels processed in a template/image patch, and N – the total amount of particles. This case applies for example to the histogram matching method described earlier, where $O(M)$ operations are required to construct the histogram for an image patch, and $O(N)$ – to construct and match the histogram for each patch, and then computing the likelihoods for the corresponding particles.

Another frequently overlooked bottleneck, is the weight normalization step. Using the simple L1-norm, involves N additions and divisions. However, one can first compute the reciprocal of the total sum of weights, to replace the divisions with multiplications, at the cost of floating point numerical errors. Overall, the computational overhead of weight normalization, compared to the previous steps is almost negligible, but it might start playing a role in some special cases.

3.3.3 Resampling step

The crucial resampling step also proves to be quite a bottleneck to the execution of a particle filter iteration. If an algorithm such as RWS (Alg. 2) is used, then the worst-case execution time of the resampling step is $O(N^2)$, where N is the amount of particles. However, since the CDF of the approximated posterior $\hat{p}(\mathbf{x}_k|\mathbf{y}_k)$ is monotonically increasing, this execution time is effectively reduced to $O(N \log N)$, by replacing the inner loop with a binary search algorithm. Moreover, if a systematic resampling algorithm is used, as described in e.g. [14], this execution time can go down to $O(N)$.

Among the operations involved, at most $O(N)$ pseudo-random number gen-

erations are performed. Fortunately most sampling algorithms rely on a uniform distribution. Thus, by itself, the resampling step by itself is not a very big computational burden to the particle filter.

The main pitfall of the resampling step in terms of computational efficiency, is that it reduces the possibilities for parallelization of the PF algorithm. As it will be discussed further in this thesis, the resampling step can introduce a global data-dependency in the flow of the algorithm. To elaborate further, suppose that the PF is implemented in parallel, by splitting processing of particles among multiple tasks. Essentially, this results into many smaller particle filters, which execute the prediction and update steps locally on the divided population of particles. However, resampling still has to be performed on the whole population, prior to proceeding further with the next PF iteration. Then a situation occurs, where each of the tasks wait for the resampling step to finish, before carrying on with its respective local prediction and update steps. This particular parallel implementation (illustrated in Fig. 3.4b) is further addressed in the section to follow.

3.4 Particle Filter Acceleration

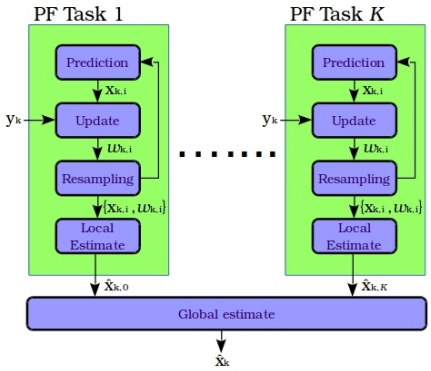
By now, the reader should be able to understand the algorithmic complexity and associated bottlenecks of a typical PF implementation, which in its standard form make it unsuitable for high-throughput, real-time applications. A lot of the times, the execution time of the PF on a single processor machine can be significantly improved, by making smart design choices, such as optimal parameter selection, choosing appropriate algorithms for each operation, etc.

Yet, there are parts encountered in almost every particle filter implementation, that simply cannot be improved any further in software, on a single processor based machines. Therefore, to further increase the throughput of software based implementations, one can consider a parallel implementation, that utilizes more than one processor. If that is also not enough, certain parts of the algorithm that limit the throughput, can be off-loaded to hardware accelerators. Ultimately, a complete hardware implementation of the filter is also possible.

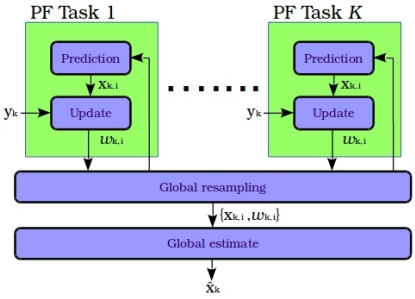
In this section, some typical multi-processor Parallel Particle Filter (PPF) implementations and their associated issues are discussed. Then the focus of the discussion is shifted to hardware acceleration of the filter. Specifically, an attempt is made to answer questions, related to the parts of the algorithm which deserve to be accelerated, and the trade-offs between a fully hardware and fully software based implementations. These and similar notions are discussed here in detail, accompanied by references to modern state-of-the-art approaches for their realization. The approaches mentioned, directly influenced the final implementation of the PF based visual tracker in this research, thus it is useful for the reader to understand the concepts and motivation behind them.

3.4.1 Parallel Software Implementation

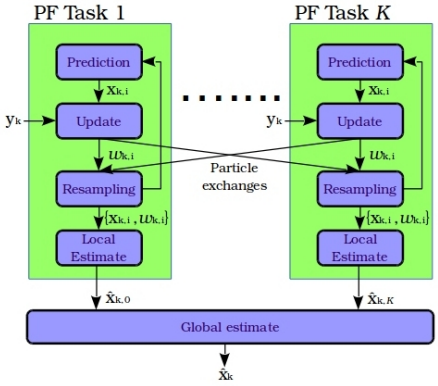
The most straightforward way to accelerate the PF algorithm, is to distribute its computation among many processing units. The total amount of particles is



(a) PPF with local resampling



(b) PPF with global resampling



(c) PPF with local resampling and Particle exchange

Figure 3.4: Three prominent PPF implementation schemes. The edges represent deterministic data transfer channels.

divided by the number of processors into sub-groups, which are then independently processed by their respective processor. If the computational effort of the PF is distributed among K cores, then the resulting implementation can be represented as a set of small, independent PFs. Each sub-filter produces its own local estimate of the state. A global estimate is then derived, by aggregating all of the local estimates. A functional block diagram of the above implementation is illustrated in Fig. 3.4a.

The straightforward parallel implementation has some limitations in terms of accuracy. Because each the sub-filters work independently of one another, with a reduced amount of particles, the accuracy of the local estimates will be reduced. This wouldn't be a problem, if the resampling was not included. As the reader may be aware by now, the purpose of the this step is to generate a fresh new set of particles, thus eliminating the so called degeneracy problem, and also increasing the diversity of the resampled particles and their chances to produce more accurate results. But to effectively do that, particles need to be resampled from the global posterior distribution. If each of the sub-filters

performs this step locally, the diversity of the resampled particles depends only on the locally approximated posterior density, which is directly affected by the amount of particles. Having less particles effectively means, that the chances of producing more various and effective off-spring is also reduced. Ultimately, each sub-PF tends to have a different idea of what the state might be, resulting in the global estimate being too far stretched, and not a good approximation of the actual result.

Resampling globally on the other hand, produces the same result as a non-parallel Particle filter, which is the most statistically suitable and accurate. However, a bottleneck is introduced, where each of the local PFs need to wait for the step to complete, before proceeding with a prediction during the next iteration. The data-dependency causing this bottleneck, is pointed out in the functional block diagram of this particular PPF implementation, as illustrated in Fig. 3.4b. Additionally, sharing memory between many processors also leads to reduced performance, since deterministic access needs to be ensured.

A hybrid approach is to share small amounts of particles with a number of "neighboring" filters, then resampling locally[27, 28], as opposed to resampling the whole set globally and then distributing it again to each of the tasks. The idea behind this approach is to reduce the overhead of sharing particles, due to moving of data, deterministic memory access, etc, while still maintaining reasonable accuracy. This gives freedom to the designer to exploit the trade-off between performance and accuracy. Also, this implementation can take advantage of built in multi-processor NoC communication architectures, as is done in this thesis for *Starburst*. This final configuration of the PPF, is illustrated in Fig 3.4c.

The key to this approach, is to send particles, which will guarantee variety in the new off-spring and keep each population more focused on the global estimate, rather than the individual local estimates. Typically, particles with relatively large weights are sent from one local population and appended to another. A second option is to replace particles with small weights, but this would generally contribute to the sample impoverishment problem. A third option is to exchange particles directly. Tests have shown that even a single shared particle per task can significantly improve the global estimate of the fully parallel PF with local resampling, as opposed to no sharing of particles at all[28].

3.4.2 Hardware acceleration

Besides the software solution of executing the algorithm in parallel on a multiple processors, one can opt to implement the algorithm completely in hardware[29]. Parallelism can then be exploited much easier than in software, while achieving very high throughput. This all comes at a great cost however - flexibility.

During design stage, it is often the case, that a parameter or a specific algorithmic step needs to be changed, such as the system model, or the number of particles. In some implementations, these parameters are even adaptively selected, like the visual PF described in [18]. Software based solutions are much easier to reconfigure in that respect. Therefore, it is generally recommended to implement the algorithm in software.

With that said, one can still benefit significantly from hardware acceleration to speed up the execution of the algorithm. As elaborated previously, there are a lot of common numerical operations, that can be accelerated in hardware,

while executing the essential tasks of the algorithm in software. This allows fast acceleration of the PF, without sacrificing too much flexibility.

To the very least, a fast and reliable, hardware multivariate RNG useful for any flavor of particle filters. Pseudo-random number generation is perhaps the "heart" of the SMC method, and a common operation that can easily be off-loaded to a hardware accelerator. Since the majority of system models(and to an extent measurement models) rely on normally distributed random processes, it is beneficial to also complement the accelerator with a full Gaussian RNG. A good implementation, based on the Box-Muller transform and the CORDIC algorithm, is described in [30].

Another useful operation, that can be implemented in hardware is the likelihood function, used to compute the weights for each particle. As pointed out earlier, the similarity measure between a real and predicted observation, is usually computed using a distance function, while the likelihood is computed using a Gaussian function. These are functions, which can easily be designed in hardware, and off-loaded from the processor.

The resampling step is also common to all filter designs, easily accelerated in hardware (especially the RWS algorithm). After all, the operations involved are mostly simple comparisons and data swaps. However, accelerating the resampling step in hardware is not of critical importance. Most implementations can get away with a fully software realization. Perhaps random sampling from a uniform distribution can be off-loaded to an accelerator, which is provided from previous accelerators.

Finally, for the visual PF tracker, acceleration is definitely desired during the image processing stages, since pure software solutions are usually insufficient. Because the methods involved vary in nature and function, a unique image processing accelerator is required for each case. Therefore during design, careful choices must be made, such as selecting adequate algorithms that map well in hardware, and ensure adequate results. Otherwise one may spend too much development time and resources, for design and testing of hardware accelerators that don't even satisfy the requirements.

3.5 Design considerations

The particle filter is a very flexible estimation technique, whose accuracy and complexity can vary a lot, depending on the observed system and the associated design choices. This section focuses on some of the design choices from the author's point of view, accompanied by a short discussion. Like in section 2.3, the purpose is to introduce the reader to the design flow behind the PF implementation. Note that although most of the considerations apply for the generic PF, the focus is strictly on a visual object tracking framework.

3.5.1 Number of Particles

Although the number of particles depends on many factors, like the dimensionality of the state vector, tests have shown that using an amount of 400-500 particles is enough, if only the location of an object is to be estimated. This number may double, if velocity and/or scale is incorporated, however the system can still adequately track an object with a lesser amount. Alternatively,

the amount can be also dynamically estimated, based on the state of the object.

3.5.2 Resampling Algorithm

Many sampling algorithms have been tried, such as RWS, Vose's Alias method[31] and Stratified sampling as described in [32]. Of all algorithms, the Systematic Resampling algorithm as proposed in [14], achieves the best trade-off between statistical and computational performance. A reference pseudo-code description is given in Alg. 5. Notice the similarity with Alg. 2.

Algorithm 5 Systematic Resampling

function $\{\mathbf{x}^*_{k,i}, w^*_{k,i}\}_{i=1,\dots,N} =$ Resample$(\{\mathbf{x}_{k,i}, w_{k,i}\}_{i=1,\dots,N})$
Input : Old particle set of the current iteration
Output : New set of particles

Initialize CDF: $c_1 = w_{k,1}$
for $j = 2 : N$ **do**
    Build the rest of the CDF: $c_j = c_{j-1} + w_{k,j}$
end for
Start from beginning of the CDF: $j = 1$
Draw a uniformly distributed starting point: $u' \sim \mathbb{U}[0, N^{-1}]$
for i = 1:N **do**
    Move along the CDF: $u_i = u' + N^{-1}(i-1)$
    while $c_j < u_i$ **do**
        $j = j + 1$
    end while
    Assign new particle: $\{\mathbf{x}^*_{k,i}, w^*_{k,i}\} = \{\mathbf{x}_{k,j}, \frac{1}{N}\}$
end for

3.5.3 Motion model

The motion model is based on Eq. 3.19. The parameters are fixed and manually selected. No learning techniques are used to derive them. Initialization of the model is based on detections from the HOG-SVM algorithm, by randomly picking the location of the first detection that occurs. Only a one target at a time is considered for tracking, although it is possible to track multiple target simultaneously. ' If a target is lost, due to some criteria, re-detection occurs with hopes of finding the target again. Otherwise, a new target is picked.

3.5.4 Observation model and features

The observation model is based on extracting color histograms from image patches at proposal object locations, as elaborated in 3.2. Colors provide a good means to distinguish the target from other potential targets and background disturbances. Being the main means of object observation and matching for many implementations, this model solidifies its place here as well. The similarity measure is based on the Bhattacharyya distance, as defined in Eq. 3.21. To fortify the tracking process, HOG-SVM detections are also incorporated, by providing an extra "prediction" for the update step. A subtle implication of this, is that the detector is still put to good use after the initial detection(s).

Chapter 4

System implementation

The object detection and tracking system consist of two top-level components: a video acquisition and visual object detection component, based on the HOG-SVM detector, and an object tracking component, based on the visual Particle Filter algorithm. In its complete state, it can be used as part of a high-level safety-critical application, such as collision avoidance on a self-driving vehicle. This chapter describes the functionality and design details of the these two components in a top-down design approach.

Since the *Starburst* platform is used as basis for development and evaluation of the implementation, it is briefly described in the first section. By providing a glimpse into the underlying architecture and principles of the MPSoC, this section tries to justify the reasoning behind its utilization in this sophisticated computer vision framework. The CMOS camera peripheral and its underlying principles of operation, is also described in this section, since it serves as an extension to the *Starburst* platform, independently of the HOG-SVM and PF tracker components.

The design and operation of the detection and tracking components, is explained in the last two sections. Since these two top-level components are comprised of lower level modules, based on different processing principles, a sub-section is provided for each of them. There, the reader is first briefly introduced to the techniques and principles utilized in each module. Then the software and/or hardware implementation of these sub components is described in detail, with help of block-diagrams and graphs.

4.1 Starburst MPSoC

Starburst is a configurable MPSoC, developed at CAES research group of the University of Twente, the same group where this research is also taking place. It is a powerful system, that utilizes many embedded processors and different hardware accelerators, to perform deterministic, real-time computing. *Starburst* was originally developed to help the design and analysis of streaming applications. It is also used as a tool for research in the field of multiprocessor real-time system analysis. These properties make this platform particularly suitable for development and analysis of real-time parallel software algorithms, such as the PPF. At the moment, the MPSoC is being deployed on a Xilinx Virtex 6 FPGA,

allowing easy implementation and testing of various streaming applications. An easy configuration interface is provided via scripts, which generate the required project files to synthesize the hardware using the Xilinx tool-chain. Based on the selected amount of cores and other associated resources, the system is shown to scale linearly in space [33].

An early version of the underlying architecture, consists of variable amount of general purpose processing tiles and hardware accelerator tiles, interconnected via a "*Nebula*" unidirectional ring NoC. Data is forwarded from one tile to a neighboring one along the ring, in a stream-like fashion. Additionally, each of the processing tiles are ensured deterministic access to shared resources, such as memory, peripherals, etc, via a second, tree based "*Warpfield*" interconnect.

