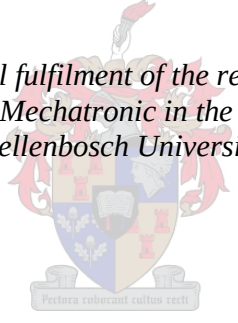
A feasibility analysis into the inference of a generic object tracking algorithm on a general-purpose single board computer

by
Gerard Louis Walsh

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering Mechatronic in the Faculty of Engineering at Stellenbosch University



Supervisor: Dr. Willie Smit

April 2019

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: April 2019

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / *Plagiarism Declaration*

1 *Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.*

2 *I agree that plagiarism is a punishable offence because it constitutes theft.*

3 *I also understand that direct translations are plagiarism.*

4 *Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.*

5 *I declare that the work contained in this assignment, except otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.*

GL Walsh Voorletters en van / *Initials and surname*	Datum / *Date*

Abstract

A feasibility analysis into the inference of a generic object tracking algorithm on a general-purpose single board computer

G. Walsh
Department of Mechanical and Mechatronic Engineering
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa
Thesis: MEng (Mechatronics)
April 2019

Algorithms that are able to track generic objects in real-time have many useful applications such as security and traffic surveillance, augmented reality and sports analytics. Practical implications of tracking algorithms are further enhanced when the algorithms are able to be processed in real-time on mobile devices. Mobile SoCs are compact and energy efficient by design (Carroll, 2010) and present a possible implementation platform.

Modern object tracking algorithms (Bertinetto, 2016) rely on computationally intensive convolutional neural network (CNN) architectures. CNNs are currently not able to be processed in real-time on mobile devices (Lu, 2017). The research conducted in this thesis aimed to address the prior shortcoming in the computation of object tracking algorithms on mobile devices. A classically-designed object tracking algorithm, CMT, was chosen for investigation due to its flexibility in configuration of image features. CMT is independent of the method used to compute classical image features, permitting the usage of binary descriptor vectors that can be effectively computed. The primary investigation was the algorithm's suitability for implementation on a general-purpose heterogeneous computing platform. This was performed since heterogeneous platforms are common in mobile devices such as smartphones (Ignatov et al, 2016). A mobile platform was chosen based on available hardware acceleration support and heterogeneous computing capacity.

Baseline performance of 2.22 FPS was initially established on the chosen mobile hardware platform utilizing a strictly CPU execution model. An investigation into the optimal choice of image features realized a 742% increase in FPS. The FPS was further increased through the utilization of on-board SIMD processors and achieved a real-time performance of 21.39 FPS. Due to OpenCV not supporting mobile GPU architecture, heterogeneous CPU-GPU acceleration on the mobile platform could not be investigated. When a desktop heterogeneous platform was utilized, the FPS throughput increased by 205% through heterogeneous CPU-GPU acceleration when compared to a CPU implementation. Results from an investigation into concurrent execution on the desktop platform did not meet theoretical expectations since the set of asynchronous GPU functions utilized did not execute completely asynchronously from the CPU.

Real-time computation was achieved by utilizing strictly CPU execution on the mobile platform. The results of heterogeneous CPU-GPU acceleration on the desktop platform are transferrable to a mobile platform, provided that the image processing library supports the mobile platform's heterogeneous capabilities. Thus, mobile devices are feasible platforms for real-time computation of classical object tracking algorithms due to the attained FPS, with further increases in FPS possible through heterogeneous CPU-GPU acceleration. This realizable increase in FPS through CPU-GPU acceleration indicates more computationally demanding algorithms can achieve real-time computation. Theoretical concurrent acceleration techniques were deemed to be of value as they present the upper limit achievable in a CPU-GPU heterogeneous execution model.

Uitreksel

'n Uitvoerbaarheids-analise in die inferensie van 'n generiese objek-naspeur algoritme op n gewone veeldoelige enkelbord rekenaar.

G. Walsh

Departement van Meganiese en Megatroniese Ingenieurswese

Universiteit Stellenbosch

Private Sak X1, 7602 Matie-land, Suid-Afrika

Tesis: MEng (Megatronies)

April 2019

Die naspeur van 'n enkele, generiese voorwerp in 'n RGB insetstroom word baie bruikbaar wanneer die naspeur algoritme onder bespreking instaat is om die inset beeld vinniger te prosesseer as wat dit die beeld ontvang, gedefinieer deur die VOT uitdaging synde 'n reële tyd naspeurder, en meer so wanneer die algoritmes direk na mobiele toestelle gelei kan word. Ultra moderne naspeur algoritmes (Bertinetto, 2016) vertrou op konvolusionele neurale netwerk argitekture, en meer spesifiek die bereken duur konvolusionele filter, om 'n ekstrak van die kenmerke van die beeld te maak, en is daarom nie geskik vir reële tyd mobiele implementasie nie (Lu, 2017). Terwyl algoritmes ontwerp word met reële tyd toepassing in gedagte, word implementering hardeware tans nie as 'n belangrike faset beskou nie, en word tafelrekenaar hardeware dus tans vir inferensie gebruik. Die gebruik hiervan is hoogs onaantreklik aangesien dit die toepassingspasie van die voorwerp naspeur algoritmes beperk tot situasies waar uitgebreide berekeningskrag/rekenaarkrag? vir inferensie beskikbaar is.

Die navorsing wat vir hierdie tesis onderneem is, het die klassiek-ontwerpte voorwerp naspeur algoritme ondersoek, daarin dat dit handgemaakte kenmerke gebruik, en die uitvoerbaarheid daarvan vir mobiele implementasie. Dit is voltooi deur die inagneming van die algoritme se toepaslikheid vir implementering, en voortspruitende werkverrigting, op 'n algemene gebruik heterogene rekenaar platform - 'n platform wat wyd beskikbaar is in mobiele toestelle soos slimfone.

Verskeie hardeware versnelling en sagteware optimalisasie tegnieke is ook ondersoek, met die inisiëel uitgevoerde eksperimente wat aangedui het dat die algoritme effektief gelei kan word op 'n suiwer CPU- gebaseerde mobile platform en dat dit reële tyd raam prossesering snelhede bereik. 'n Hindernis is teëgekom, as gevolg van 'OpenCV' biblioteek wat nie ontwerp is om die mobiele platform se GPU argitektuur te ondersteun nie, en gevolglik kon die GPU versnelling op die mobile platform nie ondersoek word nie. Nietemin is bevind dat die algoritme grootliks gebaat het by die GPU versnelling, wanneer 'n desktop heterogene platform gebruik is, en vervolgens geskik om ook tegelyke tyd uitgevoer te word. Resultate vooruitspruitende die ondersoek na die gelyktydige uitvoering was teleurstellend aangesien die stel asinkroniese funksies wat gebruik is nie vertoon het soos beoog nie, dit is nie totaal asikronies van die CPU nie. Die resultate van die geïmplimenteerde GPU en die teoreties gelyktydige versnellingstegnieke is steeds van waarde geag met betrekking tot die doelwitte van die projek aangesien die resultate in 'n mate oordraagbaar is na 'n mobiel heterogene platform, gegewe 'n beeld prosessering sagteware biblioteek wat die mobile GPU kan benut.

Acknowledgements

The author would like to firstly thank Dr Willie Smit for his continued guidance and stimulating input for the duration of the project. For the chance to pursue research in such an interesting and promising field, the author is truly appreciative and as well as for the support when pursuing overseas travels. Secondly, to the International Office at Stellenbosch University, and Ms Sarah van der Westhuizen, the chance to experience a semester aboard was once in a lifetime. Thirdly to family, friends old and new, it was a great chapter of growth - may the experiences and learning never end.

Table of contents

List of figures

Page

List of tables

Page

Nomenclature

b	Object bounding box
d	Image feature descriptor vector
FPS	Frames per second processing rate
FN	False negative classification
h	Object centre vote
I	Image array of pixels
L_K	Set K of part correspondences
m_i	Correspondence i of object parts
$p(x_i)$	Descriptor vector of keypoint x_i
R	Standard rotation matrix
s	Scale
S	Success ratio
t_n	Set n of reference descriptors
TP	True positive classification
w	Transitive predicate
x	Image patch
x_i	Keypoint i in image coordinates
x_i^t	Object part i at time t
z	Image template patch
Z	Normalized initialization image
$\propto$	Overlap ratio
γ	Second nearest neighbour ratio threshold
φ	Consensus predicate
μ	Object centre in normalized image coordinates
$\emptyset$	Overlap threshold

1 Introduction

1.1 Background

Unmanned Aerial Vehicles (UAVs) have seen a recent increase in both consumer and industrial applications. One of the novel applications is in Concentrated Solar Plants (CSPs), where UAVs assist in heliostat calibration. The increase in the application of UAVs to a broad range of tasks can be attributed to the increase in sophistication of both onboard software and hardware. The improved software sophistication is, in turn, largely accredited to the current research effort into computer vision (CV). CV is a field within machine learning whereby computers seek to interpret the information found in digital images (Szeliski, 2010). Computer vision allows UAVs to interpret the information of their immediate surroundings in a manner that can be likened to that of the visual cortex (Goodfellow, 2015). Using computer vision, UAVs can identify objects within their field of view and formulate 3D recreations of their surroundings. This gathered information of the immediate surrounding can then be utilized to plan a path of travel through the physical world, whilst avoiding or following specific objects.

The task of tracking objects of interest is not limited to static objects such as heliostats. UAVs are also tasked with tracking more dynamic and deformable objects such as people partaking in various sports, automobiles or even wild animals. The task of tracking an object is a well-defined problem since its formative application in a military context (Kalman, 1960).

There are many valid assumptions to be made in the process of tracking objects. One such assumption is that on a per frame basis, the object of interest is assumed to be displaced an insignificant amount (Lucas and Kanade, 1981). One-shot object tracking algorithms seek to make assumptions that are as conservative as possible, to ensure the methods are applicable to the widest range of scenarios. The method is not granted any prior knowledge of the object, nor does it search a restricted area in the image once knowledge has been gained of its position. For tracking algorithms to be of use in real-world scenarios the output needs to online and within real-time constraints.

1.2 State of the art

Similarly to other image processing tasks, the scientific field of visual object tracking has not managed to avoid the increasingly common application of Artificial Neural Networks (ANNs). Arguably, the popularity of ANNs in image

processing tasks is due to the success that ANNs have achieved in image classification, pioneered by AlexNet (Krizhevsky, 2012). State-of-the-art object tracking methods currently employ a moderately shallow network of layers when compared to state-of-the-art object detection networks that are some 20 layers deep, such as You Only Look Once (Redmond, 2015).

The differentiating attributes between the approach of state-of-the-art ANNs and classical one-shot object tracking algorithms are the utilization of convolutional neural networks (CNNs). CNNs are a subset of ANNs that are especially effective at image processing tasks. Predominantly, the effectiveness of CNNs is due to the weights of the convolutional filter operators are learnt offline, with a suitable dataset for the given task. Offline training of the filter weights enables extraction of image features that allow the network to be able to generalize well, particularly on unseen examples. Classical one-shot object tracking methods rely on handcrafted image feature extraction models to provide discriminative features to the classification portion of the algorithm. Classical image feature methods such as the Scale Invariant Feature Transform (Lowe, 2004), present poorer performance as the methods generally attempt to span a generic application domain. Whilst these tasks commonly require that the feature descriptor be uniquely discriminative, convolutional filter operators have the advantage of being trained for the specific task. For example, object detection networks such as You Only Look Once are for the sole purpose of object detection and localization.

A vast computational cost is however incurred in computing CNN networks like the object detection network mentioned previously – a computation cost of 30 billion floating point operations (BFLOPS) for a single prediction of You Only Look Once (Redmond, 2015). This computational burden can be attributed to the computation of the convolutional filters (Wu, Leng, Wang and Cheng, 2015). Convolutional image filters are usually computed by hardware that can accelerate data parallel operations such as Graphics Processing Units (GPU). GPUs are commonplace as of late in consumer general and high-performance personal computers. However, the devices utilized by the research and industrial communities commonly rely on proprietary *desktop* hardware and associated software and are not mobile. Thus, the application domain of real-time object tracking algorithms (Bertinetto, 2016) is constrained due to the reliance on desktop GPUs, to achieve real-time computation.

Arguably, real-time computation of object tracking algorithms on mobile platforms has not been considered due to researchers focusing their efforts on tracking effectiveness. However, object tracking algorithms that are able to be computed in real-time are documented by the real-time challenge within the Visual Object Tracking challenge (Kristan *et al*, 2016). The caveat in the evaluation of real-time trackers by the Visual Object Tracking challenge is the utilization of desktop hardware.

Classical object tracking algorithms and image features present a feasible solution when pursuing real-time performance as they do not employ convolutional filters. Research in classical image features culminated in Orientated FAST and Rotated Brief (Rublee, 2011) which performed comparatively (Rublee, 2011) to SIFT, but faster by two magnitudes of order. Classical image features can be efficiently computed due to utilizing binary descriptor vectors, as in Orientated FAST and Rotated Brief. Classical object tracking algorithms, utilizing lightweight image features, thus present a feasible alternative to CNNs for real-time object tracking computation on mobile devices. Mobile SoCs are a feasible platform for investigation as they rival desktop hardware that was available earlier in the last decade (Ignatov *et al*, 2018).

The aim of this research is to implement a classical object tracking algorithm and assess its feasibility of obtaining real-time performance on a mobile device. The initial hypothesis is that real-time performance, of the classical object tracking algorithm, can be achieved by utilizing heterogeneous CPU-GPU acceleration on the mobile platform.

1.3 Objectives

The principal objective of this thesis was to implement the chosen object-tracking algorithm effectively on a general-purpose single board computer. Real-time performance was the criteria for assessing the successful implementation of the algorithm. Real-time performance was to be achieved by efficiently using onboard hardware and through efficient software design.

Research objectives

- Concurrently establish a general-purpose single board computer platform as well as an open-source generic object tracking algorithm, the latter of which will be investigated whether suitable for inference on the chosen hardware platform.

- Identify and investigate the functions within the algorithm that present the largest computing overhead, on the chosen hardware.

- Determine how the algorithm can be accelerated with on-board hardware and software optimizations.

- Investigate which optimizations, software and hardware, should bring inference improvements and quantify the improvements on specific functions

- Implement an accelerated variant of the given algorithm on the given hardware.
- Investigate the suitability of the chosen algorithm for a concurrent execution model.

2 Literature review

2.1 Machine learning

Machine learning is the statistical science of enabling computers to be able to expose underlying patterns in data (Murphy, 2012) and make high level interpretations. Machine learning has seen an exponential rise in both its proficiency to solve problems as well as its applicability in common scenarios. Machine learning has been solving problems, such as handwritten digit recognition (LeChun *et al,* 1990) or Optical Character Recognition (OCR) since the late 1990's, but only since the proliferation of modern computers and the associated accessibility of digital data or big data (Murphy, 2012) has the discipline started to flourish.

Machine learning can be separated into three fundamental approaches that are used to infer 'learning' into an algorithm: supervised and unsupervised learning, being natural opposites to one another in their approaches, and reinforcement learning (Murphy, 2012). Supervised learning can be described as the task of learning an input to output function mapping, given a set of labelled data points that are distributed amongst the classes that we seek to be able to discriminate between. The data points, more commonly known collectively as the training dataset, and the nature of their labelling differentiates supervised and unsupervised learning. Unsupervised learning also requires a set of training examples to infer learning in the algorithm, but instead the algorithm seeks to learn the inherent structure (Murphy, 2012) in unlabelled data points, in which the class of each data point is not explicitly defined. Whilst the approach of unsupervised learning may seem more akin to the so-called forthcoming Artificial Intelligence Singularity, it still requires much input from the architect of the algorithm in order to leverage any underlying structure within the data.

A learning paradigm that is more align with the popular culture surrounding the research topic of machine learning is reinforcement learning. Reinforcement learning is a method congruent with the natural methods of learning, through trial and error. Upon first approach, the method is more easily grasped than that of the previously mentioned methods, as the algorithm simply has a defined reward (Russell and Norvig, 2010) pathway and attempts take actions that seek to maximize this reward. Reinforcement learning is an area of research that is pursued in many disciplines outside of the world of machine learning such as game theory, due to its general ability to learn an optimal policy in an environment.

A common goal between all methods is that they seek to make light work of interpreting vast amounts of data (Murphy, 2012) that is inherent to our digital world, and not easily processed by humans. Machine learning has seen

applications ranging from time series forecasting (Bontempi, Taieb and Borge, 2013) to being able to detect and classify objects in digital images (Redmond, 2015). The discerning factor between which approach is utilized to solve a problem or to leverage insight depends predominantly on the situation at hand but also the required output from the algorithm. A common machine learning application may even leverage both supervised and unsupervised learning in the same application. As a first step we might want to first learn the inherent structure within a vast dataset. The principal components, which were extracted from the unsupervised portion of the application, could then be passed to a supervised portion of the application in order to either create or train a classifier.

2.2 Computer vision

Within the vast depth of research fields of machine learning is the interdisciplinary field known as Computer vision. Computer vision (CV) presents an enormous challenge, regarding dealing with vast amounts of complex data (Szeliski, 2010). CV is both a highly active and an extremely promising area research field. The annual Computer Vision and Pattern Recognition Conference (CVPR) has seen over ten thousand publications in its short 30-year existence.

However, modelling the real world and its associated complexities with digital image sensors in a binary format presents an enormous challenge when we seek to interpret this information in a high-level manner (Szeliski, 2010). Common challenges in computer vision, and the associated digital representation of a scene, are changes in light intensity, various viewpoints of a scene and interpreting the same scene at various levels of scale. Many challenges arise as we are essentially representing a 3-dimensional space on a 2-dimensional plane (Szeliski, 2010). Not only are we attempting to model the natural world with a seemingly simple binary representation, the representations that we utilize are also high dimensional. A 640 x 480-pixel RGB colour model image contains roughly 1 million bytes of information, and essentially occupies a 640 x 480-dimensional space.

Common areas of application of computer vision are, for example, in medical imaging where a computed tomography scan may be post-processed by an algorithm to detect the presence of cancerous tumours within the human body, or in robotics application where object recognition needs to be performed in order to navigate an environment and avoid objects successfully. These tasks are further complicated when video is fed into the computer vision algorithm. To provide meaningful output on video input the image processing rate is usually required to meet or exceed that rate at which information is captured. This computational burden is further complicated by modern image capturing devices usually representing the real world with High Definition (HD) images that occupy some 1980 x 1080-dimensional spaces.

Despite the aforementioned challenges, it should be reiterated that computer vision has been successfully transforming our world and automating many tasks since the 1990s. Once such task that humans complete without perceiving the inherent complexity on hand is that of generic object tracking. Recognising objects, a separate subsection within computer vision, and tracking them through a video feed is an attractive tool. Once the field of object tracking is sufficiently solved, it promises to bring many advances to applications of machine learning. The term *sufficiently* solved is referred to, due the existence of the No Free Lunch Theorem (Wolpert, 1997), which states that no singular algorithm can solve every problem. Wolpert's theorem can be applied to the task of generic object tracking, in which we realize no singular algorithm will be superior in all situations - some scenarios requiring speed over accuracy or superior robustness to explicit accuracy.

2.3 Neural networks

Realistically, the approach of ANNs is not a recent discovery as it has been present since the 1940s. Confusion has surfaced throughout history due to the array of naming schemes ANNs accumulated over time, starting with cybernetics and later becoming connectionism (Goodfellow, 2015). Recent theoretical discoveries in ANN structure such as the activation function in the hidden layers of a network, and the development of the propagation of errors backward through a network (Rumelhart *et al*, 1986) have enabled effective training and state-of-the-art classification accuracy. Proliferation of the general-purpose computing on Graphics Processing Units (GPGPU) have realized both training and inference of ANNs within realistic time boundaries. The combination of theoretical progression and available computing power has enabled the widespread success of ANNs in a variety of applications. Neural Networks gain their artificial prefix as the connectionism, as illustrated in Figure 2.1, draws inspiration in its construction from biology, where the brain (Goodfellow, 2015) is comprised of multiple layers of neurons that are interconnected on varying levels. This common structure, illustrated in Figure 2.1, is also known as the Multi-layer Perceptron (MLP). It gives rise to the term Deep Neural Networks, when a network is comprised of many hidden layers repeatedly.

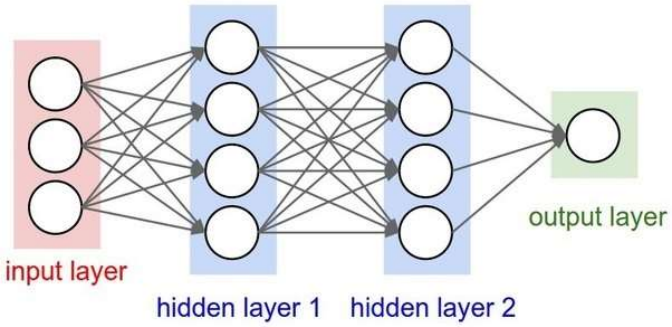


Figure 2.1: The Multilayer Perceptron illustrating the connectionism between nodes (Karpathy, 2015).