A basic block diagram of the architecture is illustrated in Fig. 4.1. The illustration shows a standard multiprocessor configuration with no hardware accelerators. Each processing tile (denoted as $P_i,\ i = 0, ..., N-1$ on the illustration) is composed of a Xilinx Microblaze CPU core, data and instruction caches, a timer and additional scratch-pad memory, used for inter-processor communication and additional storage. In particular, data is transfered from one processor to another in a stream-like fashion, via the *Nebula* ring NoC, using a software C-FIFO API, based on the C-HEAP protocol [34]. In addition, each of the processors executes a real-time, POSIX compatible kernel called "*Helix*", which supports the standard C/C++ libraries and the `pthreads` API. The kernel utilizes a TDM scheduler for context switching between threads, by making use of a processing tile's timer module in interrupt mode.

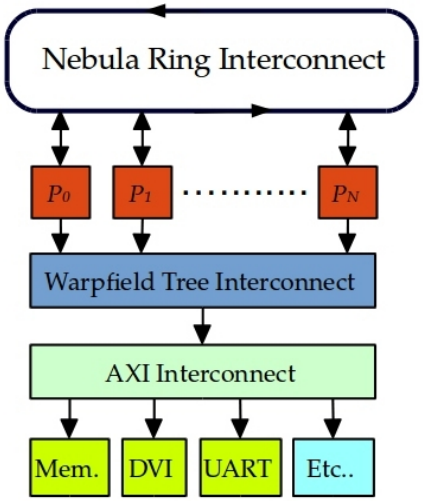


Figure 4.1: Block diagram of a typical, processor only *Starburst* configuration

Since data "travels" in only one direction, an issue arises when incorporating accelerator tiles for multiple streams. To push data into an accelerator and then receive the processed result for a particular stream, two processing tiles are required to control the data-flow of an accelerator. Thus, processing multiple streams without sacrificing the acceleration advantages gained, requires the duplication of accelerators, resulting into more area "consumed" by the MP-

SoC on the FPGA. Recent improvements to the platform resolve this issue, by allowing sharing of accelerators by multiple processors. However the detector and tracker implementations don't utilize these features, since (for now) none of the hardware components described here rely on the *Nebula* NoC for communication. Nevertheless, it is interesting to point this out for future references. For a more detailed documentation and description of *Starburst*, the reader is encouraged to review [35, 33, 36].

4.2 HOG-SVM detector

Starburst is a very powerful and flexible development platform. However its software processing capabilities are not sufficient for "smooth", real-time execution of the HOG-SVM processing pipeline. Even though real-time behavior can be guaranteed, the high-throughput constraints cannot be satisfied that easy. For example, tests have shown that even the first few stages[1] of the algorithm, implemented in software on multiple cores, can barely reach a throughput of 1 FPS. A purely software solution is therefore not suitable.

In this section, a purely hardware based implementation of the HOG-SVM detector is proposed, which can theoretically reach near maximum throughput, limited only by the acquisition rate of the image sensor (15 – 30 FPS). Such high throughput is greatly desirable, to provide a safely critical system with sufficient amount of data on time, and therefore – reduce the risk of error.

4.2.1 Overview

Although relying on the principles and design considerations discussed in Chapter 2, the implementation doesn't necessarily follow the exact same algorithmic structure. Each of the stages of the algorithm, has its own hardware design module, as seen from the block diagram in 4.2. The detector IP is designed to provide very simple hardware interfaces for data received from a standard CMOS image sensor and a Microblaze core. The input from the camera peripheral is a stream of RGB format based pixels, assumed to arrive along the width of a currently captured frame image, in a raster-scan like fashion. The output of the last module, is a binary stream of 1-bit values, with a logical '1' indicating the presence of a detected object and '0' – otherwise. In addition, a software configuration interface is provided, such as PLB or AXI, to the internal registers and memory which hold the SVM coefficients and bias values. These interfaces are also used to capture the output data, directly through a processor's data bus.

The modules make use of a straightforward streaming data interface. To maximize the throughput, the interface doesn't incorporate any handshaking signals, with the only exception of a "valid" signal. The valid signal is asserted, upon the full completion of an operation for a given module, on an "accepted" data sample. Therefore, the interpretation of this signal by "successive" modules, depends on the type of data and operation being priorly executed. However, its main purpose is to notify each processing module, when to accept and

[1] The first few stages of the HOG-SVM algorithm consist of gradient kernel filtering and polar form conversion – seemingly trivial operations

not accept data (and thus not update its state registers[1]), marked by a preceding module as "incomplete" or "invalid". An additional *enable* signal is provided to some of the modules, giving the ability to stall the image processing pipeline. The purpose of this stalling, as well as the function of the clock-domain crossing FIFO is explained later in this section.

There are several possibilities to interface the IP to *Starburst*:

1. Directly to one of the system's processing tiles.
2. To "Warpfield", thus sharing the peripheral to all of the processors, but limiting the throughput.
3. To the "Nebula" ring, acting as an accelerator.
4. Directly to the memory controller, using a DMA module.

Additionally, a hardware FIFO buffer is required to ensure no loss of data. For now, the detector doesn't take advantage of the hardware accelerator infrastructure of *Starburst*. There are a few reasons for this:

1. The hardware design is no yet compatible for the Nebula ring interconnect.
2. The IP is accessed only by one, specific MB core, eliminating the need to share it to others through the interconnect.
3. Maximum throughput is achieved, when relying on a direct data interface to a processor.

Thus, for development purposes, it is much more convenient to utilize one master MB core that reads data from the detector, as a gateway or "proxy" to other Microblazes, while managing the data-flow and configuration of the IP.

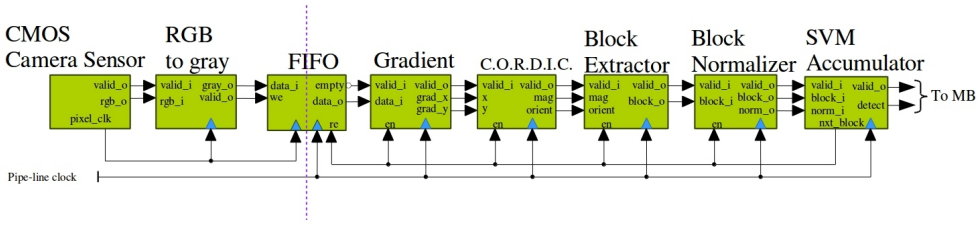


Figure 4.2: Structural diagram of the HOG-SVM detector

4.2.2 CMOS Camera peripheral

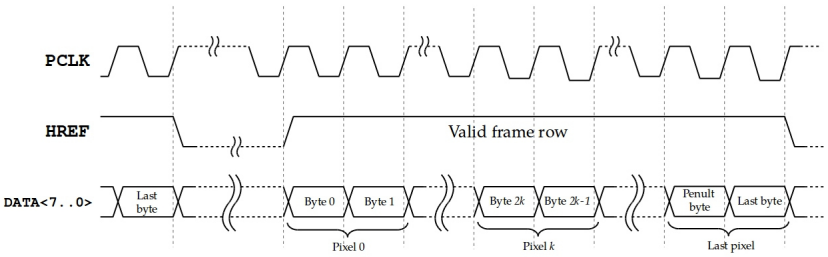
The CMOS camera peripheral provides a link between the physical connections of a standard CMOS image sensor to the FPGA, and the software and/or hardware components of *Starburst*. It is controlled by a single processing tile, by utilizing the PLB bus. Its main function is to structure the incoming raw data from an image sensor's data bus, into a stream of pixels, belonging to a currently captured frame. A separate peripheral utilizing a I^2C-like protocol, is

[1] An exception to this are the pipeline registers

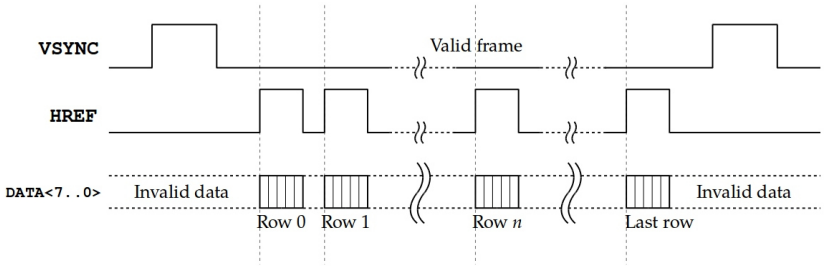
used for configuration of the image sensor's registers. Although the registers and configuration options vary per image sensor, some common configurations include a sensor's frame resolution.

Two main interfaces are utilized to transfer frame pixel data: a parallel or a serial interface. An advantage of the serial interface, is that the maximum achievable bandwidth of the transmissions is high, due to the differential pair signaling at high frequencies and the lower amount of connections. A disadvantage however, is the complicated scheme involved to decode the serial bit stream and capture a frame. This is partly due to the fact, that decoding protocols are usually proprietary.

A parallel interface on the other hand, utilizes many signals to directly stream frame pixels on each falling (or rising) edge of a pixel clock. It is quite similar to the standard VGA interface, and therefore requires very little effort to extract each pixel. As a consequence, the maximum bandwidth is not high, although high throughput is achievable because of the bus' large data-length (usually 8-bits). Also, connecting additional sensors becomes problematic, because of the high pin count per interface. Nevertheless, this peripheral utilizes the parallel interface to capture and stream pixel data from the camera into *Starburst*'s architecture, because of the associated simplicity to extract frame images.



(a) Signal waveforms of a valid frame row transaction.



(b) Signal waveforms of a valid frame transaction.

Figure 4.3: Signal descriptions and waveforms of a typical CMOS image sensor's parallel data interface

To understand the principle behind the operation of the peripheral, one may refer to figures 4.3a and 4.3b, illustrating typical signal waveforms produced by the majority of CMOS sensors when transmitting a frame image. One may observe, that this is a typical VGA pattern, where the `vsync` signal indicates the beginning or end of a video frame, while the `href` signal – the beginning and

Format	Row number	Byte number	Pixel component
YCbCr	even/odd even/odd	even odd	Y Cb or Cr
Raw/Processed Bayer RGB	even even odd odd	even odd even odd	R G G B
RGB444	even/odd even/odd	even odd	4 MSBs of R 4 MSBs of G and B
RGB555	even/odd even/odd	even odd	5 MSBs of R and upper 2 MSBs of G Lower 3 MSBs of G and 5 MSBs of B
RGB555	even/odd even/odd	even odd	5 MSBs of R and upper 3 MSBs of G Lower 3 MSBs of G and 5 MSBs of B

Table 4.1: Pixel byte interpretation per format

end of a row of pixels. Pixels are streamed in as pairs of bytes, presented on an edge of the pixel clock (in this case the rising edge), which is usually provided by the CMOS sensor itself. The interpretation of each pair depends on the pixel format configuration. The most common formats and their respective byte outputs for each pixel are described in Table 4.1. Here, *even* and *odd* refer to the first and second bytes in a pair, respectively, or evenness of a row's sequence number. Note that for raw or processed Bayer RGB, additional processing is required to "demosaic" the Bayer pattern. For YCbCr mode, the Cb and Cr bytes belong to two pixels.

Based on the interface signals' waveforms, a hardware frame acquisition module is quite straightforward to implement. It can be divided into two components: a frame capture module and a pixel grabbing module. The capture module detects the beginning and end of frame, by monitoring the `vsync` signal, thus giving a "green light" to the pixel grabber to start reading in pixel data. A state diagram of a general capturing sequence is illustrated in Fig. 4.4. This sequence is started and monitored by a processor through a PLB slave interface.

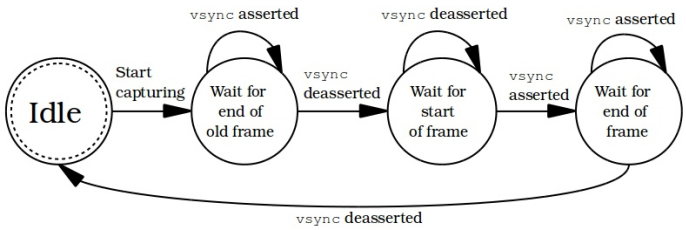


Figure 4.4: Signal descriptions and waveforms of a typical CMOS image sensor's parallel data interface

The pixel grabber waits for a "valid frame" signal from the frame capture

module, until it can start assembling each pixel, based on one of the formats in Table 4.1. It asserts the `valid` signal, once a pixel is fully assembled and upon assertion of the `href` signal. Together, the frame capture, pixel grabber and PLB slave modules form the CMOS image sensor peripheral, which can now stream pixels directly into an image processing pipeline, and/or into the master processor's data bus.

4.2.3 Gradient filter module

The gradient filter module calculates the x and y components of the image gradient, using the horizontal and vertical kernels, $[-1, 0, 1]$ and $[-1, 0, 1]^\top$ respectively. It accepts as input a stream of unsigned 8-bit, luminosity (gray-scale) image pixels, and produces two 9-bit, signed gradient image streams as output. A simplified block diagram of this filter can be seen in Fig. 4.5.[1]

The horizontal gradient filter is quite simple and it utilizes perhaps the most minimal amount of resources. It performs a simple DSP operation, defined by the equation

$$y[k] = x[k] - x[k-2]$$

Note that pipeline registers (used to reduce the combinatorial path) are distinguished from the state registers. The *valid* signal enables or disables the state registers, to prevent invalid data from corrupting the state. The *enable* signal affects all registers, in a similar manner.

The vertical filter performs a similar operation as the horizontal, defined by the discrete equation

$$y[k] = x[k] - x[k-2W],$$

where W is the width of an expected frame image. Instead of storing the state in $2W$ registers, a circular buffer is used, realized as BRAM or LUTRAM on a target FPGA.

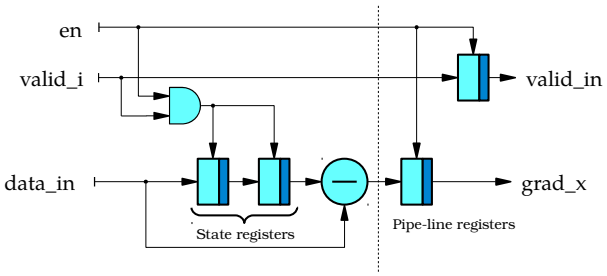


Figure 4.5: Block diagram of the image gradient filter

4.2.4 CORDIC module

To convert the image gradient from Cartesian to polar form, a CORDIC module is utilized in vectoring mode. The output of the module is a 9-bit, unsigned gradient magnitude stream, and a 16-bit fixed-point gradient orientation stream.

[1] For a description of each of the symbols, refer to appendix B

Since the pixel values of the gray-scale image are always in the range $[0; 255]$, there is no risk of bit overflow.

In vectoring mode, the CORDIC algorithm performs several iterations, based on the equations

$$\begin{aligned} x_{k+1} &= x_k - y_k * d_i * 2^{-k} \\ y_{k+1} &= y_k + x_k * d_i * 2^{-k} \\ z_{k+1} &= z_k + d_k * \arctan 2^{-k}, \end{aligned}$$

where $d_k = +1$ if $y_k < 0$, and -1 otherwise; $k = 1, ..., N$ with N – the number of iteration. As N approaches infinity,

$$\begin{aligned} x_N &= A_N \sqrt{x_0^2 + y_0^2} \\ y_N &= 0 \\ z_N &= z_0 + \arctan \frac{y_0}{x_0}, \end{aligned}$$

where

$$A_N = \prod_{k=0}^{N-1} \sqrt{1 + 2^{-2k}} \approx 1.64676$$

is a scaling constant. To compute the polar coordinates, x_0 and y_0 are set to the values of the current gradient components, while $z_0 = 0$.