It was showed that that Deep Neural Networks (DNNs), deeper than previously employed (Krizhevsky, 2012) in vision-based machine learning tasks, could outperform state-of-the-art approaches to object detection in images. This was due to the hierarchy of image features that were learnt for the specific task on hand by the network, during offline training. Object detection approaches that utilized hand-crafted image features and classifiers such as a Support Vector Machine were from 2012 onwards considered outdated or classical.

2.4 Image features

Figure 2.1 illustrates the network construction is more akin to those tasked with general regression analysis. This construction is commonly referred to as Fully Connected Network (FCN), due to the interconnection between each node in the network. DNNs commonly utilized in tasks within computer vision are referred to as Convolutional Neural Networks (CNNs). CNNs are generally employed for image processing tasks such as object detection, classification and object tracking. CNNs usually consist of a sequential series of convolutional filter layers followed by a minimal set of fully connected layers. AlexNet (Krizhevsky, 2012) is illustrated in Figure 2.2 as reference for a generic CNN employed in image processing tasks.

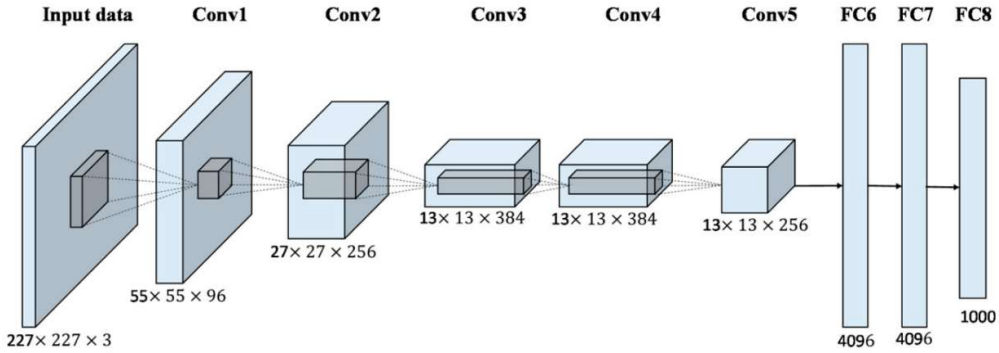


Figure 2.2: The Alexnet (Krizhevsky, 2012) CNN for image classification, with 5 convolutional layers are followed by 3 Fully Connected Layers (The intermediate max pooling layers between convolutional layer 1 - 2 and 2 -3 are not illustrated for simplicity).

2.4.1 Convolution

While the definition of convolution differs depending upon which application space it is utilized in, when we speak of the process of convolution, we refer to the image filter operation defined by

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \qquad (2.1)$$

whereby I is an input image, a 3-dimensional array, that is filtered using the filter operation *K*. This operation, illustrated below in Figure 2.3 as the kernel, is passed over the image at a defined spatial location, where the input image is now a 2-dimensional array. The input image attains it's third dimension due to the RGB colour model, with each of the three dimensions represent a certain colour channel of which the convolutional filter manages independently, as illustrated in the manner below.

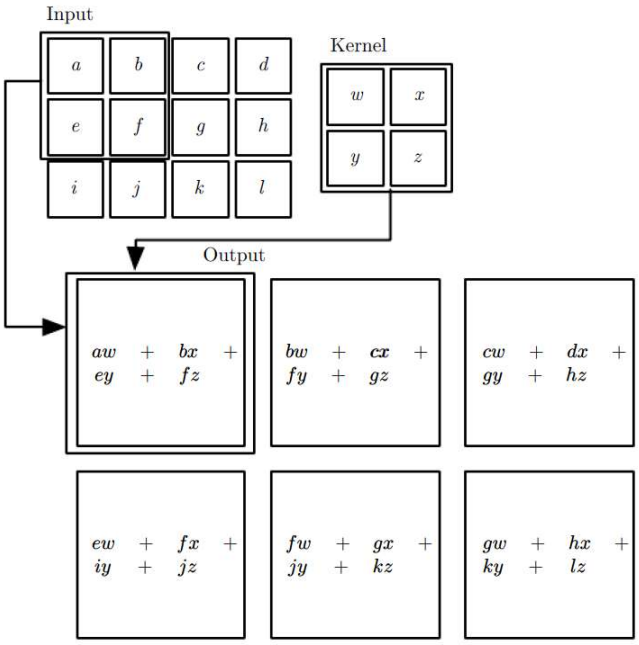


Figure 2.3: The convolutional filter processing a 2-dimensional input, with the kernel or convolutional filter and input image highlighted (Goodfellow, 2015).

Each layer of a CNN produces an output whereby the dimensionality is usually smaller than that of the input layer, as the goal is to perform effective dimensionality reduction, with an exception for the third dimension. A larger third dimension is produced due to the fact that many filters are passed over the input image and as a result an output is produced that is smaller in both input image width and height, but has more depth layers. This is illustrated in Figure 2.2 where the input is 227x227x3 and, after the initial set of convolutional filters, there are a total of 96 outputs each of width and height of 55 pixels.

The output of a convolutional filter or kernel, of which both can be used interchangeably, is referred to as a feature map and contains a 3-dimensional array of features extracted by each of the kernels. The index of the third dimension of the feature map refers to the index of which a unique kernel, as illustrated above in Figure 2.3, produced the specific feature map and the third dimension grows in size throughout the convolutional stages, with the remaining dimensions shrinking. Another fundamental aspect of the CNNs are that the kernels are not usually applied to every pixel in the input image as this would include redundant information in the feature maps. Filters are usually applied in a stride, with a stride of 2 implying that the centre of the filter only being applied to every third pixel in

a specific direction of either height or width, and that the kernel strides two pixels before operating in both planar image dimensions.

However, information is not disregarded with the application of a strided filters due to the filters possessing a 'receptive field'. Receptive fields refer to the area in the input of which a filter can 'see' and through the utilization of a stride, the amount of overlap between applications of a kernel can be controlled. In Figure 2.3, with a stride of 1, the kernels share 50% of the input data between successive applications, and thus as have a similar receptive field. DNNs tend to have a larger receptive field than shallower networks, due to the same input propagating through the network, partly enabling their effectiveness in extracting discriminative features (Goodfellow, 2015).

Zero-padding and pooling are two aspects that also influence the dimensionality of the feature maps generated by convolutional filters. Zero-padding essentially appends the input to the filter with 8-bit 0 values, where necessary, in order to preserve spatial dimensions in the output and allows independent command over both the kernel width and the dimensions of the output (Goodfellow, 2017). Pooling is another dimensionality-reducing operator that takes in a two-dimensional input and reduces the size of both dimensions by using a selective operator, such as, for example, a max pooling operator. Pooling serves to down-sample the input by separating the input into non-overlapping grids of which it selects, for example, in a 2x2 grid with max-pooling, the singular highest integer value. It is an effective method employed in practice that greatly reduces both computation and overfitting, with the intuition that specific feature location is not as critical as general location (Goodfellow, 2015).

Finally, the last notable operator in modern CNNs is the activation unit that serves to provide activation of a node based on a threshold, with inspiration drawn from the research conducted on the visual system (Hubel, 1959). Modern CNNs commonly utilize the Rectified Linear Unit (ReLu) which can be likened to that of a half-wave rectifier in electronic signal processing and, like all activation functions, serves to provide an increase in network nonlinearities, which is critical in Neural Networks as they can learn complex functions and avoid gravitating toward being deeply stacked linear operators. The convolutional portion of the network, illustrated in Figure 2.2 by the first 5 layers, generally follows the construction of a sequential series of operators by starting with a set of convolutional kernels, after which comes an activation unit and a pooling layer. This section of the network is responsible for dealing with image features, after which the generated feature maps are passed to the classification portion of the network. With the continued research efforts into more effective CNNs, novel operators like skip connections, such as those found in ResNet (He, 2015), are becoming more common place as well.

Convolutional Neural Networks produce superior features to classical features detectors and descriptors arguably for two reasons. The first being their design being closely inspired by the human visual system, which responds on many different levels to varying types of visual stimuli (Goodfellow, 2015). Secondly that there is a hierarchy in visual input, visual features and brain response. This hierarchy is learnt for a specific task and embedded in the weights of the kernels can be learnt. Arguably, a feature is designed to be as discriminative as possible, from the approach of both the classical and CNN, and as such the single feature should be useful for many tasks where we need to distinguish between image information, being it either for 3D reconstruction or for optical flow fields. Research (Bertinetto, 2016) has however showed that CNNs perform well on image recognition-based tasks.

2.4.2 Handcrafted

Classical image features are handled in two-step manner - the input image is first passed through a feature detector that filters the image to determine the spatial coordinates where interest points are present, known as keypoints, after which the area surrounding the keypoints are described with a corresponding feature vector, generated by a feature descriptor. Arguably, CNNs follow a similar procedure, but the process is completed over several steps for which the network designer has control, and the features develop as they propagate through the CNN rather than being completed in two distinct steps. While the literature available that covers the classical approaches to image features is extremely vast and beyond the scope of this research, the following two sections serve to cover the basic approach in which local image features are handled. Local features are defined by the fact that they serve to represent discrete spatial positions in images, rather than the image as a complete region, such as the Histogram of Orientated Gradients descriptor (HOG).

2.4.2.1 Feature detection

Feature detection methods aim to filter the input image, to discern the locations of features that the function deems to be both discriminative, invariant to varying transformations, and repeatedly detectable. Keypoints are generated by the feature detector in image coordinates and locate an area of interest, for instance either an edge or a sharp change in colour contrast. Initial work on keypoint detectors relied on extremely simple approaches and simply utilized a sum of squares difference approach to differentiate between and detect keypoints in image regions. Image feature detectors progressed steadily by incorporating more advanced operators such as taking derivatives in both directions (Harris, 1988). Subsequently, the GoodFeaturesToTrack approach (Shi and Tomasi, 1994) furthered the prior approach by incorporating a corner strength test. It was only with the pioneering work of Scale Invariant Feature Transform (Lowe, 2004) that

feature detectors and descriptors became of great use to the computer vision community. Scale Invariant Feature Transform (SIFT) is, however, limited due to its computational complexity. Importantly for the purpose of this research thesis, computational complexity was addressed with the approach of Features from Accelerated Segment Test (Rosten and Drummond, 2005), which considered individual pixels as keypoints by investigating a 16-pixel circular region around the pixel. The candidate pixel was considered a keypoint if there existed a set of pixels *n* which were within the circular region, and were all at the opposite end of the pixel's illumination intensity, by a threshold *t*. It should also be mentioned the approach of Lowe, as well as aforementioned approach, required the input image to be in grayscale, as this greatly simplified both the brightness check of the Features from Accelerated Segment Test (FAST) detector and the computation of SIFT keypoints.

Whilst the FAST keypoint detector was efficient in terms of inference, it had inherent issues due to its simple approach that made FAST keypoints less effective than keypoints detected the SIFT operator. The FAST detector in practice produced a large keypoint count, which was subsequently addressed by, as illustrated in Figure 2.4, first only checking a determined minimum number of pixel positions, being 1 and 9, and then if those pass the previous threshold test, 5 and 13 are also tested. Image locations that passed these preliminary tests were then added to the list of keypoints, after which a full neighbourhood test could begin.

Whilst the FAST detector was still not suitably robust for effective implementation, the author attempted to address these issues, firstly with the suboptimal choice of pixels for the reduced segment test (pixels through 1,3,9 and 13). Since these 4 pixels were aligned 45 degrees apart from one another, a corner could easily be missed if not correctly aligned and was for instance oriented vertically. The author addressed this issue, as well as many redundant keypoints being detected in close proximity to another, by testing the detector over a dataset of images from which a decision tree was learnt that yielded more robust corner detection and employed Non-Maximum Suppression in order to avoid detecting keypoints that were close to within a vicinity of one another. The reader is referred to the original paper for a further discussion (Rosten, 2005).

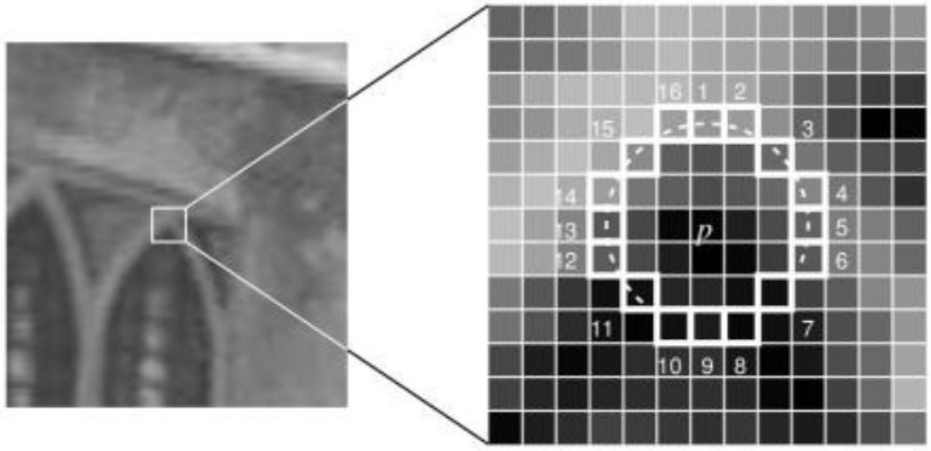


Figure 2.4: The FAST feature detector, where the circular region around a candidate pixel is investigated and demarcated with the 16-pixel region (Rosten et al., 2010).

2.4.2.2 Feature description

Following successful keypoint detection, keypoints are subsequently associated with a feature descriptor vector that assigns a unique vector to a keypoint such that it could be reliably detected in other frames. Pioneering work was achieved with very simple features in early attempts at facial recognition which utilized difference operators between sums of pixels from a pair of regions, similar to feature detection. The SIFT descriptor vector, which was demonstrated to be supreme to other feature descriptors at the time, was constructed by calculating a Histogram of Orientated Gradients that summarized the image gradient orientations of the image patch surrounding the keypoint. SIFT greatly contributed to their effectiveness of many computer vision algorithms that utilized key points such as panoramic stitching or 3D reconstruction, but was mostly impractical in real-time applications due to a 16x16 patch around the keypoint of interest being summarized with a 128-dimensional feature descriptor, having too large of a memory footprint. Researchers subsequently put significant effort into developing descriptors that were as effective as SIFTs' but were more computationally efficient. A binary feature descriptor was proposed (Calonder, 2010) that promised to bring about real-time performance by testing, after smoothing candidate image patches, brightness intensity between pairs of pixels. Binary Roust Independent Elementary Features (BRIEF) experimented with various sampling geometries for generating pairs of pixels and eventually settled scheme that sampled randomly from a Gaussian distribution centred around the patch.

The most effective Gaussian distribution had a variance of $\frac{1}{25}S^2$ and zero mean, where S is the dimension of the square image patch. Using the proposed method BRIEF allowed for the efficient processing of descriptors that rivalled SIFT in terms of effectiveness. Researchers built on novel binary feature descriptor BRIEF and attempted to include invariance to rotation with the Binary Robust Invariant Scalable Keypoints (Leutenegger, 2011). Binary Robust Invariant Scalable Keypoints (BRISK) employed varying-width Gaussian smoothing operators to area surrounding the candidate image patches that were sampled by utilizing a fixed sampling pattern.

The question then becomes: once it is possible to reliably locate points of interest in images that can be repeatedly detected and matched after changes in image rotation, scale and illumination, what algorithms and computer vision concepts these keypoints and their associated descriptor vectors are useful for. Whilst this was certainly not natural progression in the development of image features, the use-cases became more complex as image features saw a similar increase in their effectiveness. Once such application that heavily relies on image features is generic object tracking, whereby we seek to track deformable random objects in images.

2.5 Object tracking

An application where discriminative features are of utmost importance is in the task of tracking objects in a sequence of images or from a live video feed. The target of an object tracking algorithm can be summarized (Maggio and Cavallaro, 2011) as predicating the spatial location, in an image, of an object in a video sequence comprising of individual images. Authors have studied how to track objects that belong to specific classes, for instance humans (Koller *et al,* 1994). This approach supplies the algorithm prior knowledge of the object of interest and should return higher tracking effectiveness than an approach without this knowledge. However, this requires a specific subset of assumptions for each object of interest, which is unattractive as each object would have to studied prior to tracking.

Generic object tracking algorithms have no prior knowledge of the object of interest that is to be tracked and seeks to make the most conservative assumptions possible. Assumptions such as reduced-space searching for the object once the object is localized in the previous frame, or that the object will remain rigid throughout the tracking sequence are avoided in generic object tracking. In practice objects such as humans involved in physical activity can readily deform throughout a sequence.

Generic object tracking algorithms are tasked with (Nebehay, 2016) in a sequence of images or a live video feed of most predominantly the RGB colour model: given a bounding box b, comprising of the top left image coordinates and accompanying width and height, that surrounds the object from an initialization image I_1, the purpose of the algorithm is to conduct an optimization exercise to find the bounding box b in each image that presents the largest overlap with the object of interest, whilst simultaneously minimizing the overlap with background clutter for every input at time t. The utilization of a bounding box, however simple it may seem, is an exercise in trade-off between fully encompassing the object and including the most minimal number of pixels that belong to the background and is utilized in all state-of-the-art detection and localization tasks.

The task of object tracking can, algorithmically, be separated into four distinct processes (Nebehay, 2016): an initial predication step, the extraction and processing of the relevant image features, localization and subsequently the updating of the model used to represent the object of interest.

The prediction step is usually only of value when an object of fixed class is being tracked and valid assumptions can therefore be made. Useful prediction assumptions are commonly made on the object's future location, based on previous motion (Kalman, 1960). Prediction is often error prone and can in turn be result in significant drift and error aggregation (Lepetit and Fau, 2005). For the reason of being applicable only to certain objects, the prediction step is not performed in generic object tracking.

The second step of feature extraction can be likened to dimensionality reduction in terms of general machine learning. The goal is to move from a high dimensional space of an image to a more compact and condensed representation of the image. The compact representation, of image into image features, allows more convenient and efficient comparison of images and their content. An image feature should be discriminative and therefore only belong to a unique object or certain region in an image. Feature invariance is a fundamental attribute of image features and allows a unique feature to be detected in multiple different scenes. Section 2.4 covers feature extraction methods, both current and classical methods, and the methods employed to overcome the task of effective dimensionality reduction.

Localization refers to the step of correlating image features to object location, completed in a local search or by detection. On first approach local search methods may seem to employ the use of a predictive model or step to localize the object. However, solely the information of object location from the previous image is regarded as valuable (Cannons, 2008) and can be utilized as a starting point in localizing the object for the current frame. The key difference in a local search compared to a prediction step is that only information from the previous time step

is used, and subsequently does not lead to drift. Optical flow (Lucas and Kanade, 1981) have been widely utilized to provide local search information. Optical flow provides image feature location by assessing per-frame image motion, assuming minor per-frame motion. Localization by detection relies on correlating extracted image features to a database of information that is *known* to belong to the object of interest. Localization by detection is commonly performed by comparing individual image features and utilizing a difference operator such as the Euclidean distance. This approach ignores all prior object location information, but some localization by detection methods do update the database of known object information.

The final task relates to the amendment of the object model with the information from the localization step, in order to keep the model relevant with time. Amongst key challenges are to avoid drift through the inclusion of erroneous information and to complete the update in time to avoid changes in appearance. It is hypothesized that the most successful approach (Nebehay, 2016) is to update the object model as conservatively as possible. Each tracking algorithm has a specific handcrafted criterion for this step and as such should be explored with each individual tracking approach.

2.5.1 Classical approach

Previous approaches to the task of generic object tracking generally constructed an object tracking pipeline out of the four well-defined building blocks of prediction, feature extraction, localization and model update (Nebehay, 2016). The tasks were generally approached through established machine learning methods such as utilizing classical image feature description and detection, and hand-crafted object models as in Hough Forests. Hough Forests (Gall, 2011) approach to object tracking utilized an approach similar to Random Forests for classification, but rather than attempting to interpret class labels, Hough Forests utilized a Random Forest classifier to discern an object centre.

A method which built upon the Hough Forest object model, whereby the object consisted of disjointed parts with a common centre, was Clustering of Static-Adaptive Correspondences for Deformable Object Tracking (Nebehay, 2016). Clustering of Static-Adaptive Correspondences for Deformable Object Tracking's (CMT) approach improved the proposition of utilizing a star-shaped object model by allowing an individual degree of displacement to each part, and, similarly to Hough Forests, object parts voted for the object centre, where CMT's distinguishing fact was that the voting was completed by object pairs.

Following the pipeline of classical object tracking algorithms a motion model would first be assumed, such as minor per-frame displacement, and be used to make a temporal-spatial prediction. Methods such as sparse optical flow (Lucas, Kanade, 1981) were commonly employed in classical generic object tracking

pipelines. Image features were then handled by utilising a hand-crafted feature detector and descriptor such as Orientated FAST and Rotated BRIEF (Rublee, 2011), which essentially utilized a FAST keypoint detector and a BRIEF descriptor, but where Orientated FAST and Rotated BRIEF extended the BRIEF descriptor vector to include rotation invariance. These keypoints and their associated descriptors were subsequently processed in the following algorithmic step to localize the object of interest, for instance by matching candidate descriptors to a database of the object using a nearest neighbour matching scheme. The final algorithmic step was to update the object model with the newly acquired information to keep the object model current, and to balance the trade-off between stability and plasticity as well as drift. CMT approached this by combing the temporal-spatial and image feature information in an adaptive-static approach, by giving preference to robust, statically matched information.