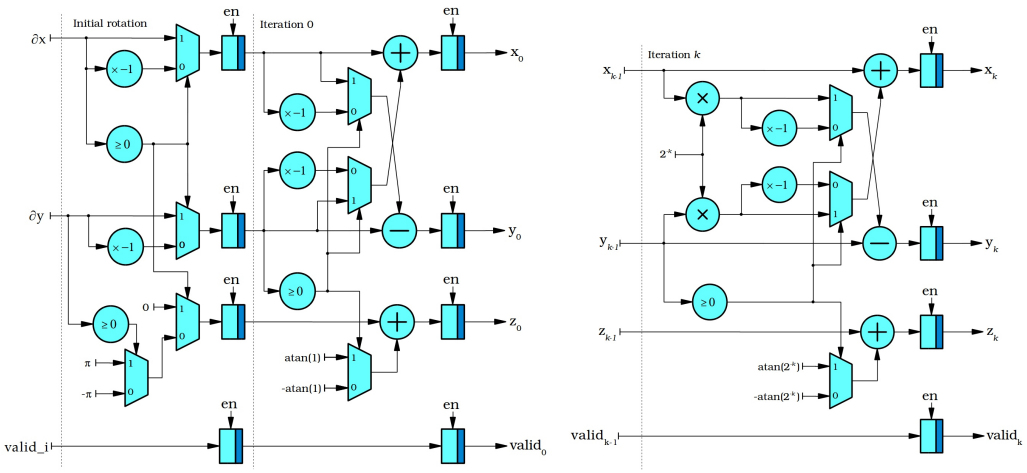
Since the algorithm is restricted to the angle range between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, an initial rotation of π or 0 radians is performed [37], such that

$$\begin{aligned} x_0 &= d * \partial x \\ y_0 &= d * \partial y \\ z_0 &= 0 \text{ when } d = 1; b\pi \text{ otherwise,} \end{aligned}$$

where $d = +1$ when $\partial x \geq 0$; -1 otherwise; and $b = +1$ when $\partial y \geq 0$; -1 otherwise, and ∂x and ∂y are incoming image gradient component values. Thus, the gradient magnitude and orientation after N iterations is approximated as

$$\begin{aligned} \|\nabla\|_2 = \sqrt{\partial x^2 + \partial y^2} &\approx A_N^{-1} x_N \\ \theta_\nabla = \text{atan2}(\partial y, \partial x) &\approx z_N. \end{aligned}$$

One of the great advantages of the CORDIC algorithm, and hence why it is being used in the HOG-SVM hardware implementation, is its simplicity. The algorithm can be easily implemented in hardware, by utilizing only simple bit-shifts, additions and comparisons. The inverse tangent values, can be pre-calculated and stored in a table. A hardware block diagram of the module, performing the initial iteration and rotation, is illustrated in Fig. 4.6a. A generic hardware module for the rest of the iterations, is illustrated in Fig. 4.6b. These modules can be easily chained together, to form a CORDIC based Cartesian to polar coordinate converter, of arbitrary number of iterations. Note that the modules operate on fixed-point data. Thus the accuracy of this module will depend on the number of iterations and the fixed-point word length. This implementation uses 9-bit integers for the magnitude, and 16-bit fixed-point values for the orientation with 10-bit fractional part. In addition, only 8 iterations are implemented.



(a) Initial CORDIC iteration and rotation module (b) Generic CORDIC iteration module

Figure 4.6: Hardware implementation modules of fully unrolled CORDIC algorithm in vectoring mode

4.2.5 Block extraction module

The *block extraction* module is rather complicated, compared to the earlier components of the pipeline. Its purpose is to directly extract HOG block feature vectors, for every newly arrived pixel. Each histogram is updated on every valid incoming gradient magnitude and orientation pixel from the CORDIC module, as the block window is being "slid" across a currently captured frame image. An extracted block vector is then directly used to compute a series of partial SVM dot-products for all detection windows, which share the block on the image, as it will be reviewed later. This module assumes parameters discussed in subsection 2.3.1, and as such, its output is a 32-element vector, containing four concatenated 8-bin gradient orientation histograms of their respective cells.

The naive approach to directly calculate the histograms for newly arrived pixels, is to store sufficient amount of pixels in line buffers, and then recalculate the sums of pixels for each histogram bin. This method however, requires a lot of hardware resources and produces quite a long combinatorial path.

A better approach is to update the histogram bins, by only adding the most recent data, and removing the "oldest" set of data. More specifically, the histogram update scheme involves adding an incoming column of pixel values to an accumulator for a particular cell, and subtracting its respective last column of pixels. A graphical representation of this scheme is illustrated in Fig. 4.7 for a 2-by-2 block of 8-by-8 pixel cells. Here, the block appears to be "sliding" in a raster scan-like fashion along the width of a currently captured image frame, as columns of pixels in front of the cells are added to the histograms' bins, while the cells' last columns are subtracted. Given a $p \times q$ block of $m \times m$ pixel cells and histogram length L, this operation on an incoming frame image of expected size $W \times H$, can be mathematically expressed as follows:

$$h_{i,j}[k] = h_{i,j}[k-1] + S_m[k - K_j] - S_m[k - m - K_j] \tag{4.1}$$

where $h_{i,j}[k]$ and $h_{i,j}[k-1]$ are the new and old values of the i-th bin of the

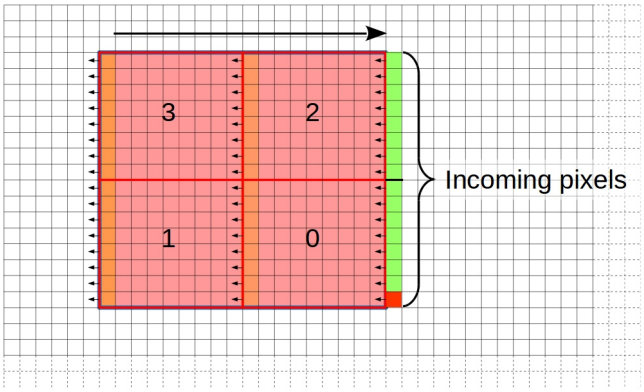


Figure 4.7: Illustration of a "sliding" 2-by-2 block of 8-by-8 pixel cells, along the width W of the gradient magnitude and orientation images. The red pixel refers to $gm[k]$ and $go[k]$, while the green pixels refer to $gm[k-Wl]$ and $go[k-Wl]$; $0 < l \leq 15$

j-th cell's histogram, and

$$S_m[k] = \sum_{l=0}^{m-1} d_i[k - Wl] * gm[k - Wl],$$

with $d_i[k] = 1$ when $go[k] = i \in \mathbb{N}_{<L}$; 0 otherwise, $go[k] = \left\lfloor \frac{|\theta_\nabla[k]|}{\pi} \right\rfloor \cdot L$ being the quantized unsigned gradient orientation, and $gm[k]$ - the gradient magnitude stream. K_j is an offset which depends on the position of cell j in a block, based on its bottom right-most pixel. For example, $K_0 = 0, K_1 = 8, K_2 = 8W$ and $K_3 = 8W + 8$, if assuming the cell numbering in Fig. 4.7.

Note that this recursive relation introduces blocks, which are "split" out across the image. In other words, some columns of the block are still on the other side of the image, as illustrated in Fig 4.8. Additionally, the first few rows of a new frame need to be buffered, until a block is within the frame's bounds. Thus a block is considered valid, when $k \geq W(p \cdot m - 1)$ and $k - \left\lfloor \frac{k}{W} \right\rfloor W \geq q \cdot m - 1$. The `valid` signal can be used to invalidate such blocks, by keeping track of the current row and column of the gradient image frames.

The hardware design begins with an elaborate buffering topology, to allow parallel processing of the gradient magnitude and orientation pixels in a block, as defined in equation 4.1. Since a single BRAM cannot be used due to the required amount of reading ports, a combination of registers and RAM based circular buffers are used for the buffering. This topology is illustrated in Fig. 4.9, where a 16-by-16 register matrix is used to store and access pixels from the active block, and 15 circular buffers for the remaining pixels on each row.

There is a separate buffer for the gradient magnitude and orientation streams, however the gradient orientation is quantized, prior to being fed into its respective buffer. This reduces the data bit width of the buffer and therefore saves considerable amount of hardware space. The quantization function is implemented as a series of comparators, instead of relying on a division and multiplication. Since the expected hardware is difficult to describe as a block diagram, the function is described in Alg. 6. One can easily unroll the `while` loop and

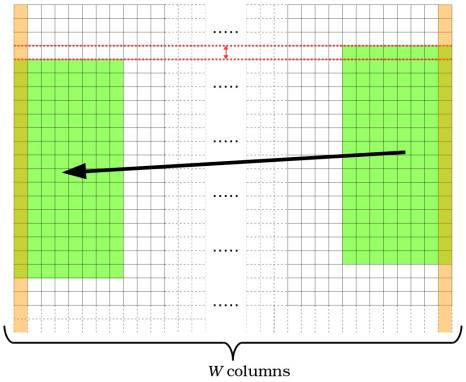


Figure 4.8: A situation, where the block is split across the image due to buffering

implement the hardware comparators in parallel, with some additional decoding and multiplexing logic.

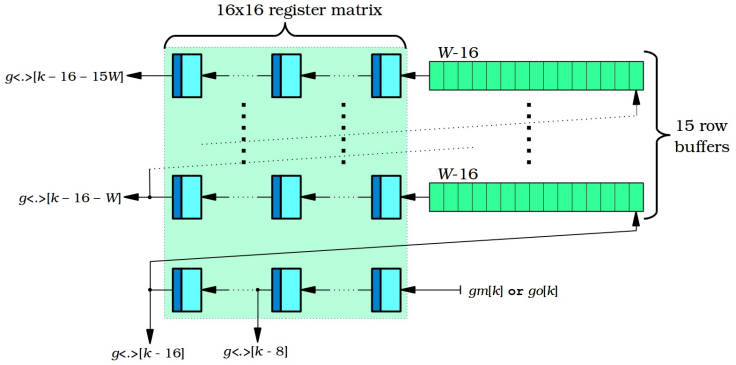


Figure 4.9: A buffering topology for the gradient magnitude and quantized orientation streams.

Algorithm 6 Gradient orientation quantization function

function $i = \text{binidx}(\theta_\nabla)$
Input : Gradient orientation
Output : Histogram bin index $i \in \mathbb{N}_{<L}$, where L is the total bin amount

1: Initial index: $i = 0$
2: Initial angle step: $d = \frac{\pi}{L}$
3: **while** $d < |\theta_\nabla|$ and $i < L - 1$ **do**
4: $\quad d = d + \frac{\pi}{L}$
5: $\quad i = i + 1$
6: **end while**

After buffering, the outputs of the gradient magnitude register matrix are fed into a series of adder trees. Each histogram bin has its own pair – one for $S_m[k - K_j]$ and $S_m[k - m - K_j]$, as illustrated in Fig. 4.10. To incorporate the

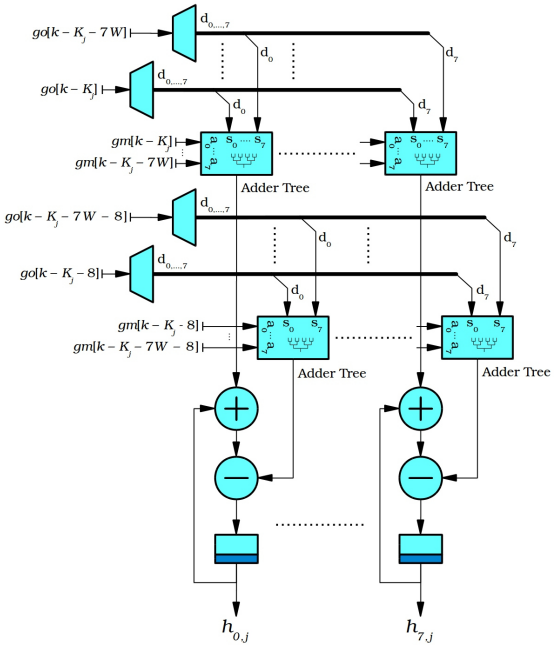


Figure 4.10: Bin accumulation hardware for one 8×8 cell j, with histogram length of $L = 8$.

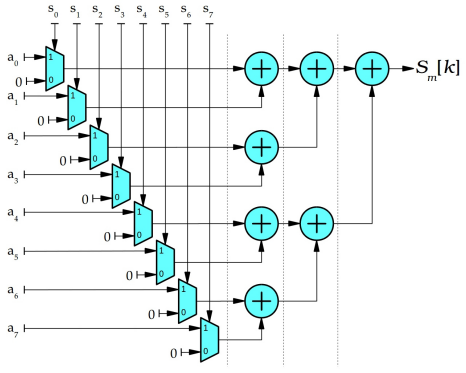


Figure 4.11: Adder tree with selective input. The dashed lines indicate potential pipe-lining.

multiplication by $d_i[k]$, each input of an adder tree is equipped with a selector switch between 0 and $gm[k]$, as seen in Fig. 4.11. The switches are controlled by decoded signals, coming from the bin index register matrix. The outputs of the histogram bin accumulators are all concatenated to form an output HOG feature vector, updated on each newly arrive valid pixel. This is perhaps the main advantage of this module, but it comes at a great hardware cost.

4.2.6 Normalization module

The normalization module computes the L1 normalization factor from a valid HOG block feature vector produced earlier, according to eq. 2.8. It is the simplest norm to compute, involving only one division and a series of additions. It is very important to note however, that the module itself doesn't normalize the block vector, but rather only computes the scale factor, which is later directly incorporated in the dot-product during classification. This saves up a lot of multiplications.

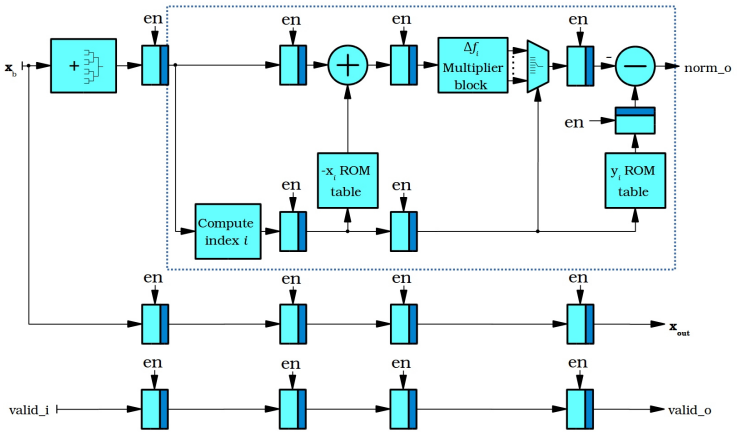


Figure 4.12: Block diagram of the HOG block vector normalizer. The dotted lines indicate the linear interpolation block for the division.

The block diagram of the design is illustrated in Fig. 4.12. The sum of vector elements is implemented as an adder tree, while the one clock cycle division - using a linearly interpolated approximation block. It utilizes the standard linear interpolation formula

$$y = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - y_i}(x - x_i) = y_i + \Delta f_i \cdot (x - x_i)$$

such that $y_i = \frac{1}{1+x_i}$ and $x_i \leq x < x_{i+1}; i \in \mathbb{N}_{<N}$, where N is the length of the interpolation tables. The domain range of the division is also known, considering that the value of an element from the feature vector is always bounded by $\mathrm{x}_{max} = p \cdot q \cdot m^2 \lfloor \sqrt{2\, g_{max}^2} \rfloor$, where $g_{max} = 255$ is the maximum value of the gradient for 8-bit pixel intensity values. This follows from the fact that the sum of bin values in a histogram is equal to the sum of pixel values in a $m \times m$ cell from a $p \times q$ block. One can thus compute and store the values for x_i, y_i and Δf_i in ROM on the FPGA.

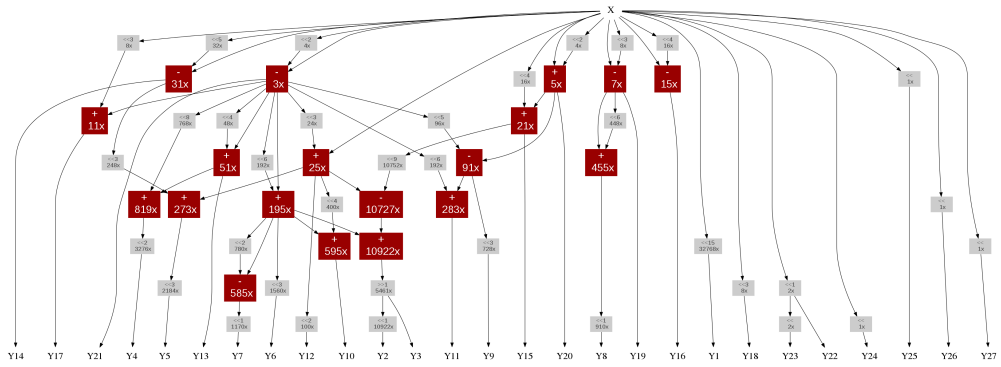


Figure 4.13: Δf_i multiplier block.

What makes this module particularly interesting, is how $\Delta f_i \cdot (x - x_i)$ is directly implemented as multiplier-less coefficient multiplication. It uses only adders and shifts to compute multiple fixed-point output multiplications from a single input, while sharing as much resources as possible. Techniques and tools, as described in [38], are used to generate the synthesis code for the multiplier block. A sample block diagram generated this way for 18-bit fixed-point values with 16-bit fractional part can be seen in Fig. 4.13.

4.2.7 SVM classification module

The final hardware module of HOG-SVM pipeline is the SVM classification module. The design is very closely related to the one introduced in [11], with the exception that this implementation also allows incorporation of more partial dot-products to increase the throughput, while working with lower pipeline clock frequencies to reduce power usage. It accepts as input a HOG block feature vector and a normalization factor. The idea behind its operation, as described in [11], is to compute the dot-product from Eq. 2.10 in parts, instead of directly, such that

$$y_k(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b \; = \; \left(\sum_{i=1}^{N} \mathbf{w}_i \cdot \mathbf{x}_i \right) + b$$

where k is a detection window index, $\mathbf{x} = [\mathbf{x}_1 \; , \; \mathbf{x}_2 \; , ..., \; \mathbf{x}_N]$ is the HOG feature vector, consisting of N concatenated block vectors and $\mathbf{w} = [\mathbf{w}_1 \; , \; \mathbf{w}_2 \; , ..., \; \mathbf{w}_N]$ – the associated SVM weight vector. A recursive relation then follows

$$\begin{aligned} y_{0,k} &= b + \mathbf{w}_0 \cdot \mathbf{x}_0 \\ y_{i,k} &= y_{i-1,k} + \mathbf{w}_{i,k} \cdot \mathbf{x}_{i,k}, \end{aligned} \tag{4.2}$$

which is executed on each incoming block vector from the block extraction and the normalization modules, “accumulating” the dot-product. It is here, that one can also include the normalization factor from the earlier module, such that

$$\mathbf{w}_i \cdot \mathbf{x}_i = \frac{\mathbf{w}_i \cdot \mathbf{v}_i}{\|\mathbf{v}_i\|_1},$$

where $\mathbf{v}$ is the non-normalized block feature vector.

To save additional hardware, the module takes advantage of the fact that a block can be shared by multiple detection windows, as illustrated in Fig. 4.15. Thus, for a currently available block vector, the relation in eq. 4.2 can be sequentially executed. One must then take care to compute the appropriate window index k and block index i within the window. [1]. Additionally, to meet the required throughput requirements, the clock frequency of the SVM classifier is much higher than the pixel clock frequency of the CMOS image sensor, prompting the use of a dual-clock FIFO buffer, as seen in Fig. 4.2. The use of an `en` signal to stall the pipeline and state registers in previous modules, now also starts to make sense – the current block vector value must be kept by the SVM classification module at its input, until it finishes computing the partial dot products that share the respective block. One may argue, that it is simpler to move the FIFO just before the input of the SVM accumulator, but this would result in huge resource usage, because of the dimensionality of a block vector and the associated bit widths of its elements.

A simplified top-level block diagram of the SVM classification module can be seen in Fig. 4.14. As one can observe, a synchronous BRAM is used for the SVM window accumulators, and for the partial weight vectors. A specialized memory controller is used to compute the window index k and block index i, and control the data flow accordingly. It only works on assertion of the `valid_i` signal. Completion of a window's dot-product is signaled, by checking whether $i = N - 1$ (the last block index in the window is reached) and asserting the `valid_o` signal. The `next_block` signal is used to stall or enable the HOG pipeline and reading from the input FIFO. Finally, a detection bit-stream is directly generated, by comparing the accumulated score with 0.

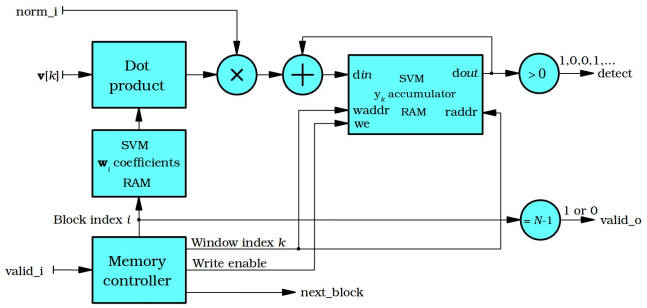


Figure 4.14: Block diagram of the SVM classification module.

To increase the throughput and process multiple detection windows per block, the hardware logic can be duplicated, while the memories – split into D smaller BRAMs, where D is the duplication factor. A memory controller is thus assigned for each of the coefficient and accumulator BRAM pairs. Nothing changes in the hardware function of the SVM module, except the memory controller, which requires a specific scheme to compute the indexes. The scheme to compute the indexes is discussed further in the report, during analysis.

[1] Note that for any window k, that shares a block with index i within the window, $\mathbf{x}_{i,k}$ is actually the same block vector present at the input, but different in the context of a detection window.

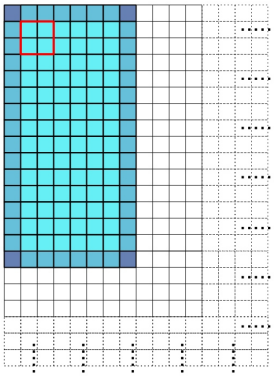


Figure 4.15: Sharing of a block (indicated in red) by four overlapping detection windows at the top-left corner of the image frame. The brighter the color, the more overlap introduced.

4.3 Multicore Parallel PF

Unlike the HOG-SVM detector, the particle filtering algorithm is implemented completely in software, with the idea of evaluating its performance on the multicore infrastructure of *Starburst*. As was discussed in the previous chapter however, hardware acceleration can and will be incorporated in the algorithm for generic and computationally intensive parts, in a future version of this implementation.

Nevertheless, this implementation relies on the parallel PF topology discussed in subsection 3.4.1 and [27, 28] by Chitchian et al. In particular, the topology consists of multiple processing cores, which execute all three steps of the SIR particle filter on a distributed population of particles. To improve the accuracy of tracking, particles are also exchanged between processing cores, before the resampling step. A key difference between this implementation and the one discussed in [27] or [28], is that here, the parallel PF topology takes complete advantage of the deterministic ring NoC present in *Starburst*, as opposed to their GPU based design, which is neither embedded, nor a real-time solution (even though the title claims so). Even though the computational speed of the GPU based implementation is impressive, one cannot make guarantees about its deterministic and real-time behavior. On the other hand, incorporation of hardware acceleration in *Starburst* can potentially bring very similar performance in our implementation, while still satisfying the imposed real-time constraints.

This section is not as elaborate as the previous one, since most of the important features of the PF algorithm have been discussed in Chapter 3, however it will focus on the mapping of the PPF algorithm to *Starburst*. Specifically, the the parallel real-time task topology is discussed in more detail. Then, more details about the particle exchange scheme are reviewed.

4.3.1 PPF topology

The authors of [27] show that executing the PF algorithm completely in parallel has its problems with respect to accuracy, due to the resampling step. Their

solution to the problem is to exchange a small amount of particles, which are "fit" enough to introduce variety and "enrich" the local populations of particles of each sub-filter, without sacrificing much of the computation advantages gained from parallelization of the algorithm. They show that exchanging even a small amount of particles, using two different communication topologies, can significantly improve the tracking performance. Another important result from their implementation, is the comparison of all-to-all and ring topologies. According to their experimental results, a ring topology can at times outperform an all-to-all one. The implication of this find, is that a one-to-one mapping of the PPF implementation on *Starburst* using a ring topology, is not only optimal, but also accurate, taking full advantage of the multiprocessor ring NoC. Additionally, the NoC is also designed to be hardware cost efficient, with minimal communication time overhead, as opposed to other shared memory solutions.

Following this notion, the topology of this implementation is illustrated in Fig. 4.16 using a directed task graph. Here, each of the nodes in the graph represent real-time tasks, executed on its own processing core, and managed by the *Helix* real-time kernel through a `pthread` compatible API. The edges of the graph represent data precedence relations and unidirectional data channels. In essence, tasks communicate through software circular FIFO buffers, which don't rely on shared memory to transport data from one processor to another, but rather – through the ring NoC. The numbering of the tasks on the graph, reflects the sequence order of processors, as they are interconnected on the ring. Thus, the communication overhead from processor 0 to processor 1 for example is minimal. If the total number of physical processors is P_{tot}, then the communication overhead from 0 to $P_{tot} - 1$ is the highest, but it is minimal the other way around, since the ring NoC is unidirectional.

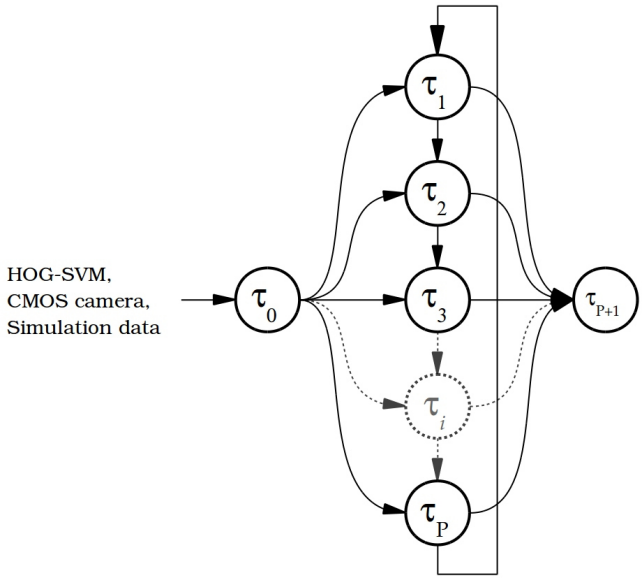


Figure 4.16: Parallel Particle filter task graph and communication topology

Each task in the graph executes a the distributed PPF algorithm as described in alg. 7, on a local particle population. Assuming that the total amount of

Algorithm 7 Distributed SIR Particle Filter Algorithm

for $i = 1 : N_{local}$ **do**
    Initialize $\{\mathbf{x}_{0,i}, w_{0,i}\}$, such that $\mathbf{x}_0^i \sim p(\mathbf{x}_0)$ and $w_{0,i} = \frac{1}{N_{local}}$.
end for
for each system iteration $k > 0$; $k \in \mathbb{N}$ **do**
    Acquire new measurement from processor 0: $\mathbf{y}_k =$ `read_fifo`(0)
    for $i = 1 : N_{local}$ **do**
        Draw a sample $\mathbf{x}_{k,i} \sim p(\mathbf{x}_k|\mathbf{x}_{k-1})$ using Eq. 3.5
        Assign a particle weight, $w_{k,i}$, using Eq. 3.11
    end for
    Exchange particles: $\{\mathbf{x}_{k,j}^*, w_{k,j}^*\} =$ xchg$(\{\mathbf{x}_{k,i}, w_{k,i}\}, D, A)$
    Resample particles: $\{\mathbf{x}_{k,i}, w_{k,i}\} =$ Resample$(\{\mathbf{x}_{k,j}^*, w_{k,j}^*\})$
    Compute local estimate $\hat{\mathbf{x}}_k$ using 3.12
    Send local estimate to processor $P+1$: `write_fifo`$(\hat{\mathbf{x}}_k, P+1)$
end for

particles used in a non-parallel PF would be N, distributed among P processors, then the amount of particles per task is $N_{local} = \lfloor \frac{N}{P} \rfloor$. For consistency and ease of analysis, it will be assumed from now on, that N is chosen, such that N_{local} is the the same for every task. Thus, each task executes a "small" version of the PF algorithm with N_{local} particles, in a fixed, predetermined amount of time.

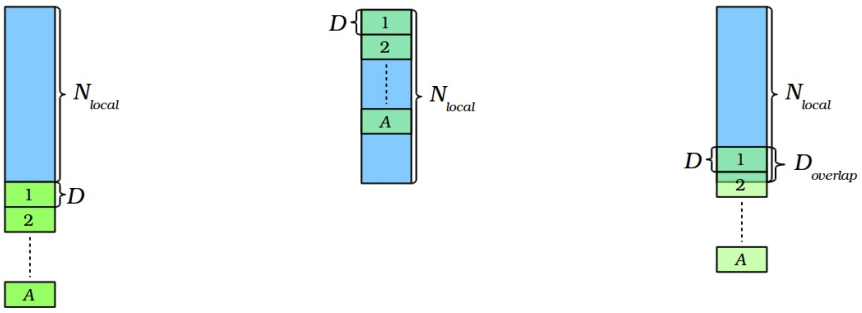
Each of the steps are performed locally, in the same manner as a non-parallel implementation. However, before any processing can begin, each processor $p > 0$ reads a new measurement from processor 0, through a dedicated FIFO buffer. This measurement can come from a sensor, or simulation data for evaluation purposes, but it is (for now) always relayed from processor 0. In a future version of the PPF implementation, the use of this core as a data distribution gateway will be omitted, and replaced directly with a hardware accelerator, such as the HOG-SVM detector or just the camera peripheral.

One may notice a newly introduced "exchange" step, inserted between the update and resampling. During this step, particles generated from the update are first sorted in descending order according to their weights. Then, each processor p with a task node τ_p shall send D amount of high weight particles to $A \leq P - 1$ amount of neighbors p_i, such that

$$p_i = \mod(p + i, P) \; ; \; 0 < i \leq A,$$

respecting the unidirectional nature of the ring NoC. The D amount of particles are then incorporated in the local population, forming a new set of particles $\{\mathbf{x}_{k,j}^*, w_{k,j}^*\}; j = 1, ..., N_{new}$, which is used to resample the local set. More details about the exchange scheme are reviewed soon.

After resampling, each of the cores sends its local estimate to core $(P + 1)$, which performs global estimation. However, as noted in [27], it is also sufficient to "pick" a local estimate from a PF task, as the estimation quality comes close to that of the global estimator. Nevertheless, processor $(P + 1)$ is also used for evaluation purposes.