2.5.2 Connectionist approach

In contrast to the traditional approach to object tracking, where the building blocks of the tracker were well defined, the connectionist approach, which utilizes ANNs of varyingly interconnected layers and nodes, is somewhat less understood. State of the art object tracking algorithms have benefited enormously from the resurgence of ANNs, and, more specifically, CNNs and their newfound success in the application to high level image processing tasks such as Object Detection.

Similarly to Deep Neural Networks (DNNs) designed for the task of object detection and classification, object tracking algorithms have arguably benefitted primarily from the increased effectiveness of convolutional filters for the task of feature detection and description. The primordial difference between the connectionist approach and those used in previous state of the art object tracking methods, culminating arguably with CMT, is that convolutional kernel weights are learnt in a supervised fashion, offline. The features extracted from CNNs are interesting to visualize and it is noted that within the early convolutional layers that the kernels learn to behave similarly to classical feature detectors, in that corners in images and similar features are also found. Image features are of extreme importance when considering the tracking accuracy of object tracking algorithms, as will be illustrated in section 5, as, no matter how sophisticated the object model, if the features that are employed are not discriminative, any object model will fail. A simple object model however, when employed with discriminative features, can perform remarkably well.

The connectionist approach to the design of trackers focuses around 3 distinct stages (Bazzani, 2011): the who-, where- and why-pathways. The who-pathway is aligned with the classical view of image features and accumulates information that can be used to perform discrimination of the object and background, the where-pathway which is responsible for perceiving object location, and the why-pathway

is responsible for with the learning objective of the network. One such network, Fully-Convolutional Siamese Networks for Object Tracking (Bertinetto, 2017), which rose to dominance in the real-time challenge with the Visual Object Tracking challenge (VOT), employs a seemingly simple Siamese network that essentially performs a similarity measure between image patches with features provided by a CNN. The approach to similarity learning by Fully-Convolutional Siamese Networks for Object Tracking (SiamFC) is illustrated below in Figure 2.5.

The approach of SiamFC trains a function $f(z, x)$ that compares a template image z to the input image patch x, with the goal of this function as previously mentioned, to compute the similarity between the two images. The resulting measure should discern whether the image contains the same object by either providing a high score for a True Positive or low score other if the object is not deemed to be present, after which a thorough localization effort is completed in the event of a True Positive. The goal of the localization effort is to find the image patch in the input image that is most similar to the template z - a former representation of the object.

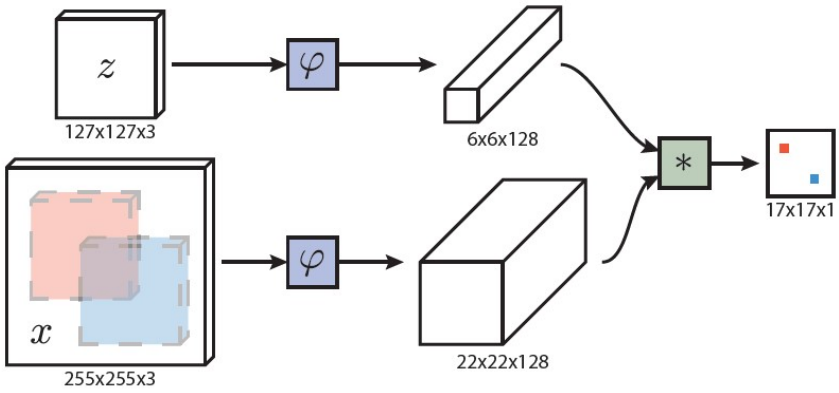


Figure 2.5: The Fully-Convolutional Siamese Networks (Bertinetto, 2017) developed for object tracking. The who and where pathway indicated by the Ø and * operator respectively.

3 Case study

The following chapter serves to cover in detail the algorithm that is proposed to be deployed to the embedded system, as well as an overview to modern approaches to handling computationally expensive algorithms.

3.1 Tracking algorithm

As mentioned previously, the generic object tracking algorithm Clustering of Static-Adaptive Correspondences for Deformable Object Tracking (Nebehay, 2016) employs a classical approach to object tracking. Classical object tracking methods that rely on hand-crafted algorithms to either handle image features or when modelling the object of interest. In this section, the theoretical approach of the Clustering of Static-Adaptive Correspondences for Deformable Object Tracking (CMT) algorithm is covered in depth and the fundamental computational steps are explained.

3.1.1 CMT

The algorithm CMT contributed to the field of generic object tracking by developing the Deformable Part Model for One-Shot Object Tracking (DPMOST). DPMOST is a part-based model builds off a star-shaped model, whereby the object is modelled with a set of interconnected object parts with a common anchor. As in all star-shaped models, the common anchor DPMOST revolves around is the object centre. Star-shaped models are attractive since they handle occlusion well, as illustrated in CMT's performance (Kristan *et al*, 2017) as not all object parts need to present for the object to be detected successfully. DPMOST further extends this attractive quality by handling object deformation in a principled manner, whereby object parts are proposed to be interconnected and can connect outlying parts through another member belonging to the part model. Highly displaced parts that serve to model severe deformations, can have their membership to the object of interest reinforced by other interconnecting parts. DPMOST can account for extreme deformation, as illustrated by Figure 3.1.

Formally, DPMOST builds an object model from a reference set of members or object parts, $\{x_1^t, \dots x_N^t\}$ obtained from an initialization image and accompanying bounding box b_1, with the reference appearance Z having been normalized with respect to the mean position in image coordinates. The so-called deformation threshold, that is assumed constant for each part in a bid to keep the approach simple, allows each vote for the common anchor point a certain amount of leeway. The set of correspondences denoted $L = \{ m_1, \dots m_n \}$ with m_n being a pair of object points (x_i^t, x_i^1), is a set of correspondences between the initialization image and the present image of interest.

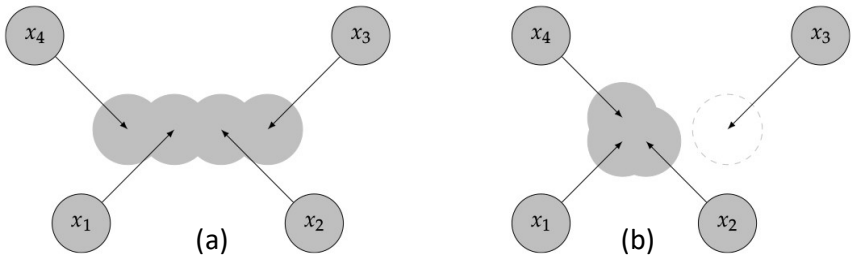


Figure 3.1: Illustration of the mediating properties of voted in CMT. In (a) the parts x_3 and x_4 are deemed to vote for the same centre, by having parts x_2 and x_1 reinforce their membership to the object model opposed to in (b) where part x_3 has severe deformation and no mediating part (Nebehay, 2016).

The objective at hand then becomes how a set of correspondences between the reference set and the current image of interest should be determined. The method employed will greatly influence the success of the tracking algorithm, as all information inferred about the object and its temporal behaviour will be based on the set of correspondences. CMT employs a two-step approach with the intention of increasing matching robustness by balancing the stability-plasticity dilemma.

Static correspondences are obtained by matching descriptors between the reference set, which are obtained from the initialization image, and the candidate set through the utilization of a similarity measure. Since this requires comparing every candidate descriptor vector against those in the reference set, computational efficiency is of great importance. Binary descriptor vectors, which is where floating point variables are avoided, present an attractive approach to the computational dilemma. Descriptors can be compared utilizing the XOR operator to compare two equal-length bit strings or to measure the distance between descriptor vectors and can be effectively computed on modern CPUs, utilizing for instance the SEE4.2 instruction set (Intel, 2007). As such we can avoid computing the floating point L2 norm, if for instance SIFT descriptors were employed in - a markedly costlier operation. The author of CMT elaborates on the choice of the matching scheme employed for descriptor matching and arrives at the conclusion of second nearest neighbour distance ratio (SNNDR) test

$$SNNDR(p(x_i)) = \begin{Bmatrix} NN(p(x_i)) \; if \; \frac{d(p(x_i),NN(p(x_i)))}{d(p(x_i),NN_2(p(x_i)))} < \gamma \\ 0 \; otherwise \end{Bmatrix} \quad (3.1)$$

The SNNDR is deemed (Mikolajczyk and Schmid, 2005) to be a robust method for determining the confidence a correct match has been made for a candidate descriptor $p(x_i)$ and its nearest neighbour. This completed by assessing the distance d between a candidate descriptor $p(x_i)$ and the nearest neighbour $NN(p(x_i))$, over a distance to the second nearest neighbour $NN_2(p(x_i))$. The intuition behind this ratio is that the lower the ratio, that must be below the threshold γ, the more confident the match. A larger denominator than numerator will identify that indeed the numerator presents a unique match, as the second nearest neighbour is not near the second nearest neighbour to the candidate descriptor. The application of the SNNDR will ensure highly discriminative matches will be established, rather than ambiguous matches when simply using a nearest neighbour matching scheme. It seems imperative at this moment to highlight the fact that in the design of CMT, the reference set of descriptors includes descriptors that belong to clutter - this is beneficial to the matching effort as the algorithm can disregard candidates that match to directly to clutter. Descriptors from the reference set are assigned class labels, foreground or background, and as such DPMOST will disregard matches that are classified to belong to background.

The second approach incorporated into CMT to identify *adaptive* correspondences between the current image and the reference set, is through the utilization of sparse optical flow. The principle intuition behind utilizing sparse optical flow (Lucas and Kanade, 1981) is for the algorithm to remain adaptive to temporal-spatial changes, and as such computes the displacement between keypoints from the previous frame to current the frame. The Forward-Backward Error (Kalal, 2010) furthers the logic of forward optical flow and calculates optical flow from the previous frame to current, as well in reverse. The difference between these two trajectories is then calculated for the specific keypoint and if the two trajectories differ by less than the specified threshold, they are identified as correct *adaptive* correspondences. While it may not be inherently clear, the keypoints that are tested in this spatial-temporal manner are exclusively those that belong to the object from the previous frame, or those keypoints that are *active.*

Finally, the object model needs to remain current throughout the tracking sequence. The model needs to adapt to changes in object appearance but also to remain stable such as not to suffer from drift, but also that we are able to redetect the object if absent for a period. Opposing other tracking methods that utilize a similar detection approach, CMT never modifies the initial reference set as these were constructed from extremely reliable information. Rather, two sets of correspondences are maintained and are fused to become the set L_*. Static correspondences L_S identified using the SNNDR are more robust, and those discerned as correct correspondences by using sparse optical flow are termed the adaptive correspondences L_A. The set L_* is constructed on a per-frame basis by fusing the two sets, L_A and L_S, and discards adaptive correspondences when the

static equivalent is available in its attempt to balance the stability-plasticity dilemma.

Once we have determined a set of correspondences between initialization and current information L_*, the next task is to process the correspondences to discern the resulting per-frame shift in object location, in-plane rotation and object centre.

Firstly, to quantify scale and rotation change, of which DPMOST accounts for when predicting object location, it was showed (Kalal, 2010) that scale could reliably be estimated through a pairwise measure

$$s = median\left(\frac{x_i^t - x_j^t}{x_i^1 - x_j^1}\right) \tag{3.6}$$

which leads to a similar measure for a rotation estimate

$$R = median\big(atan2\big(x_i^1 - x_j^1\big) - atan2(x_i^t - x_j^t)\big) \tag{3.7}$$

whereby the median rejects outlying predictions reliably, given that the count of predictions made by inliers is at least 50%. Once scale and rotation have been determined for the set L_*, we turn our attention determining a *consensus set* of votes for the object centre. The simple initial voting mechanism (Nebehay, 2016) is initially proposed as

$$h(m_i) = x_i^t - x_i^1 \tag{3.2}$$

where $h(m_i)$ simply represents translation between correspondence m_i of object parts x_i^1 in the initialization and x_i^t in the current frame t. We start to build a sense for the object location when the two correspondences m_i and m_j simultaneously vote for the centre location with the transitive operator $w\big(m_i, m_j\big)$

$$w\big(m_j, m_i\big) = \begin{cases} 1\ if\ ||\ h(mi) - h(mj)|| < deform\ thresh \\ 0\ otherwise \end{cases} \tag{3.3}$$

whereby we have *consensus* between correspondences of the object centre if the $L2$ norm is smaller than twice the deformation threshold. The aforementioned mediating property of Equation 3.3 is demonstrated by the mathematical transitive property of

$$w\big(m_i, m_j\big)\ and\ w\big(m_k, m_j\big) = w(m_i, m_k) \tag{3.4}$$

whereby correspondences m_i and m_k are mediated by correspondences m_j.

CMT approaches the computation of equation 3.3 from the approach of agglomerative clustering approach whereby we utilize the deformation threshold as a cut-off between differing levels of linkage, as illustrated in Figure 3.2. Scale and rotation invariant votes for the object centre μ are made by

$$\mu = \frac{1}{|L_w|} \sum_{(x_i^1, x_i^t)} (x_i^t - sRx_i^1) \tag{3.8}$$

where the difference in centre votes is mediated between votes, and subsequently object parts, by the deformation threshold.

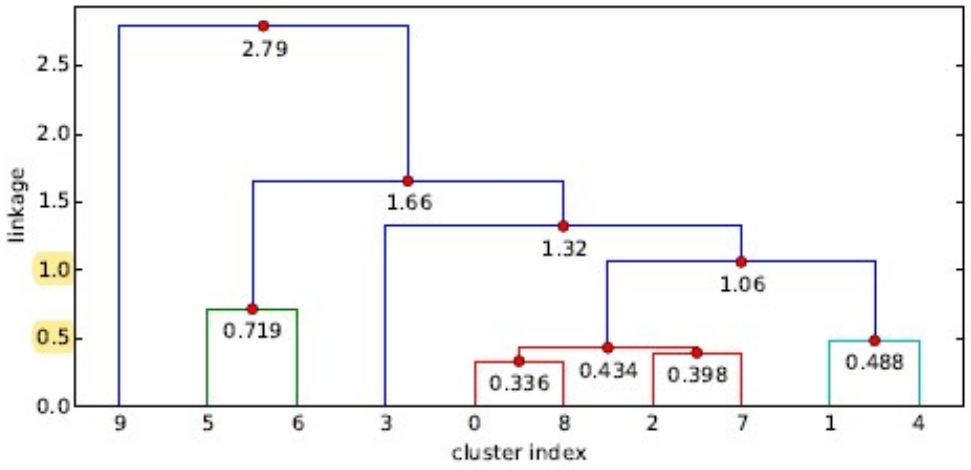


Figure 3.2: A graphical illustration of the agglomerative approach to clustering the centre votes, whereby the deformation threshold serves to discern the linkage cut-off between clusters of centre votes (Nebehay, 2016).

The consensus set L_w is then determined by taking the largest cluster of centre votes, after which the object centre is calculated. As a final reliability check (Nebehay, 2016) we employ

$$\varphi(L_w) = \begin{cases} 1 \; if \; |L_w| > \theta(\varphi) \\ 0 \; otherwise \end{cases} \tag{3.5}$$

where $\theta(\varphi)$ is a numerical value that is set experimentally.

3.1.2 Algorithmic overview

In order to clarify the computational process and how each of the previous sections are interlinked, the algorithmic psuedocode for CMT is presented below in Figure 3.3.

Initially we construct the normalized reference set of object *parts* Z by requiring an initialization image I_1 and bounding box b_1. We first proceed by extracting image keypoints in step 1, using a keypoint detector such as FAST that is implemented in OpenCV. The keypoints are then normalized, by the mean keypoint location, in step 2 to produce Z.

Descriptor vectors, such as a BRIEF descriptor that is similarly implemented in OpenCV, are then computed in step 3 for the keypoints extracted in step 1. Steps 4 and 5 once again detect keypoints, or parts, and compute their associated descriptors but for keypoints of the region outside of the bounding box b_1. This separate set of keypoints and descriptors that are inside and of foreground, or outside and of background, of the supplied bounding box b_1 are crucially separated and assigned an according class label for keypoint matching in step 8, as mentioned in 3.1.1. The reference set of *static descriptors P* utilized for matching is obtained in step 5, whereby the descriptors from step 3 are also added to *P*. The set of *static descriptors P* now contains descriptors from inside and outside of the bounding box b_1.

The repeating loop then processes the input images by first detecting keypoints over the whole input image and computing the accompanying descriptors in step 7. The implementation utilized is the same as in step 1. In step 8, the extracted keypoints are then matched, through the use of the accompanying descriptors, to the reference set *P* through OpenCV's implementation of nearest neighbouring matching. Within the algorithmic step 8, the matches are tested using equation 3.1 to produce the static correspondences $\mathcal{L}^s$. Step 9 produces the set of adaptive correspondences $\mathcal{L}^A$ by computing the optical flow, once again utilizing OpenCV's implementation thereof. Optical flow for the previous *active keypoints,* $\mathcal{L}^{t-1}$, is calculated both forward, using the current input image, as well as in reverse from the current image to the previous image I^{t-1}. After passing the test on the Forward-Backward Error, the adaptive correspondences are obtained. These two sets of candidate correspondences are then fused by the rule mentioned in Section 3.1.1, of inclusion with regards to static and adaptive correspondences, to produce $\mathcal{L}^*$.

The algorithm then proceeds to make estimates of scale and rotation in step 11 and 12 respectively, using this fused set $\mathcal{L}^*$. This is completed through utilizing Equation 3.6 and Equation 3.7. Once estimates for scale and rotation have been

determined, they are employed in step 13 to complete transitive consensus utilizing Equation 3.3. Transitive consensus allows outlying object parts to be mediated by other parts and as such this operation results in the set of *consensus set* $\mathcal{L}^W$.

Line 14 the presents the algorithmic step "disambiguate", whereby we further attempt to refine the consensus set $\mathcal{L}^W$ by attempting to include any keypoints, or object parts, that were not matched in step 8.

Input: $I_1, \ldots, I_T, b_1$
Output: $b_2, \ldots, b_T$
1: $\mathcal{L}^1 \leftarrow \text{detect\_parts}(I_1, b_1)$
2: $Z \leftarrow \text{normalize}(\mathcal{L}^1)$
3: $P \leftarrow \text{compute\_descriptors}(I_1, \mathcal{L}^1)$
4: $\mathcal{L}^- \leftarrow \text{detect\_parts}(I_1) \setminus \mathcal{L}^1$
5: $P \leftarrow P \cup \text{compute\_descriptors}(I_1, \mathcal{L}^-)$
6: **for** $t \leftarrow 2, \ldots, T$ **do**
7: $C \leftarrow \text{compute\_descriptors}(I_t, \text{detect\_parts}(I_t))$
8: $\mathcal{L}^S \leftarrow \text{match}(C, P)$
9: $\mathcal{L}^A \leftarrow \text{optic\_flow}(I_{t-1}, I_t, \mathcal{L}^{t-1})$
10: $\mathcal{L}^* \leftarrow \mathcal{L}^S \cup \mathcal{L}^A$
11: $s \leftarrow \text{estimate\_scale}(\mathcal{L}^*, P)$
12: $\alpha \leftarrow \text{estimate\_rotation}(\mathcal{L}^*, P)$
13: $\mathcal{L}^\omega \leftarrow \text{transitive\_consensus}(\mathcal{L}^*, Z, s, \alpha)$
14: $\mathcal{L}^t \leftarrow \text{disambiguate}(C, P, Z)$
15: **if** $\phi(\mathcal{L}^\omega)$ **then**
16: $\mu \leftarrow \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i^\omega$
17: $H \leftarrow \text{similarity\_transform}(\mu, s, \alpha)$
18: $b_t \leftarrow Hb_1$
19: **else**
20: $b_t \leftarrow \emptyset$.
21: **end if**
22: **end for**

Figure 3.3: Algorithmic overview of CMT, denoting the entire computation process from initialization till termination of the for loop (Nebehay, 2016).

This action of "disambiguation" is done by transforming the reference set Z through the scale and rotation heuristic and attempting to match the keypoints of step 7 to this transformed set. OpenCV's nearest neighbouring matching is nce again called upon for the matching functionality. The intention of the ambiguation step is include incorrectly rejected keypoints in step 7 such that the estimation of adaptive correspondences in the forthcoming input are more accurate (Nebehay, 2016).

Post completion of this operation, step 15 establishes whether the result of the consensus was feasible according to the prior count in the consensus set. If step 15 is passed the current bounding box, object centre and subsequent similarity transform is calculated. In the result of a false output from line 15, no bounding box is generated.

3.2 Computation

Due to the high dimensional space that computer vision pipelines usually occupy, the associated computing overhead is vast and puts great demand on the chosen hardware platform. A careful codesign however, whilst being mindful of both hardware and software, can yield promising results especially with the significant computing power that has become more accessible to developers as of late. The following section serves to highlight the industry standard approach to dealing with the high dimensional data that computer vision presents.