(a) Exchange with appending. (b) Exchange with replacement. (c) Exchange with overlap

Figure 4.17: Different exchange strategies. Here, the blue block represents the old local set of particles, while the green blocks - the sets of particles, received from neighboring processors.

4.3.2 Particle exchange

The particle exchange step is very similar to the one introduced in [27], with the exception that here, full advantage is being taken of the *Starburst* MPSoC. First, before exchanging of particles can begin, produced particles are sorted in a descending order according to their weights, such that

$$w_i \geq w_j \ ; \ 1 \leq i < j \leq N_{local}.$$

The idea is to separate "weak" particles from "strong" particles, with higher likelihood of the state. The top D "strong" particles are then sent through a dedicated software FIFO buffer to A forward neighbors, as was explained earlier. Once a PF task finishes sending the particles (which usually costs a small time overhead), it goes on to receive a set of exchanged particles from all of its previous neighbors. The received set is "merged" with the local particle set into a new one, which is used for resampling.

Assuming that the particles are received from A neighboring tasks, there are several exchange strategies that a local PF algorithm can employ when merging the local set with the exchanged set:

1. Append the particles to the local set, such that $N_{new} = N_{local} + A \cdot D$.
2. Replace $A \cdot D$ amount of particles from the local population with a newly arrived batch.[1]
3. Do both, by introducing an overlap of $D_{overlap}$ particles, such that $N_{new} = N_{local} + A \cdot D - D_{overlap}$.

All of these strategies are graphically illustrated in Fig. 4.17. In the latter two strategies, one may wish to replace only the "weakest" particles, which is another reason to introduce the sort in the beginning of the exchange step.

Since only the top D particles are exchanged, one may not need to sort the whole particle set completely. Here, a partial sorting algorithm can be used to extract the first D high weighted particles directly, such as partial heap sort or partial Bitonic sort.

[1] Note that $A \cdot B \leq N_{local}$.

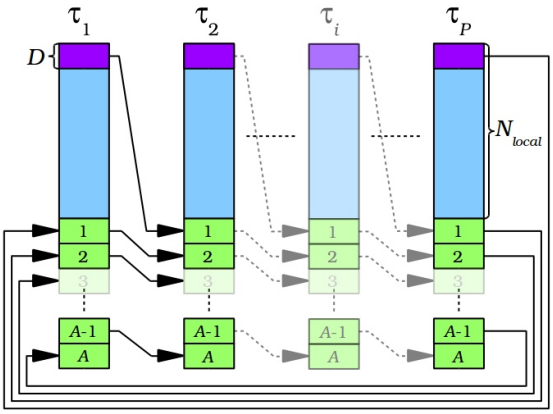


Figure 4.18: Particle exchanging by passing particles around the ring topology. The purple boxes represent the top D particles of task $\tau_i,\ i = 1, ..., P$, while the green boxes – exchanged particles from a neighbor.

Algorithm 8 Algorithm for particle exchange function

function $\{\mathbf{x}^*_{k,j}, w^*_{k,j}\}_{j=1,...,N_{new}} =$ xchg$(\{\mathbf{x}_{k,i}, w_{k,i}\}_{i=1,...,N_{local}}, D, A)$
Input : Local particle set of the current iteration, the amount of particles to exchange and the amount of neighbors.
Output : New set of particles

1: Partially sort local particles: $\{\mathbf{x}_{k,i}, w_{k,i}\} =$ psort$(\{\mathbf{x}_{k,i}, w_{k,i}\}, D)$
2: **for** $i = 1 : N_{local} - D_{overlap}$ **do**
3: Copy particles: $\{\mathbf{x}^*_{k,i}, w^*_{k,i}\} = \{\mathbf{x}_{k,i}, w_{k,i}\}$
4: **end for**
5: Let $p \in \mathbb{N}_{<P_{tot}}$ be the processor this task is executed on
6: Let $q = p - 1$ when $p > 1$, else P, be a previous neighbor processor
7: Send top D particles: `write_fifo`$(\{\mathbf{x}_{k,i}, w_{k,i}\}_{i=1,...,D},\ \text{mod}(p+1, P))$
8: Start from overlap location: $a = N_{local} - D_{overlap} + 1$
9: **for** $i = 1 : A - 1$ **do**
10: Read particles from previous neighbor:
$\{\mathbf{x}^*_{k,j}, w^*_{k,j}\}_{j=a+(i-1)*D,...,a+i*D} =$ `read_fifo`(q)
11: Forward particles to closest neighbor:
`write_fifo`$(\{\mathbf{x}^*_{k,j}, w^*_{k,j}\}_{j=a+(i-1)*D,...,a+i*D},\ \text{mod}(p+1, P))$
12: **end for**
13: Read final set of particles from previous neighbor:
$\{\mathbf{x}^*_{k,j}, w^*_{k,j}\}_{j=a+(A-1)*D,...,a+A*D} =$ `read_fifo`(q)

From Fig. 4.16, one may notice that only one FIFO data channel from a PF task $\tau_i,\ i = 1, ..., P$, is formed only to its closest neighbor[1], which contradicts the intuitive notion that a task is connected to all of its "exchange" neighbors. One of the reasons for this arrangement, is that a processor may run out of FIFO memory when allocating space for the channels. The other reason is the additional time overhead introduced, due to the larger distance to further neighbors. Thus, during particle exchange, a PF task not only sends its own best particle set to the closest neighbor, but also sends the received sets from its previous neighbor(s). In essence, each task "propagates" sets of exchanged particles around through its closest neighbor, until each task receives all A batches of D particles. This scheme is illustrated in Fig. 4.18, where arrows represent particle exchanges. It is assumed that an appending exchange strategy is used, although any other strategy is possible. Each of the exchanges are preformed by its respective source task $\tau_i,\ i = 1, ..., P$, in a top-to-down sequence, through a single FIFO channel.

The the operation of the exchange algorithm is thus described in 8. This description takes into account any exchange strategy, by setting the $D_{overlap}$ parameter accordingly. For example, for $D_{overlap} = 0$, an appending strategy is used.

[1] An exception to this is task τ_P, which takes a bit of a longer route due to the NoC

Chapter 5

Analysis and experimental results

So far, the internals of the HOG detector and PPF implementations have been described and discussed in detail. This chapter will focus on their analysis and evaluation of the two algorithms. In particular, the functional and temporal behavior of both implementations will be evaluated. Specific algorithm performance criteria like detection miss rate, hardware resource usage and tracking accuracy are also discussed.

5.1 HOG-SVM detector evaluation

The HOG-SVM detector is difficult to analyze in hardware on *Starburst*. Since the detection performance of the HOG-SVM algorithm has already been extensively studied and documented in various works, such as [1, 3, 8, 11, 4], this section will focus on findings related to the previously described hardware implementation, such as hardware resource usage and simulation. Therefore, the results presented here don't focus on the detection accuracy, but rather on more specific features of the hardware implementation. Yet, detection results, computed from the simulation of the hardware implementation are presented, demonstrating its operation.

5.1.1 Test setup and parameters

Here, the same parameters as discussed in 2.3 are used to configure and synthesize the hardware implementation. The SVM classifier is trained from the INRIA person data set, using a multi-core SVM training library called `LIBLINEAR` [39, 40]. A portion of the positive image data set is used to simulate the hardware implementation using ModelSim.

The test bench used to simulate the design is quite straightforward: a grayscale input image is fed into the FIFO buffer of the HOG detector, with a clock frequency of 12.5 MHz. Pixels are streamed into the FIFO, following the VGA protocol discussed before. The CMOS image sensor used in the real setup is the OV7670 camera module. Thus, timing characteristics of the device have been incorporated into the simulation. The idea of the simulation is not

only to compute the detection results for a given image, but also to determine optimal parameters for the implementation, such as the minimum pipe-line clock frequency and the amount of partial dot-products executed in parallel, sufficient to achieve maximum throughput. Additionally, it's important to determine the minimum buffer capacity of the FIFO, given the selected frequency and amount of dot-products. throughput constraints. Additionally, one is also interested in the hardware resource usage on the FPGA, required to satisfy these constraints. Therefore, the synthesis results for a given image size and amount of partial dot-products are also presented.

5.1.2 Simulation results

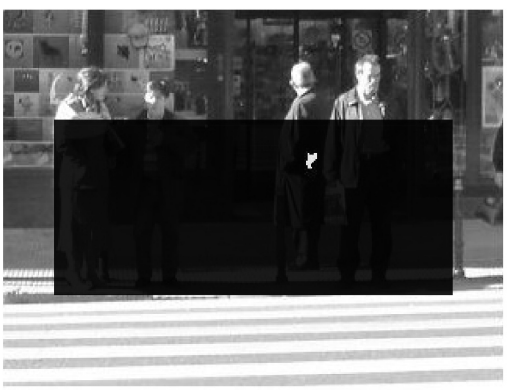
The simulation has been performed for a 320x240 image, scaled down three times by a factor of $\sigma = 1.2^k, k = 0, ..., 2$. The scaling of an image, allows detection of objects with a different size. A test image from the positive INRIA training data was used to compute the HOG detection output images for the selected scales, illustrated in figures 5.1a, 5.1b and 5.1c. The black and white detection output image has been superimposed on top of the gray-scale input image, based on the center position of each detection window. The white pixels of the detection image correspond to windows, where a person has been detected. One can observe, that the detector can successfully detect people in a given image. However, there are also some missed detections. More thorough testing and parameter tweaking is needed, to grade and improve the overall accuracy of the detector. Additionally, non-maximum suppression should also be applied, to filter out the redundant detection pixels.

5.1.3 Optimal parameters

A great advantage of the hardware implementation of the HOG detector, is the fact that its execution is completely deterministic. The feature extraction pipeline takes exactly one clock cycle to compute a new block feature vector, for each new pixel received from the camera, while the SVM classification module needs exactly one clock cycle to perform a partial dot-product with the newly computed block vector, for a specific window. Of course, since the amount of windows that share the same block varies, as the image frame is scanned further and further, the amount of clock cycles required to complete full detection is much higher than the amount of pixels in the image.

This prompted the separation of the pipe-line into two clock domains – the CMOS image sensor's pixel clock domain, and a much higher pipeline clock frequency domain. A dual-clock FIFO was thus included as a bridge between both domains, and as a buffer, to hold the pixels from a received frame for enough time, until the classification process is completed for all detection windows, and all extracted block feature vectors. An additional parameter was also included in the implementation, allowing parallel processing multiple partial dot-products, for multiple windows that share the same block vector. In other words, the parameter is used to control the amount of multiple windows processed at the same time.

In summary, there are three important parameters one may wish to determine to achieve maximum real-time performance: the minimum clock frequency of the pipeline, the maximum FIFO capacity required for that frequency and the



(a) $\sigma = 1$

(b) $\sigma = 1.2$

(c) $\sigma = 1.44$

Figure 5.1: Input image at different down-scale factors, with the detection output images superimposed on top.

amount of dot-product parallelism. The latter two achieve a similar goal, but with a trade-off – more partial dot-products results in lower frequency, at the expanse of more DSP hardware, and vice versa. A great advantage of operating at lower frequencies, is that the clock constraints can be more relaxed during synthesis, and perhaps lower power consumption during operation.

Here, results from simulations are presented, that try to find a minimum pipeline clock frequency, required to achieve maximum throughput, at a varying amount of partial dot-products. Per definition, maximum throughput is achieved, as long as one image frame can be immediately be processed after a previous frame. Therefore, to avoid buffer overflow, the amount of pixels stored in the FIFO should be 0 at the end of each frame.

Using ModelSim, the HOG detector is simulated for different pipeline clock frequencies and partial dot-products, starting from a low frequency of 100 MHz and 1 dot-product, and stepped up by 5 MHz and 1 dot respectively. An image resolution of 320x240 pixels is assumed and up to 6 PDPs are considered. The results are summarized in table bla. In the first column, the minimum frequency required to achieve maximum throughput is recorded, for a specified amount

Clock freq. (MHz)	PDP amount	Capacity (pixels)
205	1	7870
145	2	3434
115	3	3444
80	4	4958
70	5	9014
70	6	7929

Table 5.1: Optimal configuration parameters of the HOG detector, given a 320x240 frame image resolution

of partial dot-products in parallel, presented in the second column. The third column shows the minimum FIFO capacity required for the given frequency and PDP amount. Any clock frequency above the minimum threshold will result in lower buffer capacity, while clock frequencies below – bigger buffer capacity and lower throughput.[1] As one might expect, increasing the amount of partial dot-products executed in parallel per cycle, does decrease the minimum required pipeline clock frequency. But after certain point, the reduction becomes less effective.

It is important to note, that while these results provide certain assurance about the real-time performance of the implementation, they do not provide guarantees, as hardware can also experience non-deterministic behavior on the semiconductor level of the FPGA. This is especially true for designs that incorporate multiple clock-domains, due to metastability conditions of digital logic. Even though a design may be well optimized for very good performance, there is always a chance of failure, and hardware is necessary, that can detect and recover the system from such failures.

5.1.4 Hardware resource usage

Finally, the evaluated HOG detector hardware has been individually synthesized for a Xilinx Virtex-6 240T FPGA. Given the optimal parameters derived earlier, the hardware has been synthesized for image resolution of 320x240, and the amount of PDPs ranging from 1 to 6. The resource usage has been summarized in table 5.2. It is more than obvious, that the amount of logic resources scales approximately linearly with the amount partial dot-products. However, the usage of BRAMs is not as efficient. Ideally, the BRAMs used by the HOG detector should stay the same, for any PDP configuration. However, due to data granularity, most of these BRAMs are not fully utilized. Future work should focus on improving this issue as much as possible.

5.2 PPF evaluation

5.2.1 Test setup and parameters

Two test setups of the PPF are evaluated – the example system from chapter 3 and a color based visual tracking PPF, each first executed on a standard laptop

[1] In this case, one needs to wait until all detection windows are processed, before starting a new frame.

PDPs	Registers	LUTs	DSP slices	BRAMs (36E1)	BRAMs (18E1)
1	4064 (1%)	8887 (5%)	36 (4%)	65 (15%)	32 (3%)
2	4762 (1%)	10371 (6%)	72 (9%)	72 (17%)	34 (4%)
3	5483 (1%)	12558 (8%)	108 (14%)	80 (19%)	91 (10%)
4	6108 (1%)	13943 (9%)	144 (18%)	88 (21%)	36 (4%)
5	6821 (1%)	18381 (12%)	180 (23%)	110 (26%)	37 (4%)
6	7513 (1%)	22283 (14%)	216 (28%)	90 (21%)	122 (14%)

Table 5.2: HOG detector resource usage for a Virtex-6 240T FPGA, given 320x240 frame resolution

PC, and then on Starburst.

The first setup consists of a task, which simulates the system to be estimated, and generates observation data for the PF tasks. The actual state and estimated state from the PPF are then recorded for further analysis. The purpose of this setup is to study the temporal behavior of the PPF and its estimation accuracy with respect to various parameters, since test data is easy to generate and compare. Additionally, as far as temporal analysis is concerned, the difference between the first and second setup is only in the execution times of the prediction and update steps. Thus, as long as the prediction and update steps are guaranteed to be deterministic, with a constant execution time per iteration, the validity of temporal analysis should remain unaffected.

The visual PPF utilizes an AR motion model in its prediction step, and incorporates color histograms in the update step. A test sequence of images are used, where a common household object is manually detected and its color histogram extracted in the first sequence, and then tracked in upcoming ones. The sequences are made, such that they can be repeatedly "looped" in an infinite series. This setup is ideal for evaluating the tracking accuracy of the filter, with respect to parameters of the motion and observation models.

The software is written in C and compiled on both platforms using GCC, and then evaluated using MATLAB. Since the code is functionally equivalent on both platforms, it can be safely assumed, that the same filtering results will be expected on both platforms. Therefore, the PC implementation is mainly used to study and analyze the tracking performance of the PPF, with respect to various parameter changes, while the implementation on *Starburst* – its temporal performance. This is done, simply because evaluation on *Starburst* takes a lot of time to simulate and estimate a system, while proving nothing new about the accuracy of the filter, with respect to its PC counterpart. On the other hand, evaluating the temporal performance on the PC implementation will not result into meaningful or useful information. Still, to solidify the assumption made above, a couple of tests are performed on both platforms to compare the estimation accuracy, proving that the functional behavior of both implementations is indeed the same.

5.2.2 PC evaluation results

As discussed previously, the PC implementation is used to evaluate and study the estimation performance of the PPF, with respect to the total number of particles, the number of PF tasks, the exchange particle amount D and the

number of exchanges A (or equivalently, the number of exchange neighbors). The estimation accuracy is measured in terms of the RMS error (RMSE), defined as

$$RMSE = \sqrt{\frac{1}{T_{sim}} \sum_{k=1}^{T_{sim}} (\mathbf{x}_k - \hat{\mathbf{x}}_k)^2}, \tag{5.1}$$

where T_{sim} is the number of simulated iterations of the PF. Here, only the results from the first test setup are presented. The number of iterations per run is fixed to $T_{sim} = 150$. Since the error tends to vary a lot with each run of the setup, the PPF software is executed several times for the same parameter values. The mean of the RMSE values computed from each run, is then used as "global" estimation RMSE. The number of runs per parameter change is fixed to 20.

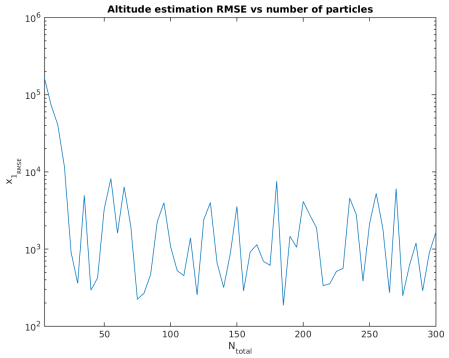
The first set of measurements tries to find an upper bound on the amount of particles needed, to effectively estimate the state. Below this bound, it is expected the estimation accuracy will start to degrade rapidly. The measurements are done on a single PF task, without any particle exchanges. One can see from figures 5.2a to 5.2c, that the RMSE of the PF tends to settle after around 50 particles. Below this threshold, the error starts to grow rapidly. This threshold will be used as a reference for the rest of the tests.

The next test set, investigates the effect of splitting the PF between multiple tasks on the estimation result. The tasks are executed on separate POSIX threads, to speed up the execution of the PPF. Here, a similar procedure is performed as before for a several number of task amounts. Again, no particle exchanges are enabled for this test set. Figures 5.2a to 5.2c show the plots of the RMSE for each state variable, and one can clearly see that the estimation accuracy degrades quite severely for larger amounts of tasks.

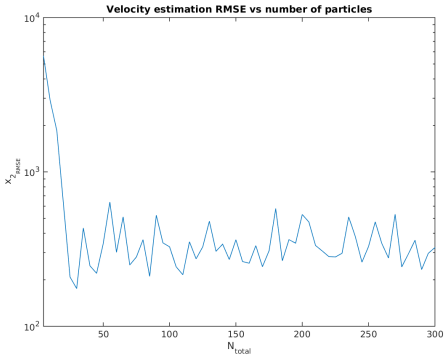
The final set of tests is intended to show the influence of particle exchanges on the RMSE. Here, the total number of particles is fixed to $N_{total} = 50$, while the exchange neighbors amount A and exchanged particle amount D are varied. The results are illustrate in figures 5.4a through 5.4i. As expected, exchanging particles between tasks greatly improves the estimation accuracy. Even one particle exchanged with one neighboring task already shows an improvement. However, it seems that exchanging particles with more neighbors doesn't improve the result any further for this particular system. This probably has to do with the fact that exchanging only a couple of particles is already sufficient.

5.2.3 Starburst evaluation results

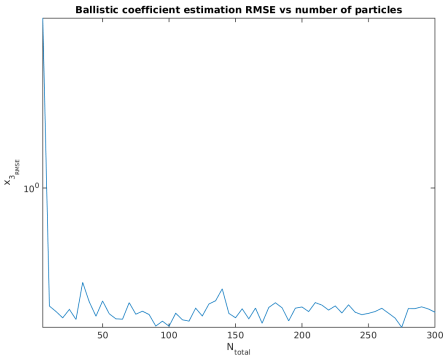
The software implementation of the PPF on *Starburst* is evaluated in a similar fashion as before. The hardware of *Starburst* is configured to accommodate 32 MicroBlaze processing cores on the FPGA, running at 100 MHz each. One core runs the Linux kernel and a file system, in order to establish a connection between the platform, and an evaluation PC. The software of the non-Linux cores, is therefore loaded from the evaluation PC through a standard Ethernet connection. The need to recompile and upload the software, based on any parameter changes, makes the evaluation of the PPF hard to execute. Thus, because of the slow run-time performance and the above mentioned communication constraints with the development platform, execution of the algorithm is



(a) Altitude



(b) Velocity



(c) Ballistic coefficient

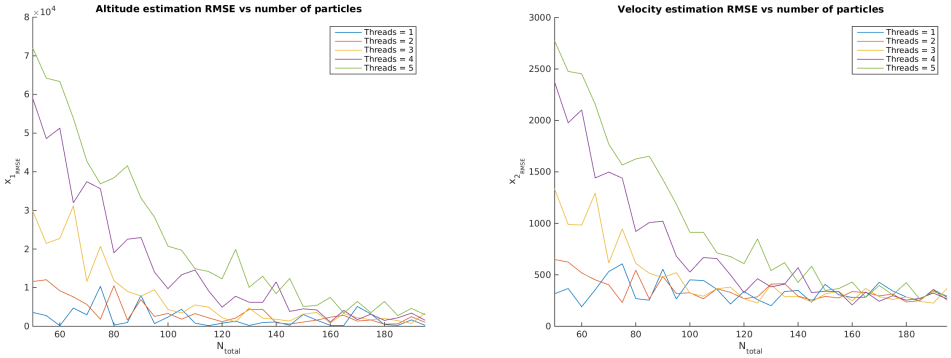
Figure 5.2: RMSE of each state variable.

restricted to one single run. A consequence of this, is that the validity of the measurements done on the estimation accuracy of the filter, are not to be fully trusted, because of the huge variance of the estimation error. Regardless, the idea is to focus more on measurements of the average worst case execution times (WCET) of PPF and its critical sections, in order to determine the potential bottleneck areas and the real-time performance.

The first set of measurements follow a similar methodology as before, where the RMSE is plotted vs. a varying amount of particles, for a given number of PF processors. In addition, the WCET of a PF iteration is also measured by the global estimation processor, by taking difference between the time measured before receiving local estimates from all PF cores, and after, for each iteration. The maximum of these time differences is taken as the WCET of the PPF, such that

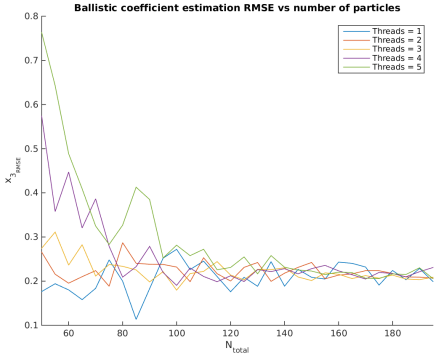
$$WCET = \max_k \Delta T_{PF_k},$$

where k is an iteration number, and ΔT_{PF_k} – the time difference. A total of 150 iterations are simulated, with particle exchanges disabled. The results can been seen in figures 5.5a-5.5c and 5.5d. In terms of accuracy, a similar outcome is observed as the PC implementation, although the picture isn't that clear



(a) Altitude

(b) Velocity

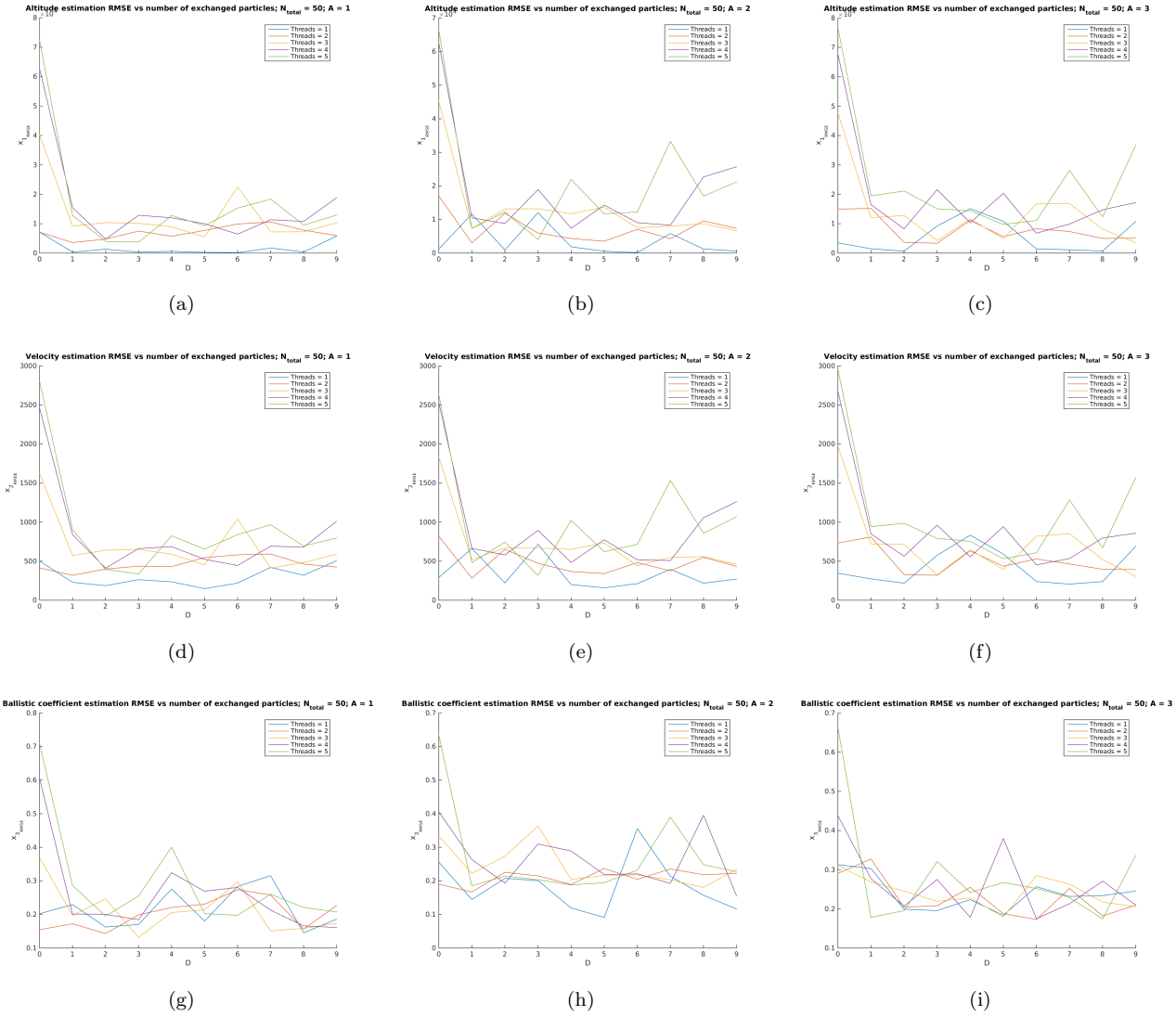


(c) Ballistic coefficient

Figure 5.3: RMSE of each state variable vs. number of particles per task amount.

for the ballistic coefficient, since the error is already quite small. But overall, accuracy is seen to deteriorate with increasing amount of processors. What is an important find, is the obvious linear relationship of the execution time with respect to the number of particles. This is one of the first signs, that the filtering algorithm behaves deterministically in real-time, which is what this research is aiming for.

Another set of measurements is aimed at the execution times of the individual steps of the particle filter. The results are illustrated in figures 5.6a and 5.6b. Here, the number of processors is fixed to $P = 5$, and the total amount of particles $N_{total} = 100$, with the execution time plotted for each iteration. As expected, the prediction and update steps prove to be the biggest bottleneck in the algorithm. Additionally, the execution of each iteration tends to consume relatively constant average amount of time, with some jitter. The main contributor to this jitter, is the normal distribution sampling algorithm. An interesting observation, is the similarity between the execution times for each processor. This finding suggests that a shared random number generator is utilized by all the cores. A thorough analysis is needed to confirm this however,



(a) (b) (c)

(d) (e) (f)

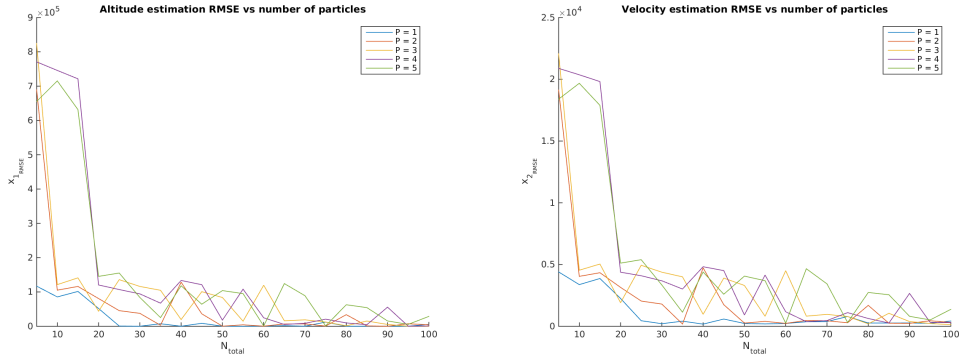
(g) (h) (i)

Figure 5.4: RMSE of each state variable vs. number of exchanged particles per task amount and number of exchanges.

left as future work. In any case, a hardware accelerator for each core of a more deterministic random sample generation algorithm is recommended.