3.2.1 Graphics processing units

Since the incarnation of NVIDIA (1999), industry as well as mainstream users have been increasingly exposed to computing platforms that contain Graphics Processing Units (GPU), which too have been increasingly applied to general computational problems, known as general-purpose computing on GPUs (GPGPU). GPUs however were not originally designed for the task of general computation such as CPUs, but rather for processing pixels for video output. Due to requirement for GPUS to processes high bandwidth input such as video, they tend to feature higher core-counts when compared to CPUs, with associated co-processors such as Single Instruction Multiple Data (SIMD) processors being present in higher counts too. GPU compute cores further differ from CPU cores in that more threads can be dispatched allowing more computation to occur concurrently, but it is also important to note that modern CPUs contain vectorized instructions sets such as Intel's AVX for data parallel computations. In general GPUs dedicate more transistors to ALU units that support floating point operations, and with each computing core having less cache memory than a CPU core and with less emphasis on flow control. Fundamentally, GPUs are clocked at lower frequencies than CPUs but feature higher memory bandwidth – an indication that they are designed for data parallel computations.

Industry and researchers soon realized the applicability of GPUs to general computing and as such one popular framework that came into fruition was NVIDIA's Compute Unified Device Architecture (CUDA) that allowed users to offload computationally expensive workloads to various GPUs and enjoy higher throughput. Since the introduction of CUDA, GPUs have seen application to many fields such as Molecular modelling (Stone *et al*, 2007) or in computational finance (Grauer-Gray, Killian, Searles and Cavazos, 2013). The common a common theme

between all these applications is the vast amount of data the needs to be processed. Fortunately, many computer vision tasks are highly data parallel, in that the same operator is carried out many times on different data points.

3.2.2 Accelerating computer vision

A common task in computer vision, that is highly data parallel, is the computation of nearest neighbour. In the process of calculating the nearest neighbour, a single input d_n and is to be matched to a training database t. The candidate vector is used to rank the entries of training database by a measure of their proximity to the candidate vector, through a measure such as the L2 norm. In the case of CMT, the second nearest neighbour is required and such the two entries that are in closest proximity to the candidate descriptor are returned. Algorithmically this is completed by: for the input d_n we iterate over each descriptor $t_{1,\dots n}$ in the database and perform a similarity measure between the current candidate descriptor from t_n. We rank the results in a descending list and store the two best results, presenting computational complexity $O(ndt)$. The operation whereby we need to compute the distance of candidate to the entire database is very suited to a SIMD processor, as the same instruction is repeated on multiple data points. Therefore, the database of descriptors can be operated on simultaneously and a decrease in computation time when utilizing a SIMD processor can be expected.

Fortunately, many computer vision functions present a similar level of data-parallelism such as the convolution filter in Equation 2.1. The convolution filter can be interpreted as a “primitive” image processing task, as it comprises of a single matrix multiplication operation. In Figure 3.4 we see that “primitive” image processing tasks can see decrease in computation time of 30 times. Another operation that is data parallel and can benefit from a magnitude decrease in computation time is keypoint detection, once again as illustrated in Figure 3.4.

In short, computer vision tasks present a challenge when it comes to real time processing due to the vast amounts of complex data. Using GPUs and efficient software design, we can expect to accelerate highly data parallel tasks.

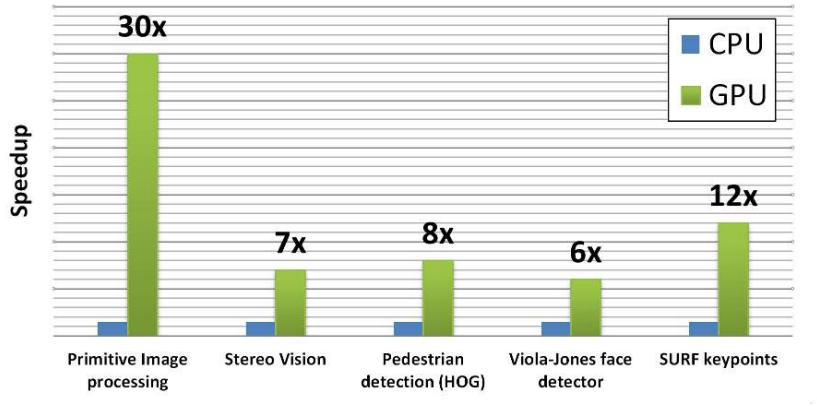


Figure 3.4: A summary of generic functions used within common computer vision pipelines and their indefinite estimated speed-ups, noting that keypoint detection, though SURF keypoints, can be sped up by an order of magnitude utilizing GPU acceleration (NVIDIA).

3.2.3 Software & heterogenous computing

Building on the promise of accelerating data-parallel computer vision tasks of the previous section, it seems pertinent to investigate how it is possible to effectively exploit on-board hardware such that we achieve high levels of hardware utilization. Figure 3.5 illustrates three cases where a heterogenous computing platform is utilized in different manners. Extending the No Free Lunch theorem to computing processors, a throughput optimized computer vision application will rely on both a CPU and GPU, in a common heterogenous platform, to achieve real-time processing and the highest levels of device utilization, with each processor being suited to a specific function.

A CPU will in theory be more suited to handling memory allocation and interfacing with various sensors due its broad spectrum of instruction sets, whereas a GPU will be more suited to handling data parallel computations. As such, we can design our algorithms to run across various processors and allocate the suitable functions to each and achieve the highest efficiency execution model by utilizing a concurrent execution model - whereby both processors are simultaneously allocated and processing data, illustrated below in the diagram third from left. Open Compute Language (OpenCL) is another framework, such as NVIDIA's CUDA, for utilizing and executing programs across heterogenous computing platforms.

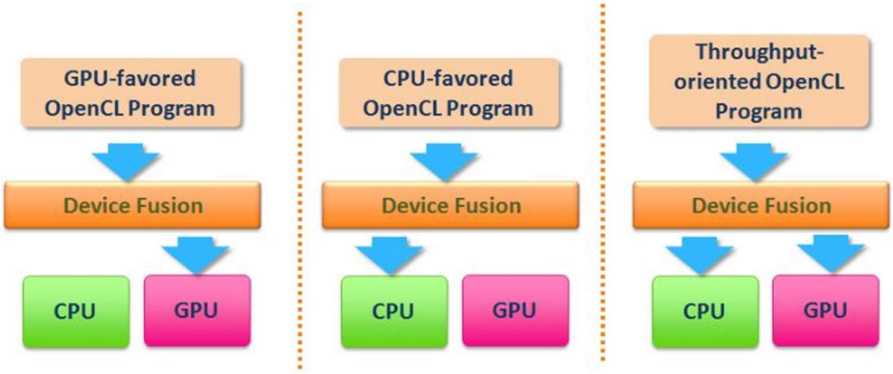


Figure 3.5: A throughput optimized program whereby we note both processors are allocated, increasing the SoC utilization when compared to only having a single processor allocated (MediaTek – available at http://cdn-cw.mediatek.com/White%20Papers/MediaTek_CorePilot%202.0_Final.pdf).

3.3 Hardware platform

Many implementation mobile platforms are suitable for the case of implementing a generic object tracking algorithms. An implementation platform should aim at balancing factors such as cost, electrical power consumption, ease of use and being suitable for the task, as well as possibly highlighting important factors for future research.

3.3.1 General-purpose single board computer

Since the goal of this research is into the *feasibility* of generic object tracking on a single board computer, a hardware platform that supports lightweight distributions of desktop operating systems would be most apt. A platform such as Raspbian, which is based off-of Debian, will allow the use of standard software libraries such Open Computer Vision (OpenCV), languages like C++ and compilers such as GCC. This will allow rapid development and testing of algorithms, without having to resort to cross-compiling or lower level implementations of algorithms, and will enable minimalizing the amount of time spent debugging - essentially enabling a development environment like those of a desktop, but with lower computation power on hand. As such algorithms will be able to be directly tested on such a mobile platform, allowing an insight into computational demands and hardware specific bottlenecks before moving to a lower-level implementation if need be.

Recently there has been a surge in the popularity of general-purpose single board computers such as the Raspberry Pi computing platform. Platforms such as these present interesting prototyping platforms to test methods like heterogenous

computing and GPU acceleration since most of the single board computers have on-board mobile GPUs that support frameworks such as OpenCL, as well as having desktop features like supporting CPU multithreading through Threaded Building Blocks. Single board computers also have adequate RAM available, an important hardware specification for testing large-model CNNs (Velasco-Montero, 2016).

Whilst we do not expect performance to rival that of dedicated hardware, such as dedicated GPUs found in desktop platforms, SoC's resources that are similar in form and hardware, like the Raspberry Pi, present a complete computing platform that presents desktop functionality, on one all-contained platform, at power levels an order of magnitude lower than that of desktop platforms. This type of hardware platform that is to be investigated is of similar specification to those found in modern smartphones, making the investigation more so compelling due to the current abundance of smartphones.

4 Use case

The following section serves to outline the use case for the experiments carried out in this research and motivates, where necessary, decisions that were made with regards to the hardware platform chosen for implementation, the choice of tracking algorithm as well as a dataset for testing the chosen implementation configuration.

4.1 Target use case

As mentioned in the introductory section to this thesis, drones are proposed to be used in the aid of the calibration of heliostats. As such, the implementation platform will certainly need to be a mobile hardware platform that does not consume vast amounts of electrical power which would significantly reduce the already limited flight time of UAVs. Since the environmental setting will be outdoors and involve the tracking of large static objects, we need to use a suitable dataset to simulate this use case, but also a dataset that enables us to compare results to highlight progress. The dataset and subsequent analysis should concentrate on outdoor scenes with occlusions and tracking sequences that feature similar objects, as in an CSP plant there will be many seemingly identical heliostats.

4.2 Algorithm

This intended use case of this research is into the feasibility of object tracking on a general-purpose single board computer, with the specific application to UAVs and for them to track generic, deformable objects. In the following section, the choice for the particular algorithm is motivated and the limiting factors of state-of-the-art approaches are highlighted.

4.2.1 Algorithm choice

CMT was highlighted as the algorithm of choice due to it scoring well in the 2015 VOT challenge, as well as not utilizing the computationally expensive convolution filter for extracting features. The algorithm uses a simple star-shaped model to represent the object of interest, which is configurable in the number of parts that are used to represent the object of interest. We further motivate the choice of CMT as it scored very highly in the 2017 VOT challenge in terms of tracking performance in situations that involved occlusion - a situation very common in UAV-based tracking.

A benchmarking effort was recently undertaken by Velasco-Montero et al (2016) to investigate the rates of inference achievable for common DNNs that are utilized for object detection. While we note that Velasco-Montero utilized software packages originally designed for desktop utilization such as Tensorflow (Abadi et al, 2016) many of these software packages have lightweight implementation such as Tensorflow Lite. These packages, however, are not as thoroughly supported as OpenCV is for the specific hardware, as we see OpenCV has the highest average inference rate, as well as highest accuracy achieved in benchmarking when the C++ API is utilized in Velasco-Montero's investigation.

The benchmarking effort however highlights a common theme - common DNN architectures are not yet feasible for mobile implementation when we have the constraint of real-time, on purely CPU based execution on a device such as a Raspberry Pi 3. Deep neural networks designed for mobile devices (Iandola et al, 2016) do attain impressive 5 FPS, and this is of particular interest to this research as the state-of-the-art real-time approach SiamFC utilizes a similar DNN to extract features for its similarity measure function. SqueezeNet (Iandola et al, 2016) was designed to emulate AlexNet's accuracy, the latter being the backbone that SiamFC used to construct its feature maps, but with a highly reduced parameter count and model size. Essentially this implies that on a Raspberry Pi 3 we could extract features from our input image at 5 FPS and feed it into the SiamFC network, obtaining at most 5 FPS, if the rest of the network took negligible time to process the features. Attaining 5 FPS inference rate while impressive is simply not up to the task of real-time tasks. The remainder of the SiamFC network would occupy a significant portion of processing time, as we would be completing similarity measure, and as such in reality an inference rate of far below 5 FPS would be realistic. In summary, it is proposed that DNNs are not feasible for the task of real-time mobile inference when utilized for the task of generic object tracking, with further motivation stemming from the issue that the task of real-time object tracking inference remains challenging for desktop hardware.

The ARM Compute Library does however present an alternative to either of the frameworks utilized in the Velasco-Montero's benchmarking and is a fruitful topic further research effort. The library consists of heavily optimized, low-level implementations of the fundamentals of machine learning and Computer Tasks, such as the convolutional filter, SVMs and all of the CNN building blocks.

Whilst in CMT an initial reported inference rate of 10 FPS on desktop hardware, the quoted rate was not it is optimum configuration for high inference rates. The algorithm was configured by the author of CMT such that it attained the highest possible tracking effectiveness, but not the highest computational efficiency. Figure 4.1 highlights that the investigation conducted by in CMT on the possible configurations of detectors and descriptors, which seemingly reach an asymptote for a certain success rate of at an average overlap per-sequence. This can be

interpreted as: for a success rate of 0.6, or 60%, of tracking sequences in the testing dataset, most of the configurations of CMT averaged between 0.1 and 0.2 per-sequence overlap, or the algorithmically produced bounding box overlapped between 10 and 20% with the ground truth. However, noting that 60% of the configurations did not average over 20% overlap for all sequences, it is promising that many configurations are possible with minimal effect on tracking accuracy; whereas certain detectors and descriptors are vastly superior in computational efficiency.

CMT thus presents an interesting case for efficient mobile inference, noting, too, that no GPU acceleration has been utilized in the original approach. As with many tasks such as object tracking, a trade-off will need to be balanced between speed, accuracy and cost.

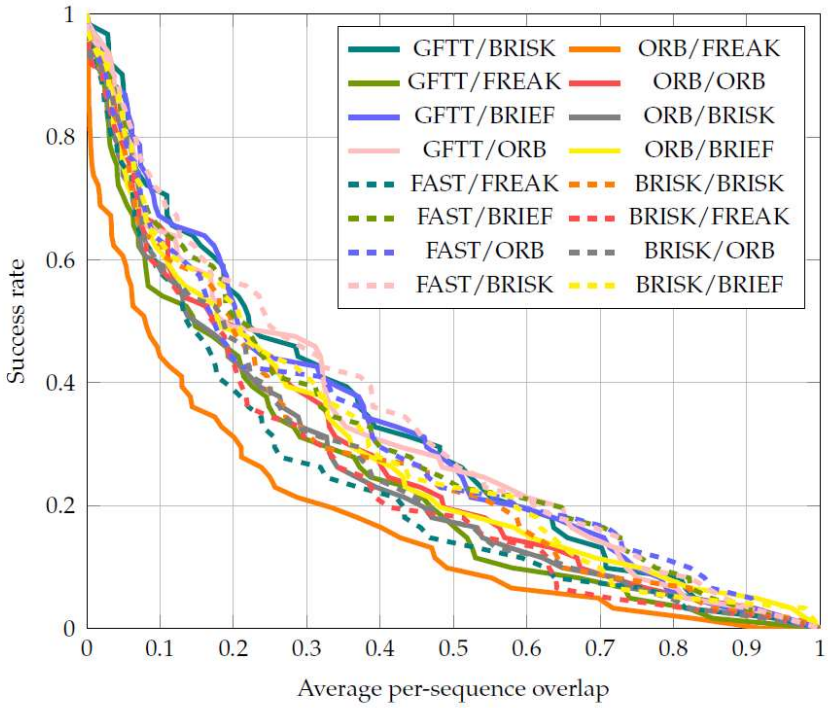


Figure 4.1: Illustration of the possible configurations of part detectors and descriptors, where it is noted that compact binary descriptors (32-byte BRIEF) outperform more larger descriptors (64-byte FREAK) highlighted by the combination of the GFFT detector and BRIEF descriptor (Nebehay, 2016).

4.3 Hardware

As previously mentioned, the specific hardware platform to investigate is a Single Board Computer that has similar attributes such as form factor, power consumption and hardware resources as those of the Raspberry Pi™ 3 computing platform. The following section serves to motivate the choice in this regard.

We propose to investigate the ASUS Tinkerboard™ for the task of inference of a generic object tracking algorithm, with the aim of achieving real-time performance. Whilst real-time is a vague description and most certainly varies between uses, it can be loosely defined as the situation where the data processing must be completed at rate faster than the rate at which the input is received (VOT, 2015). Whilst this definition is still not completely concrete in its definition, as sensors capture images differing frequencies and resolutions, we propose for the average rate to be 20 Hz with the chosen dataset.

The ASUS Tinkerboard™ is a relatively low-cost hardware platform when compared to an embedded platform such as the NVIDIA™ Jetson, whilst being considerably more computationally powerful and, as such, more suited to the task of real-time object tracking than the popular Raspberry Pi™ 3. Whereas the Jetson provides a high-end GPU computing solution and the Raspberry Pi™ an entry level device best suited for a hobbyist, the Tinkerboard™ presents an attractive solution by providing a quad-core CPU ARM™ Cortex A17 clocked at 1.8 GHz and 6 core ARM™ Mali T760 mobile GPU, on the Rockchip™ RK3288 system on a chip. It presents a cost-effective hardware platform to investigate embedded GPU accelerated computation of computer vision tasks; in contrast to the offering from NVIDIA, as it supports the Open Computer Vision (OpenCV) library's minimum OpenCL framework requirement. The Raspberry Pi™ computing platform was ruled out due to the Broadcom™ GPU not supporting OpenCL and thus preventing any possibly heterogenous computing.

Furthermore, the ARM™ Cortex A17 processor features an advanced NEON Singe Instruction Multiple Data (SIMD) coprocessor as well as VFPv4 instruction set for floating point acceleration, where the former hardware accelerator should bring significant performance increases due to data-parallel tasks being computed efficiently using a SIMD coprocessor.

In summary, the ASUS Tinkerboard™ was chosen predominantly due to the board being a heterogenous computing platform that supports a desktop-like operating system that allows rapid prototyping and testing, both in terms of software libraries commonly utilized in computer vision research as well as hardware that is used for efficiently dealing with, and possibly accelerating, computer vision tasks.

4.4 Dataset

Finally, the dataset that will be employed to evaluate the on-edge performance of the object tracking algorithm is of great importance, as any insight gained will be highly dependent on the dataset. The dataset needs represent the task which we seek to utilize the algorithm for in practice and needs to challenge the algorithm accordingly with scenes that we might encounter in practise. One such example is vast scale change. Without the dataset being representative of the scenes encountered in practice, we cannot make any realistic assessment of the performance of the algorithm and thus cannot make an estimation of how it will perform for our specific use case.

The dataset chosen to evaluate the on-edge performance of CMT on the Tinkerboard™ is that which is employed by the author of CMT, a dataset compiled by Thomas Vojir. The dataset will allow comparison between the inference rate achieved in practice with that of the original implementation of CMT, on desktop hardware, and with analysis concentrating on outdoor scenes with large amounts of background detail as in Figure 4.2, occlusion as in Figure 4.3 and the redetection as in Figure 4.4, as well as tracking similar objects. As such the dataset will be trimmed to a smaller size and by doing so will allow us to evaluate the performance quickly. The final dataset[1] employed will be the reduced dataset compiled by Vojir, consisting of the 22 tracking sequences. Primarily, the main aim of the research regards the possible on-edge inference rates of CMT rather than its accuracy, as that should carry over between mobile and desktop implementations. However, a balance will still need to be struck in the investigation between accuracy and inference rates as the two parameters most likely share a distinct coupling.

Figure 4.2: Frame 1 from the sequence "Track" that serves to represent outdoor scenes with high level of detail. This sequence is especially challenging in terms of the amount of detail, highlighted in the forthcoming section 5.3.

Figure 4.3: Frame 1 and 700 from the sequence "Person partially occluded" that serves to represent scenes with occlusion.

Figure 4.4: Frame 507, 558, and 570, clockwise from top left, from the sequence "Car 2" that serves to represent outdoor scenes with high level of detail, redetection and similar object.

---

[1] Available at http://cmp.felk.cvut.cz/~vojirtom/dataset/

5 Experiments

In the following sections, the method employed to evaluate tracking performance, as there are multiple in the field of object tracking research is first. In section 5.2 the baseline performance is established first on a desktop platform, for comparison, as well as on the mobile device. The most time-consuming functions of CMT are also identified. Section 5.3 covers the optimization regarding image features and most importantly feature descriptor choice, on the mobile platform. Section 5.4 covers the utilization of hardware acceleration on the mobile device and section 5.5 covers the theory behind a heterogeneous execution. In section 5.5 we encounter a hardware and software barrier on the mobile platform and return to the desktop platform for the remainder of section 5.5 and the section 5.6. The final section, section 5.6, covers concurrent heterogenous execution of the tracking algorithm on the desktop platform – the theoretical limit of achievable performance.

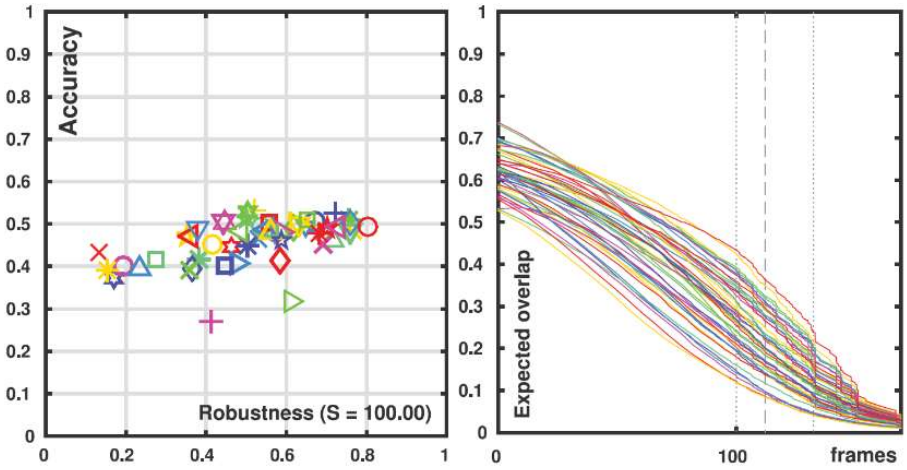
5.1 Evaluation method

5.1.1 VOT

Introduced in 2013, the annual Visual Object Tracking Challenge aims to consolidate active research in the field of Object Tracking, by providing standardized performance measures and an evaluation toolbox that seeks to automatically analyse trackers submitted to the challenge in an unbiased manner. Compared to other benchmarks, the VOT evaluation methodology differs in that once a tracker fails, it is reset, and the tracker effort continues. This was found to be of benefit in an unbiased tracking evaluation by Cohevin (2014), as it was identified that accuracy and robustness were negligibly coupled. This led to the utilization of both measures, by the VOT committee, where accuracy is the average overlap measured in frames in which tracking was successful, or an overlap between the ground truth and algorithmic output greater than 0.5, and robustness is a measure that accumulates tracker failures in a sequence and reinitializations. Accuracy-robustness plots, where a data entry is plotted with robustness on the x-axis and accuracy on the y-axis, are utilized to visualize the ranking of a specific tracker compared to other state-of-the-art methods but are not primary measure of absolute performance. Expected average overlap (EAO) is employed as the primary measure for the performance of tracking algorithms, being stricter than average overlap (AO) as shown in the 2016 VOT challenge (Kristan, 2016). EAO can be summarised as the average overlap between algorithmic output and ground truth bounding boxes, normalized with respect to the length of the sequence, for sequences of increasing length. The plot is, therefore, the increasing sequence length on the horizontal axis and the average

overlap for the specific sequence length on the vertical, illustrated in Figure 5.1 along with an AR plot on the following page.

Furthermore, the VOT challenge provides a dataset is utilized for evaluating submitted trackers. The dataset is concentrated on diversity and as such aims to be representative of a wide spectrum of tracking scenarios, rather than pursuing an arbitrarily large dataset. Importantly, each frame is labelled according to a class of established tracking scenarios; for instance, being challenging in terms of illumination and scale change. A subsequent figure, illustrated on the following page in Figure 5.2, can be produced that is both easily interpreted and extremely useful when comparing trackers – a comparison that reinforces the theorem of No Free Lunch (Wolpert, 1997), stating that no singular tracker will be superior in all respects. A recently introduced performance measure, by the VOT challenge with the goal of normalizing a submitted trackers' performance results relative to the hardware the results were generated on, is the effective filter operations (EFO). Essentially the measure attempts to account for the speed of a specific hardware platform by measuring a set of operations on the platform and normalizing any frames per second processing rates by the measured time; thus, resulting in a normalized measure.



(a) (b)

Figure 5.1: An AR (a) plot alongside an EAO (b) plot – illustrating the accompanying and primary performance measures for the VOT challenge, respectively (Kristan et al., 2017).

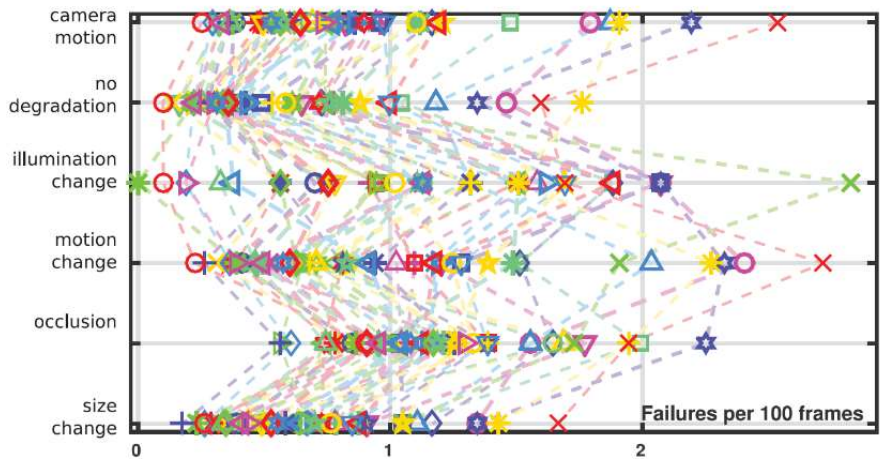


Figure 5.2: The failure rate of state-of-the-art tracking methods for various scenarios. There is a not a single method that performs well in all the scenarios (Kristan et al., 2017).

However, due to variation in software packages used in developing tracking algorithms as well as hardware, this measure cannot accurately represent the speed of the tracking algorithm, but it is a notable measure introduced by the VOT challenge as it represents the importance of speed.

The VOT challenge has made a remarkable impact on the quality of developed algorithms and the assessment of their performance, as well as encouraging quality, novel research by defining a tracker as state-of-the-art if it is above a certain performance threshold; thus, discouraging every paper to quote top performance results which skews the concept of state-of-the-art.

5.1.2 Measures

Evaluating object tracking algorithms in a fair and reproducible manner poses a lasting challenge. In order to automatically evaluate the success of the algorithm, a labelled dataset with annotated ground truth data is required. More specifically, in one-shot objecting tracking, the ground truth needs to be presented as a bounding box encompassing the object - denoted by the pixel location of the top left corner as well as the associated height and width - in each frame of the sequence. However, this comparison of algorithmic output to ground truth data may not be as simple as it is deemed to be on first impression and, as such, the following chapter servers to define the measures employed.

5.1.2.1 Per frame

As simple as denoting the object of interest with a bounding box may seem, it allows the annotation process of a large dataset's individual frames to be feasible when compared to other annotations such as a complex, multiple-variable pixel-wise segmentation, and remains the state-of-the-art approach. While the bounding box may present a simple and effective method to highlight the object of interest's position in the specific frame, it is not without its associated drawbacks - such as requiring a trade-off between wholly containing the object of interest whilst including the least possible amount of background pixels. As noted in CMT (Nebehay, 2016), the situations encountered in practice could be: wholly encapsulating the object with no regard to background inclusion, a bounding box only encapsulating pixels belonging to the object of interest, or a trade-off between these two extremes, which is most commonly employed. However, since the chosen method of representation incorporates a flexible definition for the bounding the object of interest, in the ground truth data and the algorithmic output, this metric cannot be used to assess the accuracy of the tracker and should be used to assess the robustness, as discussed by Nebehay (2016).

The method used to compare the algorithmic and ground truth bounding boxes can be traced to Jaccard (1912), where the overlapping area of the two bounding boxes is measured. This method is attractive since the measure is normalized between 0 and 1 and as such does not introduce significant bias into its measure when compared to other measures such as a centre error measure. A centre error measure introduces significant complication in its interpretation as the error is, for example, dependent not only on the size of the image but the size of the image too and is difficult to normalize. In order to negate the ambiguity inherent in ground truth data - that being the ratio of pixels belonging to the object and those to the background - it is necessary to introduce the terms true positive, false negative, false positive and true negative. These terms are borrowed from binary classification and are employed to determine whether a data point (or data points) belongs to either of the two classes in binary classification. In our case, the objective is to compare the output of the algorithm to the ground truth data in terms of the location of the object of interest. In order to compensate for the ambiguity in the ground truth data, we define the term true positive (TP) to be

$$TP = \begin{cases} 1 \; if \;\; \propto > \emptyset \\ else \; 0 \end{cases} \tag{2}$$

and noting that overlap measure $\propto$ may not be defined, either if there is not algorithmic output or the object is not present in the current frame, as indicated by in CMT (Nebehay, 2016). A false negative (FN) is defined when either the algorithm fails to produce output or the overlap measure $\propto$ is not greater than the threshold, when the object is present in the frame

$$FN = \begin{cases} 1\ if\ \propto \leq \emptyset\ \cup\ (A_{out} = \emptyset\ \cap GT\ != \emptyset) \\ 0\ else \end{cases} \quad (3)$$

Nebehay fully defines all eventualities, but since they are not employed in the measures in this thesis, the reader is referred to Nebehay (2016).

5.1.2.2 Accumulated

Since we are attempting to evaluate the success of the tracking algorithm over an entire sequence of frames, we need to define measures that can be employed to assess the success of the tracking algorithm over the entire sequence, which can be done as an *accumulation* of the per-frame measures. To measure the success of long-term trackers, noted by Nebehay (2016) as the algorithms that can correct themselves automatically after failure by re-detecting the object of interest, the concept of recall is defined as

$$recall = \frac{\sum TP_i}{\sum TP_i + \sum FN_i} \quad (4)$$

accompanied by the measure precision, defined as

$$precision = \frac{\sum TP_i}{\sum TP_i + \sum FP_i} \quad (5)$$

which differs in recall by providing a metric strictly when the algorithmic outputs a prediction. Nebehay (2016) provides further insight into the measure of recall as being the measure of many elements from the relevant population recalled; whereas precision estimates how many elements that have been retrieved are relevant. Nebehay continues by stating that a compromise exists between the two measures and that the balance between them being directed by the internal threshold of the algorithm on the prediction confidence. A higher threshold on the required prediction confidence will yield a more conservative behaviour, but a lower threshold will not increase the recall - as in binary classification - as the prediction is made with a bounding box, not a class label.

In order to graphically display the results of the discussed accumulative measure, a success plot can be interpreted as

$$S(q) = 1 - ECDF(q), \quad (6)$$

which is related to the empirical distribution function by the above identity, interpreted as an empirical tally of those measurements above a certain specified value; which in our case is how many sequences are above a certain recall value. In order to easily interpret the success plot, the area under the curve (AuC) is

utilized, which is equivalent to the mean of the individual measure - in this case, it is recall - a higher AuC indicates a higher recall rate on average.

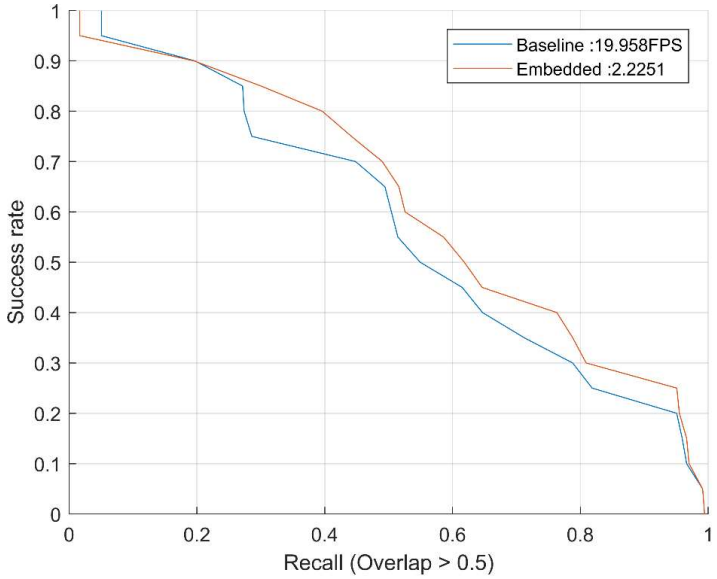


Figure 5.3: Success plots of CMT, when inferred in the author's original configuration of a FAST keypoint detector and BRISK descriptor. The 'baseline' curve belongs to the implementation on a desktop platform whereas the 'embedded' curve belongs to the mobile device. We note a tenfold difference in FPS achieved.

5.2 Baseline

In order to establish a baseline FPS inference rate, as well as tracking accuracy, CMT's performance was first evaluated on a desktop platform. This served as a benchmark to which the embedded board could be compared. If the FPS inference rates achieved on the embedded platform eclipsed this baseline, significant progress in embedded performance would be highlighted.

5.2.1 Desktop

The original configuration of CMT, as released by the author on their Github[2] repository, utilizes a FAST feature detector with accompanying BRISK descriptors.

[2] Available https://github.com/gnebehay/CppMT

OpenCV 3.4.3 was built with CMake in its standard configuration with no hardware specific optimization flags specified. The desktop hardware platform most importantly comprised of an Intel i7 7700 HQ CPU with 16GB of system RAM. The success plot is illustrated on the previous page in Figure 5.3 with the average inference rate, in processed frames per second over the entire dataset by Vojir, highlighted.

5.2.2 Embedded

The same test was then performed on the embedded platform where it should be noted that OpenCV, when built with CMake in Linux, as standard detects relevant hardware and will automatically enable optimization flags when building and as such the arguments

$$-D\ ENABLE\_NEON = OFF$$

$$-D\ ENABLE\_VFPv3 = OFF$$

were passed to ensure the OpenCV library was not build with the specified hardware accelerations for the embedded platform.

5.2.3 Embedded baseline performance

Illustrated on Figure 5.3, is the success plot of CMT on both the embedded and the desktop platforms. We note the large discrepancy between the two hardware platforms - this can be attributed to a variety of differences between the two platforms which is summarized in Table 5.1. Notably, the desktop platform has twice the core-count that each operate at twice the rate, as well as 8-fold the amount of available L1 cache memory, compared to the mobile platform. The last attribute is crucial as we infer fewer cache misses when computing the algorithm – cache misses being when we exhaust the available cache memory and need to access the relevant instructions or data from other cache levels or even possibly RAM. By having more available L1 cache memory, this allows the CPU cores to spend less time accessing slower-access memory and more time computing. Of equal importance is that the desktop platform has advanced system cooling compared to the embedded platform having simple passive cooling only.

Table 5.1: CPU hardware difference between the embedded and desktop platform. Whilst arguably not vastly different in terms of core count and frequency, the L1 cache memory size differs on one order of magnitude.

Hardware element	Desktop	Embedded
CPU core	8	4
CPU clock rate	3.6 GHz	1.8 GHz
CPU L1 cache	256 KiB	32 KiB

Figure 5.4 is a distribution of time per function. Referring to Figure 3.3, the entry "Consensus" in Figure 5.4 refers to the algorithmic step 13 in Figure 3.3. Similarly, for the entries "Track", "MatchLocal", "MatchGlobal", "Fuse" and Estimate" in Figure 5.4 correspond to the algorithmic steps 9, 8, 14, 10 as well as 11 and 12 combined, in Figure 3.3. Finally, the entries "Describe" and "Detect" in Figure 5.4 refer to the single algorithmic step 7 in Figure 3.3. The entry "Detect" is the second argument to the algorithmic step 7, and a separate algorithmic step but has been group together for simplicity.

The values for each function were averaged over each frame over the entire dataset, and, whilst not being the ideal approach to the situation as each sequence will present different overheads per function, it is an effective method to generate a general understanding of the computationally demanding functions or image processing tasks. A more detailed analysis of function behaviour for a specific scene is conducted in section 5.3.

We note that the majority of the time is spent matching candidate descriptors to the database and computing the descriptors for each keypoint, both being a function of the count of keypoints, which illustrates that an excessive count of keypoints was generated by the FAST feature detector. Cache missing may be occurring when the candidate descriptors are attempted to be matched to the database, as the entire database will likely not fit into cache memory. Slower memory access may need to be made to access the remainder of the database of descriptors that did not fit into cache memory to complete the nearest neighbour matching. The priority is then to investigate alternatives where less keypoints are generated per frame as collectively 75% of the time per cycle of processing an input image is spent of functions that are heavily dependent on keypoint count.

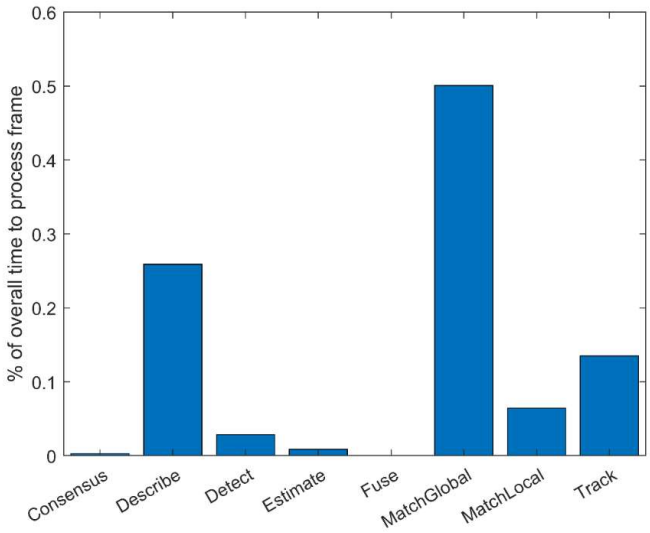


Figure 5.4: Time distribution amongst the image processing functions of CMT, with the majority of time spent on functions are a function of the number of descriptors.

5.3 Image feature optimizations

While it may seem logical to first build OpenCV with the optimization flags set for the relevant on-board hardware, it in fact blurs our results. Although hardware acceleration will be utilized in the final performance measurements of the algorithm, it is more logical to first investigate the configuration that is most suitable in terms of which detector and descriptor to utilize. Once we have completed this software investigation, we can investigate the performance increase that we achieve through the utilization of on-board hardware accelerators; thus, the increase in performance that we achieved with the chosen configuration will be clearer.

5.3.1 Feature detector

As previously mentioned, the first configuration option that should be investigated is the choice of feature detector, as most of the data that flows through the algorithm are keypoints and their descriptors. A minimal count of image features will enable efficient inference, and for this investigation we keep the descriptor fixed and determine which set of detectors produce the lowest amount of keypoints per frame, whilst still monitoring accuracy and inference rates, so as to highlight which detectors produce a low amount of *discriminative* keypoints.

Table 5.2 on the following pages highlights the fact that the GoodFeaturesToTrack detector, as well as the ORB detector, produce a conservative amount of keypoints per frame whilst still yielding a comparable AuC for the given dataset when coupled with a BRISK descriptor. Both detectors yield an inference rate an order of magnitude higher than that of the remaining detectors and as such are chosen for further investigation. Figure 5.5 reinforces how the feature detectors respond in an outdoor scene, as outdoor scenes are of great importance to the application space of this research. The two selected keypoint detectors respond sparsely, whereas the FAST and BRISK detectors overreact to the detail in the scene and are clearly unsuited for outdoor use.

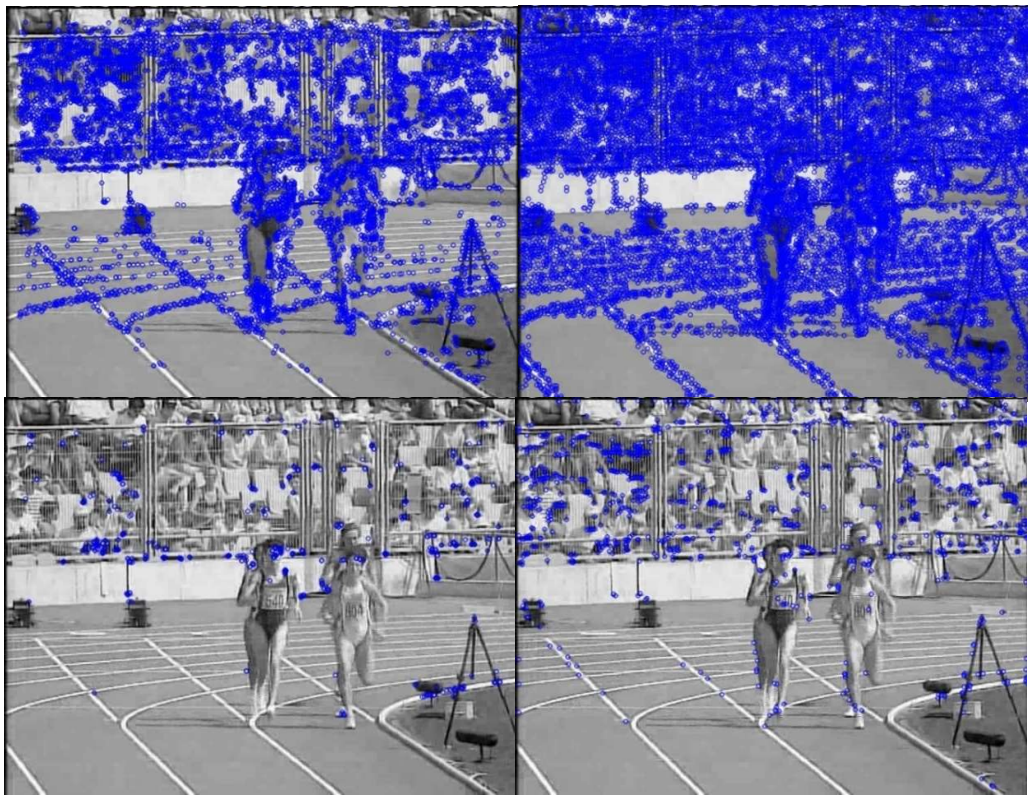


Figure 5.5: The investigated detector responses clockwise from top left to the scene 'track': BRISK, FAST, ORB and GFTT, with the top two detectors responding excessively in the outdoor scene. We see a distinct separation between ORB and GFFT, and BRISK and FAST.