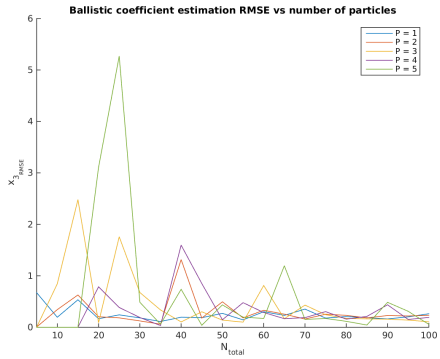
To show the improvement due to parallelization of the filter, the WCET is measured for a varying amount of processors, starting from 1 and ending with 25 processors. The measurement is performed for fixed total amount of particles. The results are illustrated in 5.7a, showing an inversely proportional trend, with the execution time being equal to:

$$WCET_{PPF} = \Delta T^*_{PF} \frac{N_{total}}{P} \tag{5.2}$$

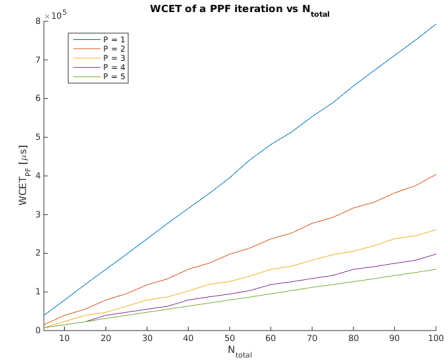


(a) Altitude

(b) Velocity



(c) Ballistic coefficient

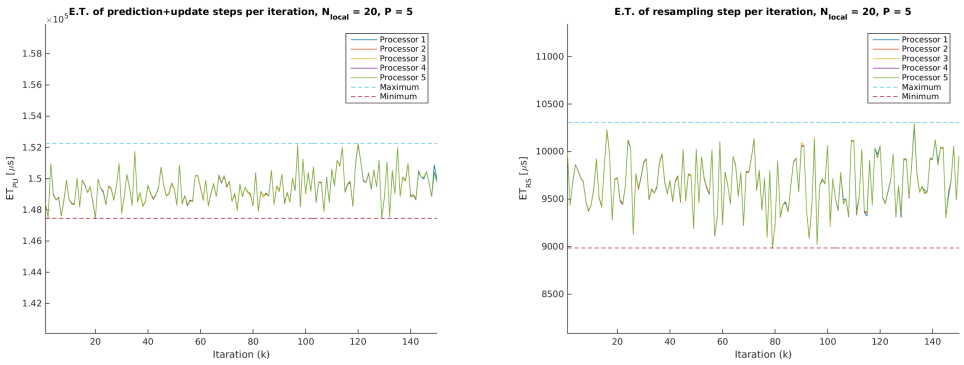


(d) WCET

Figure 5.5: RMSE and WCET vs. number of particles per processor amount.

where ΔT^*_{PF} is the maximum execution time of a particle filter iteration per particle, given $N_{total} = 1$ and $P = 1$. This suggests, that the throughput is linearly dependent on the number of processors, which is the ideal situation one is looking for, as one can observe in 5.7b.

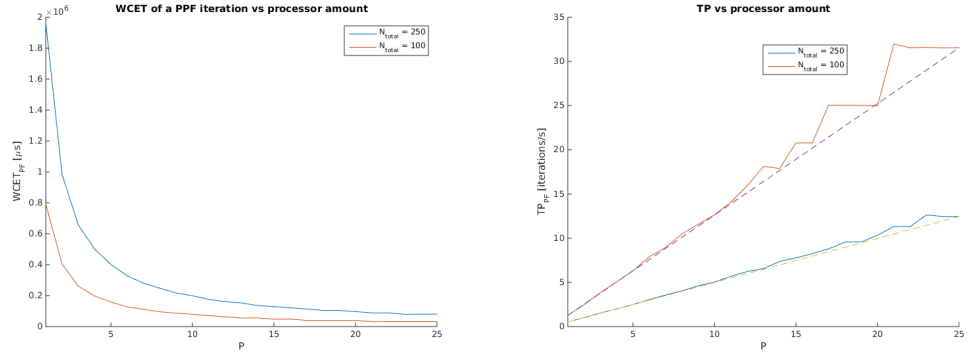
The story becomes quite different, when particle exchanges are introduced. To show the influence of particle exchanges, one may refer to figures 5.8a – 5.8c. Figures 5.8a and 5.8b show the time difference before and after completing a full exchange, plotted for each iteration, while 5.8c – the average time difference of the first processor, with respect to varying amount of exchanged particles and neighboring cores. One can clearly observe, that exchanging more particles with more neighbors affects the execution time of the exchange step. Yet, the least exchange time overhead is experienced by the last processing core, staying relatively constant throughout the execution of the PPF, compared to the rest of the processors, which tend to experience particularly large amount of varying communication overhead. This effect becomes particularly dominant for increasing amount of neighbors, most likely suggesting that all of the other processors wait on the last processor, where the particles travel around the ring NoC, back to the first one.



(a) Prediction and update steps

(b) Resampling step

Figure 5.6: Execution time of the exchange step, vs .



(a) WCET

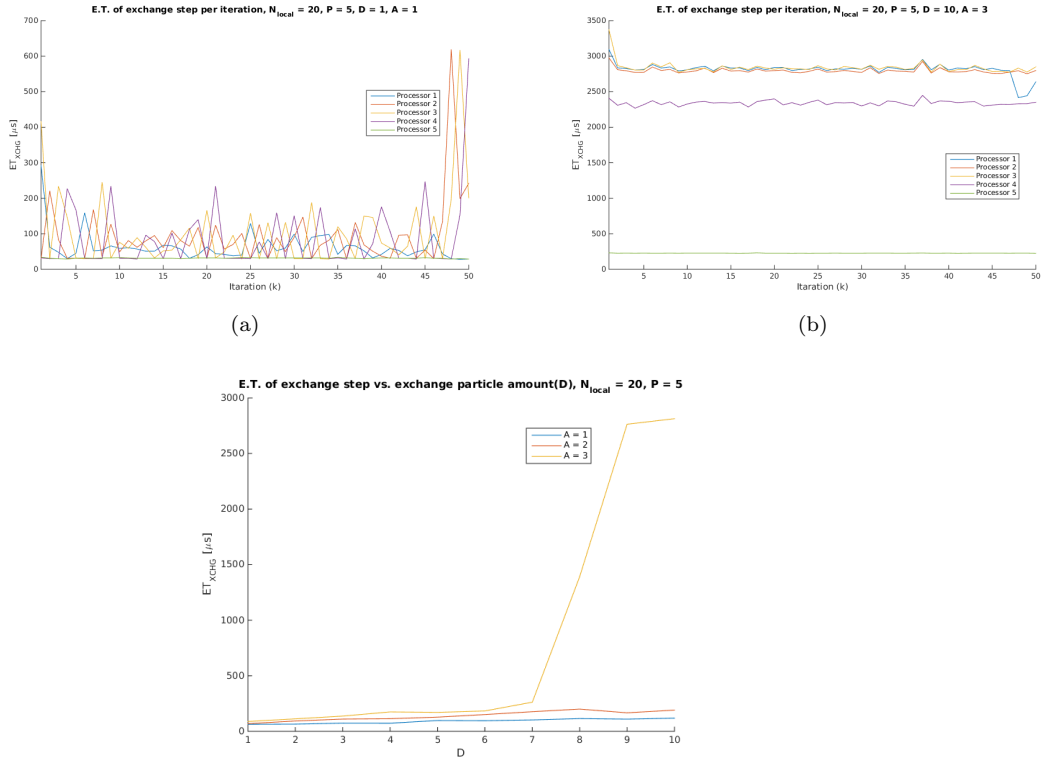
(b) Throughput

Figure 5.7: WCET and TP vs. number of processors

The final set of measurements shows how particle exchanges affect the WCET of a PPF iteration, illustrated in 5.9. One can see, that while particle exchanges also introduce jitter, its affect is superficial on the overall execution time of the PPF. However, future developments of the implementation should aim to reduce this jitter, by e.g. reducing the amount of processors to an optimal value and therefore – the communication overhead. Otherwise, this jitter would greatly contribute to the throughput of the algorithm, if hardware acceleration comes into play.

5.2.4 Visual tracking performance

The particle filter has also been evaluated in terms of its visual tracking performance. The software implementation is identical to the previously evaluated one, differing only in the prediction and update steps. Since the only expected difference should be in the execution times of the prediction and update steps, the temporal behavior of the visual PPF tracker is not evaluated on *Starburst*,



(a) (b)

(c)

Figure 5.8: Typical measured execution time of the exchange step.

since any measurements in that regard don't contribute with anything new.

The prediction step makes use of the AR motion model, described by eq. 3.19, which is a relatively straightforward. The update step makes use of color histograms to estimate the likelihoods of each particle. A template image is used to extract a reference color histogram, which is then used to compute the particle weights through Bhattacharyya's distance, defined in eq. 3.21. The total color histogram is composed of individual 16 bin histograms for each channel. Particle histograms are extracted with a similar method as the reference histogram.

To test the tracking performance of the filter, a sequence of still images of an orange in a cluttered environment is filmed. The positions of the orange in each frame, are set in a predictable circular pattern, to allow easy looping of the whole sequence. Some of the frames, demonstrating the tracking procedure can be seen in figures 5.10a to 5.10d. The green circles represent the particles, while the estimated position of the orange is at the center of the red bounding box. The reference template image can be seen in 5.10e.

To test the visual tracking performance of the filter, 80 iterations were simulated of the filter, where the filmed sequence of 20 images is looped 4 times. The trajectories of the orange have been plotted and illustrated in fig. 5.11a and 5.11b. As it turned out, it was relatively easy to track such a generic object, based on colors only. However, there have also been a lot of runs of the PPF

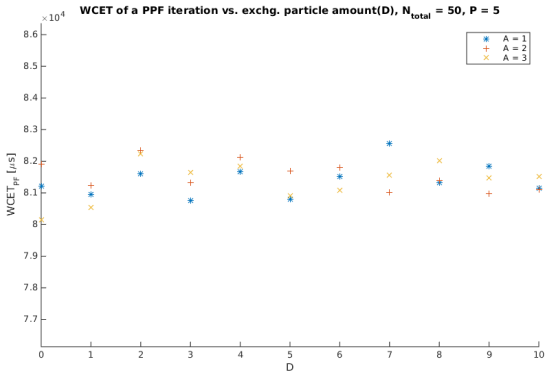


Figure 5.9: WCET of a PPF iteration vs. exchanged particle and neighbor amount.

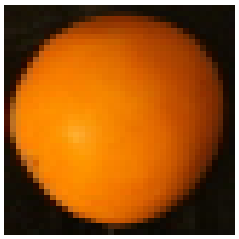
(not documented), where the algorithm misses its target. There are a lot of factors and parameters that need to be explored, such as the covariance matrices of the AR model, bounding box size, histogram length, etc, to find the most optimal configuration of the particle filter. This process however will be left as future work.

(a)

(b)

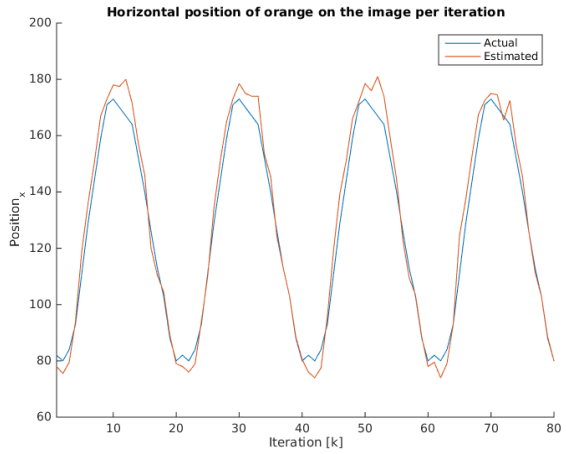
(c)

(d)

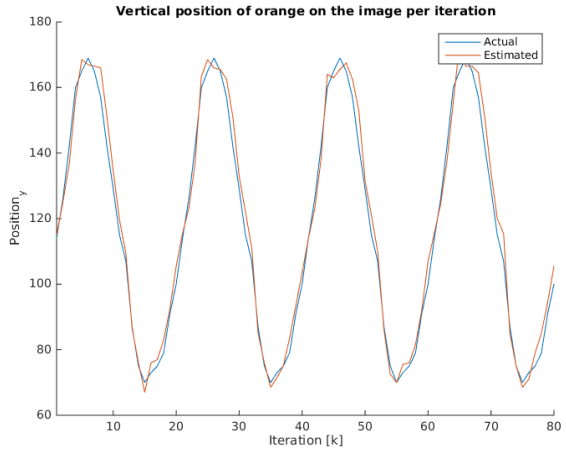


(e) The template image

Figure 5.10: Some example frames of the PF object tracking process and the reference frame image.



(a)



(b)

Figure 5.11: The real and estimated x and y trajectories of the orange, over a 80 of iterations.

Chapter 6

Conclusions

The objectives put forward by this research were to investigate, implement and evaluate two computationally expensive computer vision algorithms in the form of HOG object detection and Particle filter tracking, to determine the suitability of *Starburst* for real-time computer vision algorithms. As a result, a unique hardware implementation of the HOG detector has been designed, accompanied by a multi-processor software implementation of the Particle filter.

At this point, the results are not fully conclusive to answer the research questions convincingly. What was made clear however, is that*Starburst* is not suitable for software only implementations of object detection and tracking, without the aid hardware acceleration, or complete mapping of the algorithms in hardware.

6.1 HOG detector

First, the HOG feature extraction and classification technique presented in [1], has been decomposed into its basic building blocks to derive a straightforward and deterministic image processing pipeline. However, the pipeline proved too much of a computational burden to be mapped in software on *Starburst*, so it was directly implemented in hardware, capable of achieving single cycle processing per pixel. Essentially, the hardware detector can directly process incoming video frames from a camera sensor, and send the detection results to a processing core, allowing the software to concentrate on tracking.

The biggest challenge associated with the hardware detector, was handling the high dimensionality of the HOG feature vector. It introduced two major problems - high amount of multiplications and large storage demands. Fortunately, an intuitive data dependency was exploited to drastically reduce this amount and the associated storage requirements.

The best feature of the hardware implementation, is its scalability and reconfigurability. A designer can easily exploit the trade-off between processing clock frequency and area, as the user increases the amount of partial dot products. Because of the deterministic nature of the implementation, simulations can indeed provide guarantees about the maximum throughput achievable in theory. RT analysis tools can be employed to determine the optimal parameters of the hardware, such that the real-time throughput constraints can be satisfied.

However, one can never trust simulation results entirely, as a hardware design can always contain flaws and various sources of non-deterministic behavior.

6.2 Parallel particle filters

Perhaps the main highlight of this research, was the analysis and evaluation of the parallel Particle filter. As it turned out, it wasn't sufficient to directly split the workload between many tasks, since the estimation (or tracking) accuracy deteriorates significantly. A clever solution proposed in [27, 28] proved quite effective, where a communication topology was utilized to take full advantage of the the multi-core architecture of *Starburst*.

First, the estimation accuracy of the algorithm was studied on a standard PC platform, showing the effect of varying the amount of particles and threads. As expected, distributing the filter among more tasks indeed deteriorates its estimation accuracy, but allowing the tasks to exchange some particles can greatly improve it, at the cost of a small communication time overhead.

Next, the temporal real-time behavior was evaluated on *Starburst*. It was shown that the filter indeed behaves deterministically, while exhibiting a linear relationship between WCET and the number of particles. Additionally, the expected bottlenecks experienced in the prediction and update steps have been confirmed, identifying key targets for hardware acceleration.

In any case however, it has been shown that increasing the amount of processors drastically improves the throughput of the filter, even without any hardware acceleration involved, but at the hardware cost of many processors involved. But most importantly, exchanges of particles were observed to not pose as a critical bottleneck, therefore the advantages gained through distribution of the filter across many processors are still preserved, with a high estimation accuracy.

Finally, the filter was adapted to the problem of visual object tracking, by means of color features. Since the amount of non-linear operations involved in computing the particle weights proved too much to handle by the simple Microblaze processors, additional hardware accelerators should be added to rectify this bottleneck.

Results show that the particle filter algorithm can indeed be mapped deterministically on *Starburst*, such that it can sufficiently satisfy the real-time constraints for most practical cases. Guarantees however can not be made about its real-time performance.

6.3 Future Work

Although both algorithms have been shown to work independently, they are yet to be integrated together, to form the complete visual object tracking system. The first objective of future development, would involve getting both the detector and tracker to work together and in synergy. In addition, future work will try to implement error detection and failure recovery mechanisms, to ensure the safe behavior of the system.

An optional goal, would be to move the system away from the expensive Virtex 6, to a range of lower end FPGAs, such as the Artix 7. Zynq 7000, or

an Altera Cyclone V. This would significantly lower the cost and power usage of the system, while increasing its accessibility to a wider range of applications.