Table 5.2: Summary of results regarding investigation into detector response.

Detector	Average extracted keypoints	FPS	AuC
ORB	388	13.92	0.66
GFFT	534	11.14	0.68
BRISK	844	4.2	0.70
FAST	1781	2.25	0.62

We note that there is no direct correlation to the amount of keypoints detected in each image to the inference rate achieved, but are aware of a possible relationship between certain detectors and descriptors, and certain detectors possibly producing more keypoints. How these keypoints are described, once produced by the detector, will be interesting to investigate, as we may see a detector producing many keypoints according to its internal configuration but the descriptor not 'agreeing' on these keypoints, resulting in many descriptors being rejected in the SNN test.

This leads us to investigate the relationship between the detectors and descriptors, and we can effectively measure interaction by the number of keypoints that are *active* - the lower the active keypoints, the more effective inference will be as well. We see later that there is a more direct correlation between the number of active keypoints and the rate of inference.

5.3.2 Feature descriptor

Following the previous investigation, it is pertinent to subsequently choose a descriptor that is as lightweight as possible, yet still discriminative to enable accurate and efficient inference. For this test, we retain the aforementioned detectors and evaluate their performance with an ORB and BRIEF descriptor, both of which are 32 bytes in their standard configuration, with ORB being an extension of the BRIEF descriptor to incorporate rotational invariance. For this experiment we investigate the performance of the algorithm with both the ORB detector, which is an extension of the FAST keypoint detector, and the GoodFeaturestoTrack detector, essentially an extension to the Harris Corner Detector, when combined with the ORB and BRIEF descriptors. We aim to investigate the combination that produces the highest rate of inference, but still keep accuracy in mind and

subsequently show that a smaller descriptor is of great importance to the rate of inference achievable.

From Table 5.3 we see that a clearer correlation exists between the number of keypoints that are *active,* and the rate of inference achieved. This is logical as the less data (keypoints and descriptors) that travels through the algorithm, the faster the functions can be executed, as there is less data to be processed. We notice two anomalies to this observation: 1) the configuration utilizing an ORB detector and BRIEF descriptor - this could be because, although there is a higher amount of active keypoints than other configurations, but still a high FPS rate, the lower keypoint count allows for less time to be spent matching this smaller count of descriptors and 2) the configuration utilizing an ORB detector and ORB descriptor - this could be due to the costlier operation of computing an ORB descriptor compared to the BRIEF descriptor in the aforementioned configuration.

Table 5.3:- CMT performance when configured with an ORB and GFTT detector, along with 32-byte binary descriptors, highlighting the GFFT detector when coupled with a 32-byte ORB descriptor being superior.

Detector	Descriptor	Average extracted keypoints	Average active keypoints	FPS	AuC
GFTT	ORB	639	58	15.94	0.695
ORB	ORB	391	77	11.91	0.705
GFTT	BRIEF	639	56	14.44	0.7207
ORB	BRIEF	391	69	15.16	0.6517

Table 5.3 serves to highlight the importance of a compact descriptor whereby we utilized a GoodFeaturesToTrack detector, due to it leading partly to the highest inference rate achieved with an ORB descriptor, to investigate the implications of using the same descriptor but in varying sizes.

The BRIEF descriptor can be configured to utilize a 16, 32 or 64-byte descriptor and the resulting discovery is logical, whereby the 16-byte descriptor achieves the highest FPS. An illogical result was realized in that the most lightweight configuration also yielded close to the highest accuracy, a result that is not predictable and could only have been found empirically due to the complex interrelation between detectors and descriptors and the tracking algorithm of

choice. The configuration of GoodFeaturesToTrack detector and 16-byte BRIEF descriptor is thus chosen as the configuration that is utilized for the rest of the investigation, unless stated otherwise. This was chosen as it yielded that fastest inference rate, but also with a relatively high AuC.

Table 5.4: CMT performance when utilizing the GFFT detector and binary descriptors of varying length, highlighting the importance of a compact descriptor.

Detector	Descriptor	Extracted keypoints	Active keypoints	FPS	AuC
GFFT	BRIEF (64)	639	65	13.12	0.7168
GFTT	BRIEF (32)	639	56	14.44	0.7207
GFFT	BRIEF (16)	639	59	16.48	0.7158

5.4 CPU hardware acceleration

In order to partly justify our choice for the chosen hardware platform, the following sections serves to highlight the available hardware acceleration options available for the ARM™ Cortex A17 processor and increase in inference rate achieved.

5.4.1 NEON

The Cortex A17 processor supports ARM's NEON Single Instruction Multiple Data (SIMD) architecture extension and incorporates a separate NEON engine into the SoC. The general approach to SIMD execution is illustrated in Figure 5.6 on the following page, whereby we have a singular instruction stream and multiple input data points that are handled concurrently, with SIMD being particularly suited to image processing tasks. CMT utilizes many operations that are data-parallel, or tasks that are suited to SIMD, such as keypoint detection, descriptor and kNN matching and thus we expect to realize significant improvements in inference rates through building the OpenCV library with NEON support.

5.4.2 VFPv3

The VPv3 is an optimization developed by ARM in order to effectively process floating point data, in both single and double precision, an optimization that should in theory be less attractive due to our application utilizing binary descriptors comprised of 8-bit binary strings, with a total of 16 strings per descriptor vector. It should still bring a minor speed up as other data points are

represented by floating point in the algorithm, but we expect the NEON instruction set to bring the greatest acceleration.

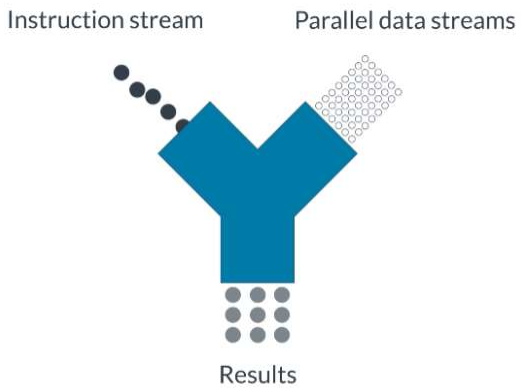


Figure 5.6: Single Instruction Multiple Data architecture (ARM – available online at: https://developer.arm.com/technologies/neon)

5.4.3 Results

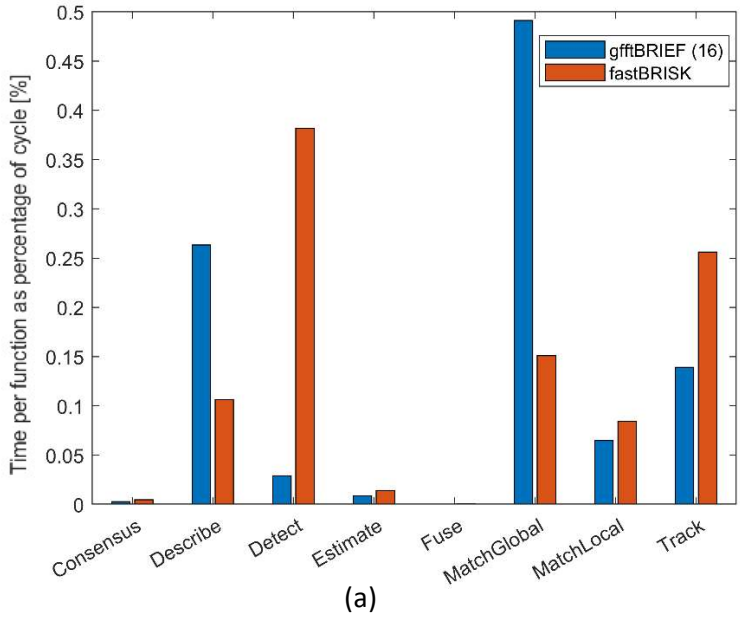
Post the investigation into the optimal software configuration for CMT in terms of inference rates, we repeat the prior tests in this optimal configuration with three different builds of the OpenCV library. The results from the prior test were first carried over, after which the OpenCV library was then rebuilt with NEON support and subsequently NEON and VFPv3 support, with the results summarized below in Table 5.5.

Table 5.5: Performance summary of CMT configured with the optimal GFTT detector and BRIEF descriptors, rivalling the desktop benchmark inference rate.

Optimization	Detector	Descriptor	FPS	AuC
N/A	GFTT	BRIEF (16)	16.4774	0.7158
NEON	GFTT	BRIEF (16)	20.3937	0.7391
NEON VFPv3	GFTT	BRIEF (16)	21.3921	0.7391

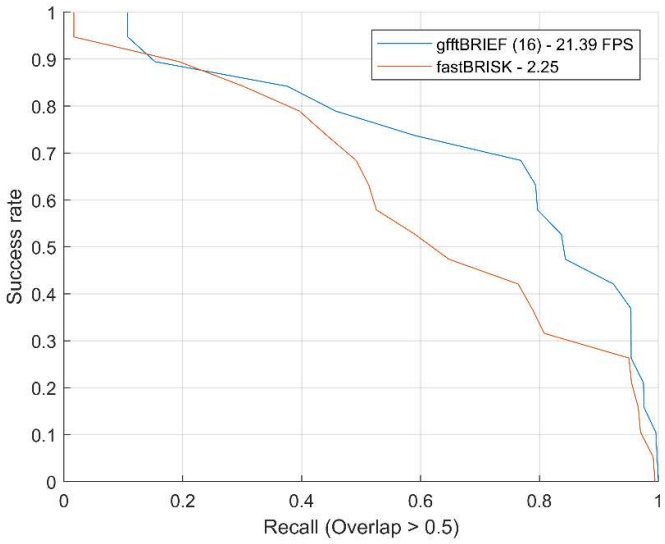
We notice that when utilizing the OpenCV library with NEON support enabled, an improvement of nearly 25% was achieved on average over all tracking sequences, and subsequently a mere 4% improvement was gained due to utilizing a floating-

point optimization. Even though the results are in line with the expectation that utilizing the NEON engine would bring the greatest improvement, the magnitude of improvement resulting from NEON acceleration was not expected – a sign that heterogenous implementation should bring significant improvements too, as GPU computation is also well suited to data-parallel tasks. Figure 5.7 b) on the following page serves to illustrate the performance of the final configuration of CMT on the embedded platform, with the interesting observation that not only is the final configuration faster by a magnitude of order than the baseline embedded configuration, but more accurate as well. It is important to note too that the final configuration achieves real-time performance and has surpassed the baseline implementation on the desktop platform in terms of average inference rate. Figure 5.7 a) also illustrates the more uniform distribution of time amongst the different computational steps of CMT in the final configuration, depicting that there are less significant bottlenecks.



(a)

Figure 5.7 (a): Time distribution amongst functions as a percentage of overall cycle time - a much more uniform distribution compared to the baseline configuration indicating less major computation bottlenecks.



(b)

Figure 5.8 (b): Success plot of CMT post relevant hardware and software optimizations, on the embedded platform. We note a near tenfold increase in inference rates on the embedded platform.

Figure 5.8 on the following page serves to highlight the benefit on individual functions of CMT due to utilizing NEON acceleration - a promising sign when GPU acceleration will be employed for these functions in the following section.

5.5 Heterogenous hardware implementation

5.5.1 Design

Assuming that we approached the highest rate of inference by solely utilizing the CPU of the embedded device, with the associated optimizations, we turn our attention to the design of a heterogeneous implementation that aims to increase the utilization of the on-board hardware, by allocating certain functions between the two available hardware resources. As depicted in Figure 5.7 a), two of the most time-consuming functions of CMT are detecting keypoints and computing the optical flow for these keypoints, both of which are highly data-parallel and as such suited to GPU acceleration.

Utilizing a certain hardware resource for effective acceleration is not as trivial as it may seem, whereby simply offloading functions, that have available implementations on the specific hardware, may likely lead to a disappointing increase in performance due to a variety of factors.

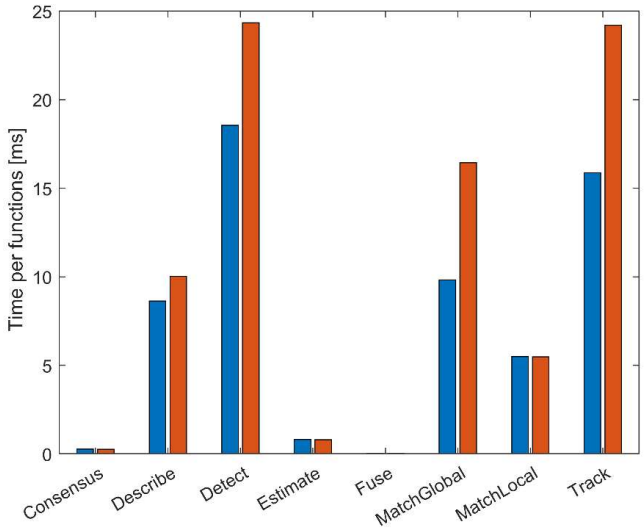


Figure 5.9: Performance benefit of utilizing NEON acceleration for all image processing functions in CMT. A significant decrease in computing time is realized in most data-parallel functions.

One such barrier to a significant performance increase is memory transfers between the host, the CPU, and the device, the GPU, especially when memory transfers occur in a cyclic fashion such as in CMT. As such, minimizing memory transfers is of great importance and as such we suggest the allocation as depicted below on the following page in Figure 5.9. In Figure 5.9 it is important to highlight the current resource allocation of CMT, of which we note the GPU is idle for the entire cycle, in which we process a single input frame.

In Figure 5.9, a reordering of the computational sequence of CMT was performed to ease of the complexity of depicting proposed heterogenous allocation. The function ‘track’ was initially computed first in the authors implementation of CMT, but we schedule the function after image features have been computed and processed - or features have been detected, described and the candidate keypoints and their descriptors matched globally to the database. Subsequently the computation of image features is allocated to the GPU as each task is highly data parallel, as the results of Figure 5.8 highlights in that utilizing NEON acceleration provided significant improvements in these three functions and have a tightly coupled data-dependency between functions. The variables returned from each of these functions is, as illustrated, passed directly to the following functions - keypoints are first detected, then passed to the descriptor whereby a descriptor is computed and associated with each keypoint after which each keypoint-descriptor pair is matched to the database of descriptors.

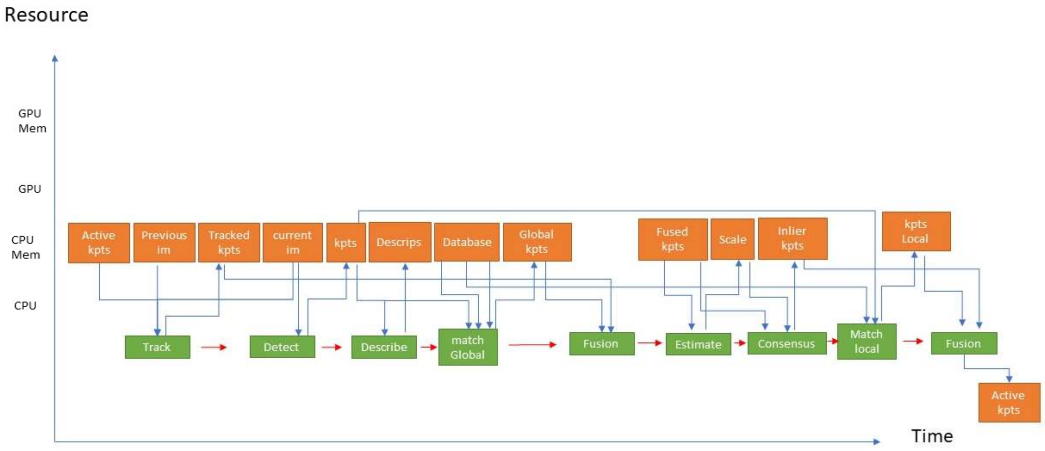


Figure 5.10: The CMT task graph. Orange symbols illustrate the variables that are stored in memory and that are returned by the functions, depicted in green. This figure serves to illustrate the allocation of functions to the available resources on a heterogenous computing platform, noting the GPU is unallocated. Red arrows illustrate algorithmic flow and blue input/outputs of functions.

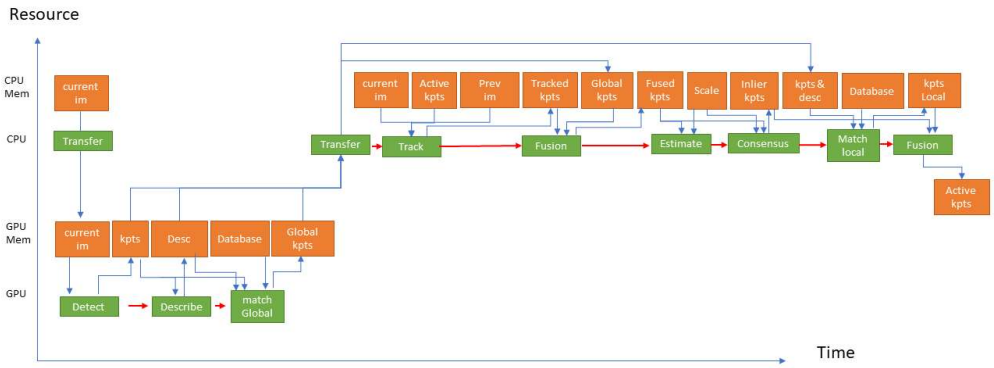


Figure 5.11: An efficient allocation of CMT's functions on a heterogeneous computing platform, with a single set of memory transfers to the device and back to the host. The functions 'Detect', 'Describe' and 'matchGlobal' have been allocated to the GPU.

This is attractive attribute as intermediate memory transfers back to the host between each function could be avoided, as these intermediate variables are not immediately required in the rest of the computing pipeline.

5.5.2 Embedded implementation

In order to allocate functions to the GPU, we utilize the so-called Transparent API (TAPI) within OpenCV that utilizes OpenCL framework to distribute computation between heterogenous platforms, since our chosen hardware platform supports the minimum required 1.2 full profile OpenCL framework. In order to allocate functions to the GPU we simply need to alter a single variable declaration that specifies the memory location of the basic image container 'Mat', that predominantly resides in CPU memory, with 'UMat' that will then be a GPU memory bound variable. The compiler will see this variable and handle GPU memory allocation automatically by creating the memory on the device, uploading the data synchronously to the device, and downloading the results back to host after the CPU has waited, in a synchronous fashion, for the GPU to complete computation. As such this ease of use leads to the Transparent API naming scheme as it designed to allow ease of use - if we examine the source code of the chosen feature detector's class, 'GoodFeaturesToTrack' and it's 'detect' method, we obtain an intuition for how the allocation is handled as illustrated in Figure 5.10. We see that the function ascertains the type of image container, 'Mat' or 'UMat', and calls the relevant function, 'goodfeaturestotrack()', accordingly.

```
void detect( InputArray _image, std::vector<KeyPoint>& keypoints, InputArray _mask )
{
    CV_INSTRUMENT_REGION()

    std::vector<Point2f> corners;

    if (_image.isUMat())
    {
        UMat ugrayImage;
        if( _image.type() != CV_8U )
            cvtColor( _image, ugrayImage, COLOR_BGR2GRAY );
        else
            ugrayImage = _image.getUMat();

        goodFeaturesToTrack( ugrayImage, corners, nfeatures, qualityLevel, minDistance, _mask,
                             blockSize, gradSize, useHarrisDetector, k );
    }
    else
    {
        Mat image = _image.getMat(), grayImage = image;
        if( image.type() != CV_8U )
            cvtColor( image, grayImage, COLOR_BGR2GRAY );

        goodFeaturesToTrack( grayImage, corners, nfeatures, qualityLevel, minDistance, _mask,
                             blockSize, gradSize, useHarrisDetector, k );
    }

    keypoints.resize(corners.size());
    std::vector<Point2f>::const_iterator corner_it = corners.begin();
    std::vector<KeyPoint>::iterator keypoint_it = keypoints.begin();
    for( ; corner_it != corners.end() && keypoint_it != keypoints.end(); ++corner_it, ++keypoint_it )
        *keypoint_it = KeyPoint( *corner_it, (float)blockSize );

}
```

Figure 5.12: Source code snippet from the 'GoodFeaturesToTrack' detector, where we see the 'detect' method's Transparent API functionality, whereby the function checks the input type and responds accordingly (OpenCV 3.4.0 – available online at: https://docs.opencv.org/3.4.0/).