Besides this, there are a couple of other goals related to each algorithm independently, which one should look forward to in future work.

6.3.1 HOG detector

The very nature of the object detection algorithm prevents straightforward testing of the hardware implementation. As a consequence, guarantees cannot be made about the detection accuracy from software simulations alone. This is a drawback, since this research has mostly relied on the successful evaluation and analysis of the HOG detector by its respective authors and other third parties. Thus, having a dedicated real world test setup would be most beneficial and a logical step forward to improve this research.

Additionally, in its current state, the implementation still has a lot of room for improvement with respect to hardware resource usage. By attempting to reduce the amount of hardware resources used even further, the author expects a bigger performance boost, more reliable operation and more space to fit more object detectors. This is useful for multiple scale object detection, or for future work to explore the deformable parts model presented in [2, 3].

Finally, non-maximal suppression is yet to be implemented. The purpose of this operation is to get rid of redundant detections computed by the detector. Although the process is quite straightforward to implement in software, it could potentially bottleneck the processing pipe-line. Thus future versions of the HOG detector would also incorporate non-maximal suppression in hardware.

6.3.2 Parallel Particle Filter

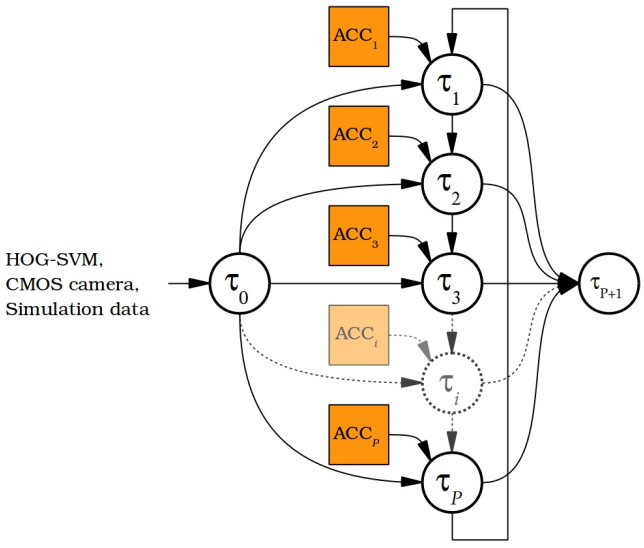


Figure 6.1: Parallel Particle filter task graph and communication topology

The Particle filter algorithm has been successfully mapped on the *Starburst* architecture. Due to limited development time however, it was not possible to

explore different approaches and possibilities for hardware acceleration. This has left the algorithm in a state, where it still lacks the required speed and throughput to satisfy the demanding requirements of safety-critical computer vision applications. Therefore, future efforts will be concentrated on utilizing the hardware accelerator architecture of *Starburst* to rectify the performance, deteriorated by the bottlenecks identified earlier. An illustration of a proposal task topology with accelerators can be seen in 6.1. It is expected that the processors will communicate with the accelerators through the ring NoC. Particles will most likely be directly produced from the accelerators, while the task of flow control, particle exchanges and resampling will be left to the processors.

A final consideration for future work, would be to try incorporating local features to the PF based object tracker, instead of global features (such as color). SIFT[5] - an algorithm for extracting and describing interest regions in an image - is a good example of local features, unique to a specific object. Tracking an object's local features directly decreases the chances of loosing the target, due to occlusions or appearance of similar objects. Incorporating SIFT-like object descriptors would be thus very useful to the system.

6.3.3 Real-time Analysis

Unfortunately, it wasn't possible to also analyze the algorithm implementations using RT tools. Future work would focus on the validation and analysis of Data-flow graph based RT models of the HOG detector pipeline and PPF task topology. The motivation of using RT analysis tools, is to more easily explore all parameter possibilities of each implementation, and determine the theoretical bounds on the throughput, prior to actual deployment of the system.

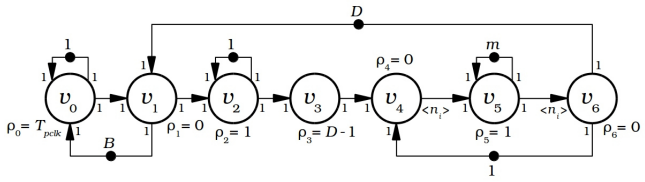


Figure 6.2: HOG detector RT CSDF model.

A model proposal for the HOG hardware pipeline, based on CSDF graphs, can be seen in Fig. 6.2. Here, T_{pclk} represents the average pixel arrival period of the CMOS sensor, assuming that the HOG pipeline clock is synchronous to the pixel clock and pixels; m determines the amount of partial dot-products performed in parallel, by duplicating the SVM accumulation module; D is the clock cycle latency, due to pipelining registers; and B is the dual-clocked FIFO capacity. The idea of this model, is to determine the minimum buffer capacity B, constrained by m and T_{pclk}. In other words, one would like to first find an optimal trade-off between T_{pclk} and m, to determine the maximum buffer capacity.

For the PPF, an SDF model is proposed, illustrated in Fig. 6.3. The model represents the task graph topology described in section 4.3, and parameters A and P refer to the PF exchange neighbor amount and number of processors respectively. T_{XCHG} is the execution time of a single particle batch exchange,

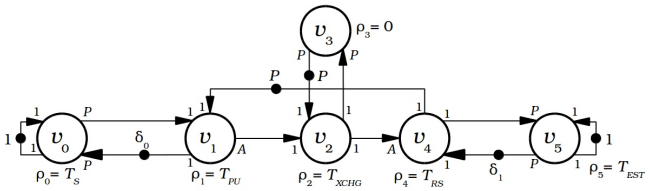


Figure 6.3: Parallel Particle filter SDF model.

while T_{PU} and T_{RS} are the execution times of the prediction+update and re-sampling steps, respectively. The ET of any intermediate operations of the exchange step, according to alg. 8 such as sorting, are also included in T_{PU}. These times can be measured off-line, following a similar as the one during the evaluation of the PPF implementation. The idea of this model is to both find an optimal RT schedule for each of the PF tasks, and also determine the buffer capacities δ_0 and δ_1.

One must keep in mind, that both of these models are just proposals, and are yet to be validated and analyzed. This will perhaps be on highest priority of any future work.

Appendices

Appendix A

Function definitions

A.1 atan2 function

The atan2(y, x) computes the arctangent of two arguments y and x, based on the quadrant of the computed angle. It is defined as

$$\text{atan2}(y,x) = \begin{cases} \arctan\frac{y}{x} & \text{when } x > 0, \\ \arctan\frac{y}{x} + \pi & \text{when } x < 0 \text{ and } y \geq 0, \\ \arctan\frac{y}{x} - \pi & \text{when } x < 0 \text{ and } y < 0, \\ +\frac{pi}{2} & \text{when } x = 0 \text{ and } y > 0, \\ -\frac{pi}{2} & \text{when } x = 0 \text{ and } y < 0, \\ \text{undefined} & \text{when } x = 0 \text{ and } y = 0. \end{cases}$$

A.2 mod function

The mod(a, b) function computes the remainder, when dividing two integers a and b, defined as

$$\text{mod}(a,b) = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

Appendix B

Hardware block diagram symbols

In the implementation chapter, hardware block diagrams were introduced to describe the HOG detector implementation. The purpose of these block diagrams was to give an idea of how the implementation is formed, without exposing too much details. Therefore, the reader might find the symbols used confusing. Here, a list of basic symbols and a brief description is provided in Fig. B.1.

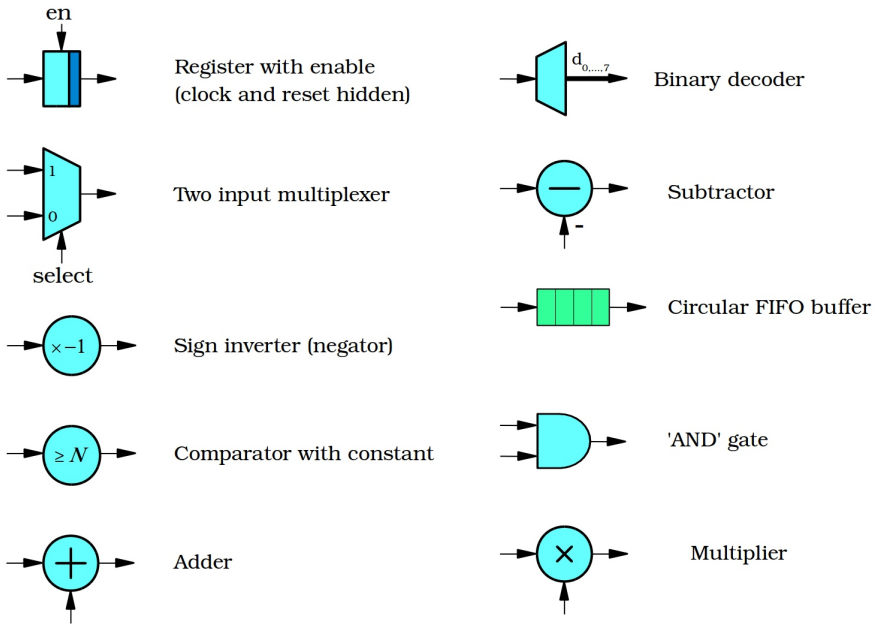


Figure B.1: Basic hardware block diagram symbols

Bibliography

[1] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 886–893, IEEE, 2005.

[2] R. B. Girshick, P. F. Felzenszwalb, and D. McAllester, "Discriminatively trained deformable part models, release 5." http://people.cs.uchicago.edu/ rbg/latent-release5/.

[3] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part based models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.

[4] H. Bristow and S. Lucey, "Why do linear svms trained on HOG features perform so well?," *CoRR*, vol. abs/1406.2419, 2014.

[5] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2, pp. 1150–1157, Ieee, 1999.

[6] ARM® Holdings plc, *Cortex™-A9 NEON™ Media Processing Engine, Technical Reference Manual*.

[7] Intel®, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

[8] T. Wilson, M. Glatz, and M. Hödlmoser, "Pedestrian detection implemented on a fixed-point parallel architecture," in *Consumer Electronics, 2009. ISCE'09. IEEE 13th International Symposium on*, pp. 47–51, IEEE, 2009.

[9] F. Porikli, "Integral histogram: A fast way to extract histograms in cartesian spaces," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 829–836, IEEE, 2005.

[10] C. Yang, R. Duraiswami, and L. Davis, "Fast multiple object tracking via a hierarchical particle filter," in *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, vol. 1, pp. 212–219, IEEE, 2005.

[11] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "Fpga-based real-time pedestrian detection on high-resolution images," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2013 IEEE Conference on*, pp. 629–635, IEEE, 2013.

[12] N. J. Gordon, D. J. Salmond, and A. F. Smith, "Novel approach to nonlinear/non-gaussian bayesian state estimation," in *IEE Proceedings F (Radar and Signal Processing)*, vol. 140, pp. 107–113, IET, 1993.

[13] A. Haug, "A tutorial on bayesian estimation and tracking techniques applicable to nonlinear and non-gaussian processes," *MITRE Corporation, McLean*, 2005.

[14] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *Signal Processing, IEEE Transactions on*, vol. 50, no. 2, pp. 174–188, 2002.

[15] D. Simon, *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons, 2006.

[16] T. Li, T. P. Sattar, Q. Han, and S. Sun, "Roughening methods to prevent sample impoverishment in the particle phd filter," in *Information Fusion (FUSION), 2013 16th International Conference on*, pp. 17–22, IEEE, 2013.

[17] M. Athans, R. P. Wishner, and A. Bertolini, "Suboptimal state estimation for continuous-time nonlinear systems from discrete noisy measurements," *Automatic Control, IEEE Transactions on*, vol. 13, no. 5, pp. 504–514, 1968.

[18] S. K. Zhou, R. Chellappa, and B. Moghaddam, "Visual tracking and recognition using appearance-adaptive models in particle filters," *Image Processing, IEEE Transactions on*, vol. 13, no. 11, pp. 1491–1506, 2004.

[19] M. D. Breitenstein, F. Reichlin, B. Leibe, E. Koller-Meier, and L. Van Gool, "Robust tracking-by-detection using a detector confidence particle filter," in *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 1515–1522, IEEE, 2009.

[20] L. Mihaylova, P. Brasnett, N. Canagarajah, and D. Bull, "Object tracking by particle filtering techniques in video sequences," *Advances and Challenges in Multisensor Data and Information Processing*, vol. 8, pp. 260–268, 2007.

[21] M. E. O'NEILL, "Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation,"

[22] M. Z. Islam, C. Oh, J.-S. Yang, and C.-W. Lee, "Dt template based moving object tracking with shape information by particle filter," in *Cybernetic Intelligent Systems, 2008. CIS 2008. 7th IEEE International Conference on*, pp. 1–6, IEEE, 2008.

[23] J. Lewis, "Fast normalized cross-correlation," in *Vision interface*, vol. 10, pp. 120–123, 1995.

[24] S. A. de Araújo and H. Y. Kim, "Ciratefi: An rst-invariant template matching with extension to color images," *Integrated Computer-Aided Engineering*, vol. 18, no. 1, pp. 75–90, 2011.

[25] T. A. B. G. Marsaglia, "A convenient method for generating normal variables," *SIAM Review*, vol. 6, no. 3, pp. 260–264, 1964.

[26] G. Marsaglia, W. W. Tsang, *et al.*, "The ziggurat method for generating random variables," *Journal of statistical software*, vol. 5, no. 8, pp. 1–7, 2000.

[27] M. Chitchian, A. Simonetto, A. S. van Amesfoort, and T. Keviczky, "Distributed computation particle filters on gpu architectures for real-time control applications," *Control Systems Technology, IEEE Transactions on*, vol. 21, no. 6, pp. 2224–2238, 2013.

[28] M. Chitchian, A. S. van Amesfoort, A. Simonetto, T. Keviczky, and H. J. Sips, "Adapting particle filter algorithms to many-core architectures," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 427–438, IEEE, 2013.

[29] R. Wester and J. Kuper, "Design space exploration of a particle filter using higher-order functions," in *Reconfigurable Computing: Architectures, Tools, and Applications*, pp. 219–226, Springer, 2014.

[30] J. S. Malik, A. Hemani, and N. D. Gohar, "Unifying cordic and box-muller algorithms: An accurate and efficient gaussian random number generator," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pp. 277–280, IEEE, 2013.

[31] M. D. Vose, "A linear algorithm for generating random numbers with a given distribution," *Software Engineering, IEEE Transactions on*, vol. 17, no. 9, pp. 972–975, 1991.

[32] G. Kitagawa, "Monte carlo filter and smoother for non-gaussian nonlinear state space models," *Journal of computational and graphical statistics*, vol. 5, no. 1, pp. 1–25, 1996.

[33] B. H. Dekens, *Low-Cost Heterogeneous Embedded Multiprocessor Architecture for Real-Time Stream Processing Applications*. PhD thesis, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands, okt 2015.

[34] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. P. Llopis, and P. Lippens, "C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, vol. 7, no. 3, pp. 233–270, 2002.

[35] B. H. Dekens, M. J. Bekooij, and G. J. Smit, "Real-time multiprocessor architecture for sharing stream processing accelerators," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 81–89, IEEE, 2015.

[36] G. G. Wevers, "Hardware accelerator sharing within an mpsoc with a connectionless noc," September 2014.

[37] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp. 191–200, ACM, 1998.

[38] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 2, p. 11, 2007.

[39] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *The Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[40] M.-C. Lee, W.-L. Chiang, and C.-J. Lin, "Fast matrix-vector multiplications for large-scale logistic regression on shared-memory systems,"