5.5.3 Shortcomings

Upon attempting to utilize the TAPI on the ASUS Tinkerboard™ certain barriers were encountered related to the specific device architecture. All the methods, notably 'detect()', 'compute()' and 'knnMatch()', from each respective class, when attempted to be used with the TAPI returned the error: "CL_OUT_OF_RESOURCES". This error has to do with the fact that the functions and more specifically the OpenCL kernels were developed with scalar desktop hardware in mind, opposed to Mali mobile GPU vector architecture. In order to rectify these issues, much developmental work would be required and is out of the scope of this research, which a feasibility analysis whilst making use of available, functioning tools. The developmental work would require a complete rewriting of OpenCL kernels such that they are functional on vector architecture (vectorisation), as current operation is on scalar architecture. Once the kernels are functional on vector architecture, an optimization process would have to be undertaken to realize the specifics of a mobile GPU – i.e. to take advantage of the 128 bit wide registers. This would involve first determining the variable type, either int, short half-float, for the specific task and how to best fit as many elements of the variable type in the 128-bit registers. We thus turn our attention to a functioning implementation platform, that being the desktop platform utilized in the beginning of the experiments which is equipped with an NVIDIA™ GPU and utilizing the mature and widely utilized CUDA API in OpenCV.

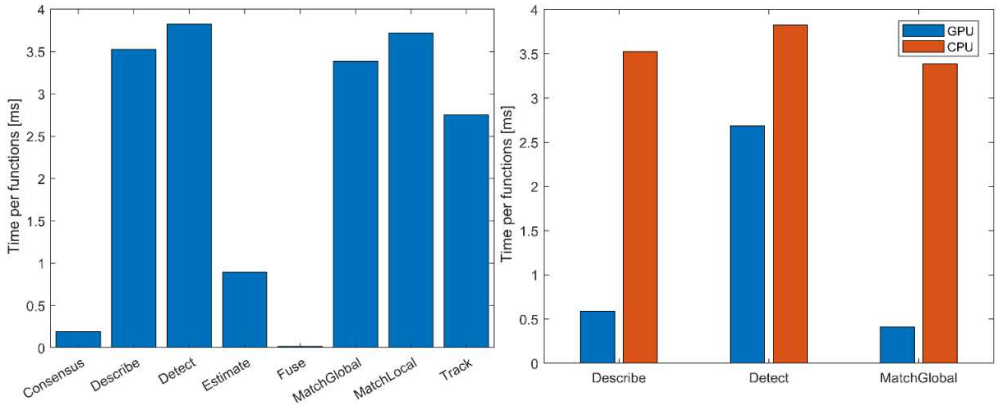
5.5.4 Desktop implementation

Due to the embedded system being deemed to be an unsuitable prototyping platform for a GPU-accelerated implementation of CMT as discussed in the previous section, we return to the original desktop hardware platform. After investigating the feature detectors and descriptors that have been implemented in the CUDA API of OpenCV, we settle on an ORB descriptor and detector from the available FAST and ORB keypoint detectors and solely available ORB descriptor. Table 5.3 in section 5.3.1 illustrates that ORB detector and descriptor yields high accuracy, but most importantly a superior keypoint detector response when compare to the FAST keypoint detector, and since hardware acceleration will be utilized, the lower inference rate that an ORB keypoint paired with an ORB descriptor presents when empirically investigated may be neglected.

Referring to Figure 5.11 on the following page, in (b), we allocated the functions detect(), describe() and matchGlobal() to the GPU as they present large overheads in the overall pipeline of processing an input image, illustrated in (a), when measured on the desktop hardware platform. Further motivation for allocating these functions to the GPU will be discussed in the coming section. As envisioned, a vast decrease was realized, as illustrated in Figure 5.11 (b), for each of the functions allocated to the GPU, as they are highly data parallel. Both the

computation of descriptors and matching them to the database realized a 7x reduction in computing time per frame, averaged over all frames in the dataset. It is however noticed that the decrease of 30% in detecting keypoints is less impressive - postulated to be since the input images in the database are predominantly of relatively low pixel count at 340x240 pixels, compared to a 1080x720 pixel HD image. Subsequently the GPU has lower opportunity, when compared to computing descriptors or matching them to a database, to present a significant acceleration in computation due to the low dimension input to the 'detect' method.

This fact is reinforced whereby the GPU acceleration achieved, for the three allocated functions, was investigated for the sequence 'tracking running' where is the input is 768x576 pixels and results for this singular sequence is summarised in Table 5.6 on the following page. Table 5.7, also on the following page, summarises the result of utilizing GPU acceleration in the tracking algorithm over the entire dataset.



(a) (b)

Figure 5.13: The distribution of time amongst the functions, illustrated on the left in (a) of CMT on the desktop platform when solely utilizing the CPU and in (b) the measured decrease in computation time for functions allocated to the GPU, with an on average decrease of 5x over the three functions against a CPU implementation is illustrated. Experiments were carried out utilizing a NVIDIA GTX1060 6GB.

Table 5.6: A summary of the time taken per function for the sequence 'track running' whereby a larger decrease, nearing 2x reduction in time, can be seem for the method 'detect', illustrating the dependence of the keypoint detector

on high dimensional input data to achieve significant acceleration when allocated to a GPU. The remaining functions see similar decreases due to the number of keypoints and descriptors being fixed by the user.

Hardware platform	Detect	Describe	MatchGlobal
CPU	4.66	3.90	2.67
GPU	2.66	0.63	0.29

Table 5.7: The summary of the impact on inference rate and accuracy by allocating data-parallel, high-complexity computations to the GPU, compared to a pure CPU allocated implementation of CMT. The algorithm is well suited to GPU acceleration, due to the presence of commonly-utilized and intensive image processing tasks being well suited to GPU acceleration, and as such efficient GPU implementations of these functions being available. It is noted that by utilizing GPU acceleration for certain functions, the AuC decreases by 5% - an acceptable decrease in accuracy for a 2x reduction in inference rate.

Hardware platform	Detector	Descriptor	FPS	AuC
CPU	ORB	ORB	42.79	0.739
CPU & GPU	ORB	ORB	87.92	0.7039

5.6 Concurrent execution model

After the investigation and the subsequent reinforcement of suitability of CMT for GPU acceleration, we take Figure 5.9 into account and note that data processing still proceeds in a sequential manner - whilst each resource is allocated and busy computing, the other resource in the heterogenous platform remains idle.

The following section serves to highlight the feasibility and design of a concurrent execution model, whereby both hardware resources are allocated and processing data at the same time.

5.6.1 Design

To justify the developmental effort for designing a concurrent execution model, it seems pertinent to highlight the motivational factor which is in mobile computing, since hardware resources and power supply is limited, it is of utmost importance that we utilize the available hardware resources to their highest capacity to achieve the highest possible throughput. An approach to concurrent execution in computer vision pipeline is to overlap tasks between two successive cycles, such as in CMT's cyclic processing pipeline, to ensure a resource avoids being left idle and is rather continually being utilized at most points in a cyclic processing pipeline.

The GPU, although allocated a certain portion of the functions of CMT's object tracking pipeline in Figure 5.12 below, is left idle for the remainder of the cycle achieving low hardware utilization when considering the span of the entire frame-processing pipeline. The remainder of the cycle, the stage assigned with updating the object model, is completed by the CPU.

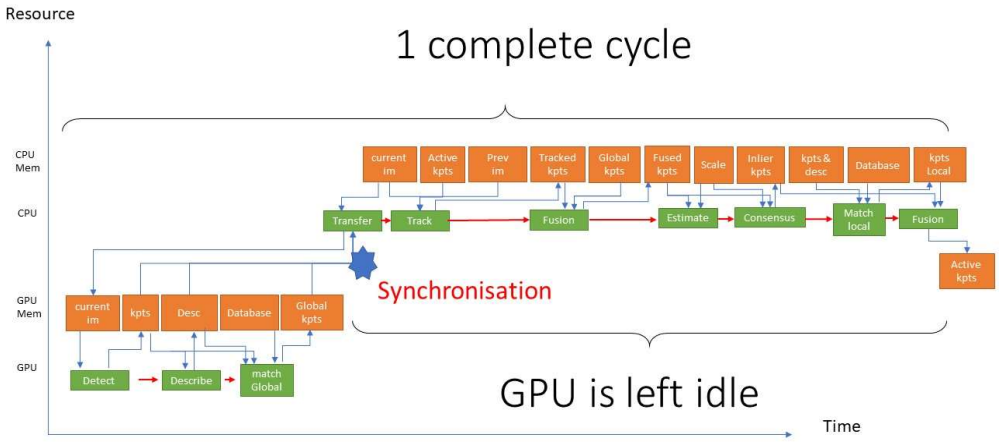


Figure 5.14: An illustration of allocation of functions to specific hardware resources, with a low-level utilization of hardware resources.

However, for an algorithm to be suitable to be executed in a concurrent manner or to be overlapped between cycles, a seemingly sequential set of computations, as illustrated in Figure 5.12, must be decoupled of data dependencies for the computations to be executed in parallel. In terms of the algorithmic steps in Figure 5.12, there are essentially two tightly coupled and data dependent computational stages to the algorithm. The image feature stage, which Nebehay refers to as the static component, is tightly coupled in terms of the intermediate data shared between tasks and is independent of remaining tasks in the cycle, until a much later point in the cycle. This presents an opportunity as the two stages in a single cycle of the object tracking pipeline are distinctively independent of each other.

As previously stated, this allows the tasks suited to GPU acceleration to computed with a single transfer of data to the device upon initialization, and subsequently only one synchronization post completion is required with the host, illustrated in Figure 5.12. Thus, upon completion of a single cycle's GPU allocated functions and after the memory transfer from device back to the host, it seems pertinent for the CPU to first capture next image from the sensor. By capturing the next image, the same functions can then be reallocated to the GPU, and by doing so *fill* the pipeline. The image features for the next frame can then be computed by the GPU concurrently and by doing so, overlap computation between two cycles as illustrated in Figure 5.13. A concurrent execution model has thus been achieved and the utilization of the hardware platform should be increased and as such present a higher throughput.

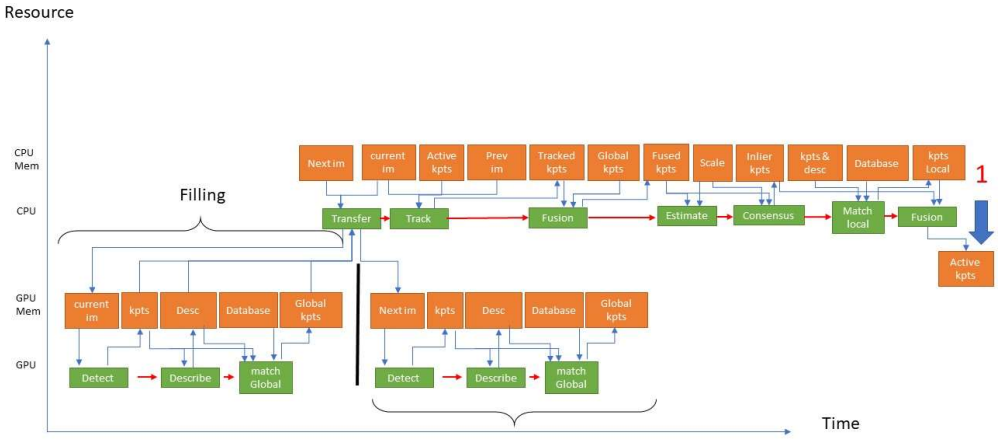


Figure 5.15: Design of the pipelined execution model, with both resources allocated during a single cycle of CMT. We see that whilst the CPU process the current image, the GPU is processing the next image. Synchronisation occurs at "1".

However, to develop an effective concurrent execution model, it is important to consider the trade-off between utilizing a specific computational resource and the overhead between memory transfers. Whilst the theoretical speedup achievable by a GPU may be magnitudes of order when compared to the same task on a CPU, as illustrated in Figure 5.11 (b), inefficient memory handling may result in an execution model that achieves a lower throughput and overall higher processing time per cycle. By utilizing the asynchronous set of functions from the CUDA API for feature detection (detectAsync()), computing descriptors (computeAsync()), and matching of descriptors (knnMatchAsync()), all data that is shared between the image feature tasks allocated to the GPU resides in GPU device memory. The asynchronous functions accept GPU-memory allocated data as arguments,

opposed to the synchronous versions that only accept CPU-memory allocated data as their arguments. Further, no explicit synchronization is required with the CPU upon the completion of these individual functions, and as such it possible to achieve an execution model whereby only a single synchronization is required between host and device. It should also be pointed out that the 'database' variable is static, and a copy is thus held in both CPU and GPU memory, such that we avoid repeatedly transferring the variable per cycle.

5.6.2 Results

To be able to draw conclusions from the results of the investigation into a concurrent execution model, it seems pertinent to first determine the percentage of a complete cycle that the tasks that were intended to be executed concurrently take to compute when allocated to the GPU. This would serve as the upper bound and would highlight if *perfect pipelining* was achieved - if the cycle time was reduced by the amount of time occupied by these functions. Measurements were made utilizing OpenCV's function getTickCount() and getTickFrequency(), that returns timing in a resolution of milliseconds.

Table 5.8: Time taken to process functions allocated to the GPU, as illustrated in Figure 5.12, with a cycle decrease of 27.34 % achievable in the limit of perfect pipelining.

Dataset	Average time to process a single frame (1 cycle) [ms]	Average time to process GPU tasks per cycle [ms]	Upper limit achievable by pipelining [%]
Vojir dataset (22 sequences)	8.63	2.09	27.34

Clearly, the functions allocated to the GPU occupy a significant portion of each individual cycle, and as such we can expect to see a considerable decrease in cycle period if the functions can be computed concurrently.

Table 5.9 - Complete cycle period measurements, post concurrent execution.

Dataset	Average time to process a single frame [ms]	Average time to process a single frame w/ pipelining [ms]	Measured decrease in cycle time [%]
Vojir dataset (24 sequences)	8.63	7.97	7.64

It is noted that the average decrease in cycle time was not as expected, and another experiment was undertaken to measure the time taken to transfer data from the CPU to the GPU and to launch the tasks on the GPU.

Following the results of the second experiment, the unexpected results obtained in the first experiment can be attributed to the fact the execution of the functions are not entirely asynchronous as initially assumed. Following a profiling exercise of individual functions, using the NVIDIA NSight profiler, that were to be executed concurrently such as detectAsync() illustrated below in Figure 5.14, it is observed that the memory allocation and transfer between the host and device, as well as device kernel launching is continual throughout the functional call from the runtime API (of which the host controls).

Table 5.10: Memory management and launching of kernels on the GPU

Dataset	Average time to transfer data and launch tasks on GPU per single frame [ms]	Average time to process GPU tasks per cycle [ms]	Realizable decrease in cycle time [ms]	Average time to process a single frame (1 cycle) [ms]	Realizable decrease in cycle time [%]
Vojir dataset (22 sequences)	1.42	2.09	0.67	8.63	7.71

Asynchronous computation does indeed occur, such as in Figure 5.14 where there is a kernel invoked on the device and whilst the kernel is busy being executed on the device a memory copy is taking place, but it is not to the extent as assumed in the case of the results of Table 5.8. It was assumed that once the function was called from the API, the CPU would allocate the required memory, transfer the data to the device and launch the kernels on the GPU and return control for further processing. It is postulated that as the API was designed to be simple, robust and user friendly, as well as portable across many heterogenous platforms with a NVIDIA GPU, to include such asynchronous functionality could comprise the aforementioned goals.

The decrease in processing time of 7.71% highlighted in Table 5.10 is postulated to be as a result of two factors. The first contribution to the decrease in processing time is since the asynchronous implementations of the previously mentioned

functions accepting variables residing in GPU memory as arguments, unnecessary memory transfers are thus eliminated. Further, the decrease in processing time can also be attributed to the fact the asynchronous implementations of the functions, such as detectAsync(), make no explicit device synchronization calls.

Device synchronization (cudaDeviceSynchronize) calls from within the runtime API block the CPU until all kernels are complete on the GPU, and subsequently a profiling investigation was undertaken into the behaviour of the functions detect() and detectAsync(), with the result of the illustrated below in Figure 5.15. The large difference between the two functions can be attributed to the synchronous implementation making explicit device synchronization calls, as only the synchronous implementation of detect() having a cudaDeviceSynchronize entry in Figure 5.15, and cumulatively the CPU spends a large portion of the function detect() idling waiting for the GPU to complete computation.

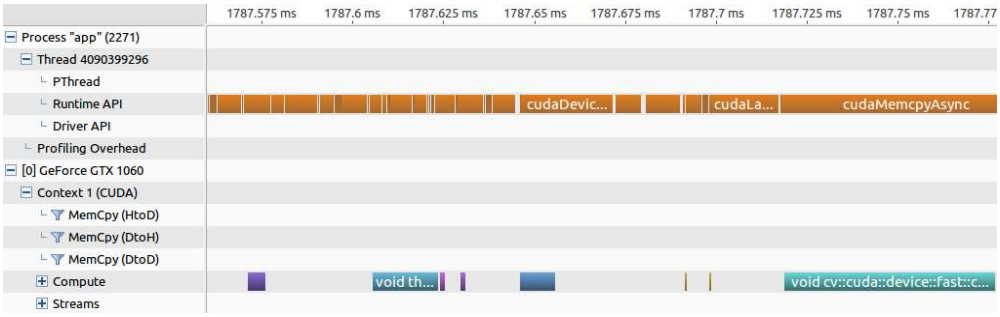


Figure 5.16: Profiling results of detectAsync() function, illustrating the continual interaction between CPU (runtime) in orange and GPU compute kernels in turquoise/blue. Asynchronous computing is occurring, such as where the kernel 'cv::cuda::device::fast' is being computed on the device and the host is allocating memory with 'cudaMemcpyAsync'.

The time distribution of kernel execution on the device is not presented as the implementations of the function detect(), synchronous and asynchronously, have near identical time distributions amongst kernels and have identical kernel invocations.

As the goal of this last investigation was into the feasibility of the object tracking algorithm for a concurrent execution model, with the eventual goal of implementation on a low cost, single board computer equipped with heterogenous hardware, the investigation is considered successful as a concurrent execution model was successfully designed. The sequential execution of the original implementation was decoupled of data dependencies, leading to

concurrent execution being possible, and the poor results could be attributed to the functionality of the API that was utilized.

However, it should be pointed out the approach would not produce a significant decrease in cycle period nor boost throughput as envisioned on an embedded platform, using standard software libraries. As the approach utilized high end hardware for the investigation, the results achieved on a single board computer would even less impressive due to the generally lower clock speeds for memory and processors, as well as a narrower memory interface width.

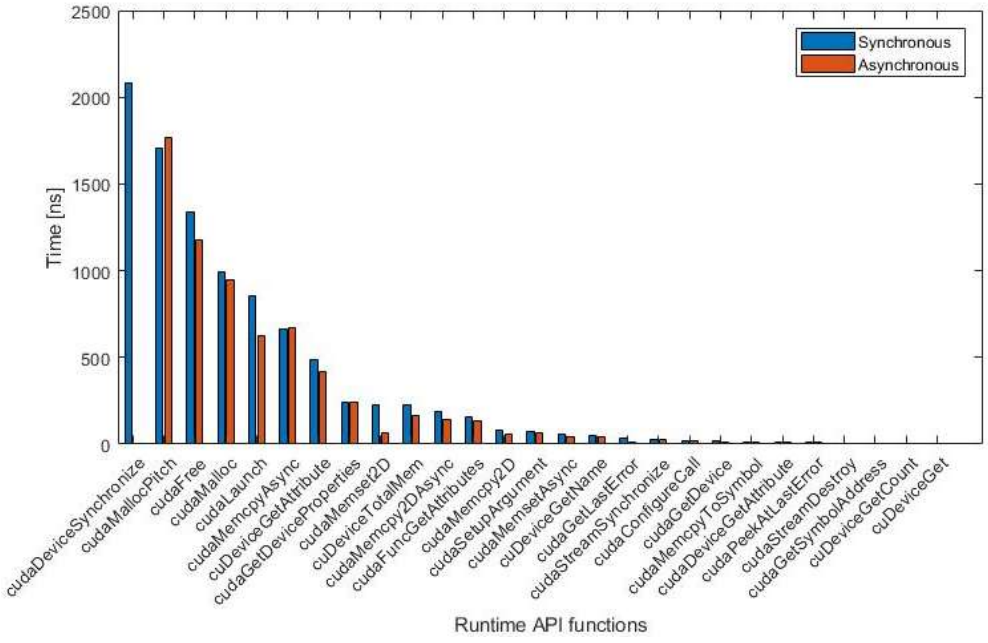


Figure 5.17: Time distribution of function detect() in the runtime API, illustrating that the synchronous implementation's costly deviceSynchronise calls.

Even though approach does not increase the cost of the system on hand, only an investment of development time is required, the suboptimal results are an indication that another approach should first be investigated if the goal is to vastly improve throughput and decrease cycle period.

6 Conclusion

Generic object tracking has become a well-researched topic following the success and attention that Tracking-Learning-Detection (Kalal, 2012) received. A benchmark for evaluating single object, generic object trackers was developed in 2013 and aided standardized methods to compare trackers in an efficient and simple manner. The Visual Object Tracking Challenge (VOT) has made a large positive impact on the field of research in generic object tracking and as such significant annual improvements in the accuracy and robustness of generic object trackers are continually made.

With the introduction of the real-time challenge in 2015, as a sub-challenge within VOT, the importance of inference rates gained attention as well. The real-time challenge serves to rank a subset of generic object trackers that can process the input stream faster than the sensor can capture its digital representation of the real-world, with the same accuracy and robustness measures as in the overall challenge in the VOT. Less attention has however been given to inference of generic object trackers on the edge, on energy-efficient mobile devices.

The purpose of this research was to investigate the feasibility of a generic object tracker for inference on a single board computer. The algorithm CMT was highlighted for investigation as initial research showed that the algorithm is highly configurable, had not yet investigated for hardware acceleration, and performed well on tracking scenes that are suitable to the intended use case. The required software library for inferring the algorithm on the single board computer was built and the accuracy and inference performance was investigated, in the author of the algorithm's configuration. This configuration was deemed unsuitable for inference on mobile, resource limited computational platforms and an investigation was undertaken to discover more optimal configurations.

Using a feature detector that did not overrespond to highly detailed scenes, such as those found outdoors, and a lightweight but still discriminative descriptor vector, inference rates were increased 7x on the mobile platform without any significant loss in tracking accuracy, on the chosen dataset. The software library that was utilized for generic image processing tasks was then rebuilt in order to utilize the onboard hardware acceleration engine for the CPU, as well as another specific data-type optimization, yielding a final inference rate that improved the initial rate on the mobile platform by an order of magnitude, to a frame processing speed of value of above 20 FPS.

Another form of hardware acceleration on the single board computer was then attempted to be utilized, notably a heterogenous computing approach using GPGPU for intensive image processing tasks, as a GPU is present on the single board computer, but it was found that the software library's GPU accelerated

functions were developed for desktop hardware. As such the functions that were intended to be accelerated using the GPU and the software library's API for GPU acceleration, using the OpenCL heterogenous computing programming framework, did not function on the single board computer and thus investigation into heterogenous computing on the mobile device was abandoned.

Subsequently the prototyping platform was altered to a desktop platform to make a general investigation into whether the chosen object tracking algorithm would benefit from GPU acceleration, and whether further developmental effort should be undertaken to obtain a functioning set of GPU acceleration functions, on the single board computer. The inference rate of the algorithm was subsequently doubled by allocating the computationally intensive image processing tasks to the GPU, which could be done efficiently due to a reshuffling of the computational flow of the algorithm to enable a set of data-independent functions to be accelerated by the GPU. A final step was then taken to investigate whether the utilization of a hardware platform, in general, could be improved significantly in the execution of the specific tracking algorithm by investigating concurrent execution. This was also investigated by utilizing the desktop hardware platform due to the hardware being well supported by with a set of asynchronous functions. The envisioned result was not achieved due to the utilized asynchronous functions not interacting with the hardware as assumed, with the assumptions that the interaction between the CPU and GPU would be totally asynchronous, but the algorithm was deemed to still be suitable for such an execution model due the existence of two subsets of data-independent tasks. To the authors knowledge, to realize an effective concurrent execution a large developmental effort would have to be undertaken by a team of hardware and software engineers. In general the algorithm was successfully accelerated with the GPU, reaching a final inference rate of over 100 FPS, and subsequently is well suited to GPU acceleration due to the presence of data parallel image processing tasks.

In closing, we return to the fact that the investigation was into the feasibility of real-time inference of a generic object tracking algorithm on a general-purpose single board computer. Since a FPS rate of 20 FPS was achieved, with a purely CPU implementation, the algorithm and platform is deemed suited to the task. This mostly as a result of extensive research into effective computational method of image features, and the field object tracking can expect even more promising results in terms of mobile inference once ANNs are efficiently computed on mobile devices. Through a careful design, a suitable speed vs accuracy balancing exercise and careful selection of a suitable algorithm and hardware platform, the task real-time object tracking can be achieved on a general-purpose single board computer.

Future work should investigate heterogenous computing with a suitable software library. The ARM Compute library is software package designed to efficiently compute machine learning and computer vision functions specifically on ARM

heterogenous SoCs such as the Tinkerboard. Support for common CNN functions such as the convolutional filter and pooling operators are present. An approach would be to investigate the usage of the ARM Compute library to implement object tracking algorithms that are labelled as *real-time,* from the VOT challenge, on mobile heterogenous SoCs.

7 References

Abadi M., Agarwal A., Barham P., Brevdo E., Chen Z., Citro C. and et al, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” arXiv:1603.04467, 2016.

Bazzani L., Freitas N., Larochelle H., Murino V., Ting J.A., “Learning attentional policies for tracking and recognition in video with deep networks,” in: Proceedings of the 28th International Conference on Machine Learning (ICML-11), June 2011, pp. 937–944.

Bertinetto L., Valmadre J., Henriques J. F., Vedaldi A., and Torr P. H. S., “Fully-convolutional siamese networks for object tracking,” in: Computer Vision-ECCV 2016 Workshops: Amsterdam, The Netherlands, October 8-10 and 15-16, 2016, Proceedings, Part II, G. Hua and H. Jégou, Eds. Springer International Publishing, 2016, pp. 850–865.

Bontempi G., Taieb S., Le Borgne Y-A., "Machine Learning Strategies for Time Series Forecasting," in: Neurocomputing, 2010.

Calonder M., Lepetit V., Ozuysal M., Trzcinski T., Strecha C., and Fua P., “BRIEF: Computing a Local Binary Descriptor Very Fast,” in: Transactions on Pattern Analysis and Machine Intelligence 34.7, 2012, pp. 1281–1298.

Cannons K., "A review of visual tracking". Department of Computer Science Engineering, York University, Toronto, Canada, 2008.

Carroll A., Heiser G., "An Analysis of Power Consumption in a Smartphone," in: USENIX, 2010, pp. 21-22.

Cehovin L., Kristan M., and Leonardis A., “Is my new tracker really better than yours?” WACV 2014: IEEE Winter Conference on Applications of Computer Vision, 2014.

Gall J., Yao A., Razavi N., Van Gool L., and Lempitsky V., “Hough Forests for Object Detection, Tracking, and Action Recognition,” in: Transactions on Pattern Analysis and Machine Intelligence 33.11, 2011, pp. 2188–2202.

Goodfellow I., Bengio Y., Courville A., “Deep Learning”, MIT Press, 2016.

Grauer-Gray S., Killian W., Searles R., Cavazos J., “Accelerating financial applications on the GPU,” in: Sixth Workshop on General Purpose Processing Using GPUs, 2013.

Harris C. and Stephens M., “A Combined Corner and Edge Detector,” in: Alvey Vision Conference. 1988, pp. 147–151.

He K., Zhang X., Ren S., Sun J., “Deep Residual Learning for Image Recognition,” arXiv: 1512.03385, 2015.

Hubel D. H., Wiesel T. N., “Receptive fields, binocular interaction, and functional architecture in the cat’s visual cortex”. Journal of Physiology (London), 160,106–154. 1961

Intel Corporation. Intel SSE4 Programming Reference. 2007.

Iandola F. N., Han S., Moskewicz S., Ashraf K., Dally W. J., Keutzer K., “SqueezeNet: AlexNet-level accuracy with 50xfewer parameters and <0.5MB model size,” arXiv:1602.07360, 2016.

Ignatov A., Timofte R., Chou W., Wang K., Wu M., Hartley T., Van Gool L., “ I Benchmark: Running Deep Neural Networks on Android Smartphones,” arXiv:1812.01109, 2018.

Jaccard P., “The Distribution of the Flora in the Alpine Zone,” in: New Phytologist 11.2, 1912, pp. 37–50.

Kalal Z., Mikolajczyk K., and Matas J., “Forward-Backward Error: Automatic Detection of Tracking Failures,” in: International Conference on Pattern Recognition. 2010, pp. 23–26.

Kalman R.E., “A New Approach to Linear Filtering and Prediction Problems,” in: Journal of Basic Engineering 82.1, 1960, pp. 35–45.

Koller D., Weber J., Malik J., “Robust multiple car tracking with occlusion reasoning,” in: European Conference on Computer Vision. 1994, pp. 189–196.

Kristan M., Pflugfelder R., Leonardis A., Matas J., Porikli F., Cehovin L., Nebehay G., Gustavo F., Vojir T., “The VOT2013 challenge: overview and additional results,” in: Computer Vision Winter Workshop. 2014, pp. 61–68.

Kristan M., Leonardis A., Matas J., Felsberg M., Pflugfelder R., Cohevin L. and et al, “The visual object tracking VOT2016 challenge results,” in ICCV 2017 Workshops, Workshop on visual object tracking challenge, 2017.

Kristan M., Leonardis A., Matas J., Felsberg M., Pflugfelder R., Cohevin L. and et al, “The visual object tracking VOT2016 challenge results,” in ECCV 2016 Workshops, Workshop on visual object tracking challenge, 2016.

Krizhevsky A., Sutskever I., and Hinton G. E., “ImageNet Classification with Deep Convolutional Neural Networks,” in: Conference on Neural Information Processing Systems. 2012, pp. 1097–1105.

Leutenegger S., Chli M., and Siegwart R. Y., “BRISK: Binary Robust invariant scalable keypoints,” in: International Conference on Computer Vision. 2011, pp. 2548–2555.

Le Cun Y., Boser B., Denker J. S., Henderson D., Howard R. E., Hubbard W., Jackel L. D.,"Handwritten Digit Recognition with a Back-Propagation Network," in: Advances in Neural Information Processing Systems, 1989)

Lepetit V., Lagger P., Fua P., “Randomized Trees for Real-Time Keypoint Recognition,” in: Conference on Computer Vision and Pattern Recognition. 2005, pp. 775–781.

Lowe D.G., “Distinctive Image Features from Scale-Invariant Keypoints,” in: International Journal of Computer Vision 60.2 (2004), pp. 91–110.

Lu Z., Chan K., Rallapalli S. and La Porta T., “Modelling the Resource Requirements of Convolutional Neural Networks on Mobile Devices,” arXiv: 1709.09503, 2017.

Lucas B.D. and Kanade T., “An iterative image registration technique with an application to stereo vision,” in IJCAI, 1981.

Maggio E., Cavallaro A., "Video Tracking: Theory and Practice," 2011.

Murphy K.P., “Machine Learning. A Probalistic Perspective,” MIT Press, 2012.

Nebehay G. and Pflugfelder R., “Clustering of static-adaptive correspondences for deformable object tracking,” in Computer Vision and Pattern Recognition, 2015.

Norvig R., Russel S., “Artificial Intelligence. A Modern Approach,” Pearson, 2010.

Redmon J., Divvala S., Girshick R., Farhadi A., "You only look once: Unified, real-time object detection," in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016.

Rosten E. and Drummond T., “Machine Learning for High-Speed Corner Detection” in: European Conference on Computer Vision. 2006, pp. 430–443.

Rublee E., Rabaud V., Konolige K., and Bradski G., “ORB: An efficient alternative to SIFT or SURF,” in: International Conference on Computer Vision. 2011, pp. 2564–2571.

Rumelhart D., Hinton, G. Williams, R. Williams, McClelland J., "Learning representations by back-propagating errors," in: Nature. 323, 1986, 533–536.

Shi J. and Tomasi C., “Good Features to Track,” in: Conference on Computer Vision and Pattern Recognition. 1994, pp. 593–600.

Stone J.E., Phillips J.C., Freddolino P.L., Hardy D.J., Trabuco L.G., Schulten K., "Accelerating molecular modeling applications with graphics processors," in: Journal of Computing Chemistry, 2007.

Szeliski, R., “Computer Vision: Algorithms and Applications,” Springer, 2010.

Velasco-Montero D., Fernandez-Bernia J., Carmona-Galana R., Rodrıguez-Vazquez A., “Performance analysis of real-time DNN inference on Raspberry Pi,” in: SPIE Commercial and Scientific Sensing and Imaging, 2018.

Wolpert D. H., Macready W. G., “No Free Lunch Theorems for Optimization,” in: IEEE Transactions on Evolutionary Computation, Vol. 1, No. 1, April 1997.

Wu J., Leng C., Wang Y., Hu Q., Cheng J., "Quantized Convolutional Neural Networks for Mobile Devices,", in: IEEE Conference on Computer Vision and Pattern Recognition, 2016.

Appendix A

```
115 void CMT::processFrame(Mat im_gray) {
116         double hookMain = cv::getTickCount();
117     FILE_LOG(logDEBUG) << "CMT::processFrame() call";
118
119     //Track keypoints
120     vector<Point2f> points_tracked;
121     vector<unsigned char> status;
122         double hook = cv::getTickCount();
123     tracker.track(im_prev, im_gray, points_active, points_tracked, status);
124         double hook1 = cv::getTickCount();
125         trak += (hook1-hook)/cv::getTickFrequency();
126
127
128     FILE_LOG(logDEBUG) << points_tracked.size() << " tracked points.";
129
130     //keep only successful classes
131     vector<int> classes_tracked;
132     for (size_t i = 0; i < classes_active.size(); i++)
133     {
134         if (status[i])
135         {
136             classes_tracked.push_back(classes_active[i]);
137         }
138
139     }
140
141     //Detect keypoints, compute descriptors
142     vector<KeyPoint> keypoints;
143         hook = cv::getTickCount();
144     detector->detect(im_gray, keypoints);
145         hook1 = cv::getTickCount();
146         detekt += (hook1-hook)/cv::getTickFrequency();
147
148     FILE_LOG(logDEBUG) << keypoints.size() << " keypoints found.";
149
150     Mat descriptors;
151         hook = cv::getTickCount();
152     descriptor->compute(im_gray, keypoints, descriptors);
153         hook1 = cv::getTickCount();
154         kompute += (hook1-hook)/cv::getTickFrequency();
155
156     //Match keypoints globally
157     vector<Point2f> points_matched_global;
158     vector<int> classes_matched_global;
159         hook = cv::getTickCount();
160     matcher.matchGlobal(keypoints, descriptors, points_matched_global, classes_matched_global);
161         hook1 = cv::getTickCount();
162         matshG += (hook1-hook)/cv::getTickFrequency();
163
```

Figure A1: Illustration of the timing method used for each individual function whenever timing was employed throughout section 5, starting with Figure 5.4.

```
133 void CMT::processFrame(Mat im_gray, int frame, Mat im_gray_next, int pipeline) {
134         float start = cv::getTickCount();
135     FILE_LOG(logDEBUG) << "CMT::processFrame() call";
136 
137     vector<KeyPoint> keypoints;
138     Mat descriptors;
139 
140     //Initialize variables in device memory
141     cv::cuda::GpuMat imGrayScaleP, keypointsTestGpuP, mask1P, keys1P, desc1P, matchesGpuP;
142 
143           //Check and fill pipeline
144     if(pipeline && frame == 1){
145 
146                 imCurrent = im_gray;
147                 imNext = im_gray_next;
148                 imPrevious = im_prev;
149 
150                 //Variables for holding in device memory
151                 cv::cuda::GpuMat imGrayScale, keypointsTestGpu, mask1, keys1, desc1;
152 
153                 //Upload image into device memory
154                 imGrayScale.upload(imCurrent);
155 
156                 //Detect keypoints and compute descriptors - detectAndCompute() uneccessary eliminates memory transfers
157                 gpuOrb->detectAndComputeAsync(imGrayScale, cv::noArray(), keys1, desc1, false, stream1);
158 
159                 //Match keypoints globally
160                 matcher.matchGlobalAsync(keys1, keypointsLocal, desc1, descriptorsLocal, points_matched_global, classes_matched_global,
161                   stream1, true);
162 
163                       }
164 
165     if(pipeline){
166 
167 
168 
169                  //Grab hold of next image in pipeline
170                  imNext = im_gray;
171 
172                  //Upload NEXT image in pipeline into device memory
173                  imGrayScaleP.upload(imNext);
174 
175                  //Detect keypoints and compute descriptors for next image in pipeline
176                  float pipelineStart = cv::getTickCount();
177                  gpuOrb->detectAndComputeAsync(imGrayScaleP, cv::noArray(), keys1P, desc1P, false, stream1);
178 
179                  //Perform knn (2) matching per keypoint for next image in pipeline
180                  matcher.matchGlobalNoSync(desc1P, matchesGpuP, stream1);
181                  float pipelineEnd = cv::getTickCount() - pipelineStart;
182 
183                          }
184 
185     else{
186 
187                  //Upload CURRENT image in pipeline into device memory
188                  imGrayScaleP.upload(im_gray);
189 
190                  //Detect keypoints and compute descriptors, for current image
191                  float T1 = cv::getTickCount();
192                  gpuOrb->detectAndComputeAsync(imGrayScaleP, cv::noArray(), keys1P, desc1P, false, stream1);
193 
194                  //Perform knn (2) matching per keypoint current image
195                  matcher.matchGlobalNoSync(desc1P, matchesGpuP, stream1);
196 
197                  stream1.waitForCompletion();
198                  float T2 = cv::getTickCount();
199 
200                  matcher.processMatches(matchesGpuP, keys1P, points_matched_global, classes_matched_global, desc1P, keypoints,
201                   descriptors);
202                  }
203 
204     FILE_LOG(logDEBUG) << points_matched_global.size() << " points matched globally.";
205 
206     //Track keypoints
207     vector<Point2f> points_tracked;
208     vector<unsigned char> status;
209 
```

Figure A2: Illustration of uploading of variables into device memory, utilization of the CUDA API functions for the three functions allocated to the GPU as well as an illustration of how the concurrent execution is achieved, in Nebehay's function processframe(). If the first frame is being processed and concurrent execution is desired, the processing pipeline is first *filled* as in line 144. On line 197, the runtime API makes the call for the host to wait for device completion, and the discerning factor between a concurrent and sequential execution. The 'else' statement serves to initiate the sequential computation, illustrated in Figure 5.9, whilst the true statement serves when concurrent execution is enabled.

```
286     FILE_LOG(logDEBUG) << points_matched_local.size() << " points matched locally.";
287
288     //Clear active points
289     points_active.clear();
290     classes_active.clear();
291
292
293     //Fuse locally matched points and inliers
294     fusion.preferFirst(points_matched_local, classes_matched_local, points_inlier, classes_inlier, points_active, classes_active);
295
296     FILE_LOG(logDEBUG) << points_active.size() << " final fused points.";
297
298     //TODO: Use theta to suppress result
299     bb_rot = RotatedRect(center,  size_initial * scale, rotation/CV_PI * 180);
300
301
302     if (pipeline){
303
304             //Synchronise device with host
305             stream1.waitForCompletion();
306
307
308             //Device must be synchronised prior to calling processMatches()
309             matcher.processMatches(matchesGpuP, keys1P, points_matched_global, classes_matched_global, desc1P, keypointsLocal,
310              descriptorsLocal);
311
312             //Update images to be processed
313             imPrevious = imCurrent;
314             imCurrent = imNext;
315
316                 }
317
318
319     else im_prev = im_gray;
320
321
```

Figure A3: Synchronization point in the function processframe() if concurrent execution has been enabled, where the extra image storage on line 313 should be noted.

```
52 void Matcher::matchGlobalAsync(cv::cuda::GpuMat & keypoints, vector<KeyPoint> & keypointsCpu, cv::cuda::GpuMat & descriptors,
Mat & descriptorsCpu, vector<Point2f> & points_matched, vector<int> & classes_matched, cv::cuda::Stream & stream1, bool flag)
53 {
54     //vector<KeyPoint> keypointsCpu;
55     vector<vector<DMatch> > matches;
56     cv::cuda::GpuMat matchesGpu;
57
58     //Queue knnMatch kernel to defined stream
59     bfmatcherAsync->knnMatchAsync(descriptors, data, matchesGpu, 2, cv::noArray(), stream1);
60
61     //Synchronise device
62     stream1.waitForCompletion();
63
64     //Now is the right time to download the keypoints into host device memory to minimize host to device memory transfers
65     gpuOrbConvert->convert(keypoints, keypointsCpu); //Move data from device to host memory
66     descriptors.download(descriptorsCpu); //Move data from device to host memory
67     bfmatcherAsync->knnMatchConvert(matchesGpu, matches); //Move data from device to host memory
68         //std::cout << "size of matches usual function functions/before processing" << matchesGpu.size() << std::endl;
69
70    for (size_t i = 0; i < matches.size(); i++)
71     {
72         vector<DMatch> m = matches[i];
73
74         float distance1 = m[0].distance / desc_length;
75         float distance2 = m[1].distance / desc_length;
76         int matched_class = classes[m[0].trainIdx];
77
78         if (matched_class == -1) continue;
79         if (distance1 > thr_dist) continue;
80         if (distance1/distance2 > thr_ratio) continue;
81
82         points_matched.push_back(keypointsCpu[i].pt);
83         classes_matched.push_back(matched_class);
84     }
85
86 }
87
88 void Matcher::matchGlobalNoSync(cv::cuda::GpuMat & descriptors, cv::cuda::GpuMat & matchesGpu, cv::cuda::Stream & stream1)
89 {
90     //Queue knnMatch kernel to defined stream
91     bfmatcherAsync->knnMatchAsync(descriptors, data, matchesGpu, 2, cv::noArray(), stream1);
92 }
93
```

Figure A4: Function matchGlobalAsync() serves to illustrate how data is moved from the device back to the host. The concurrent equivalent matchGlobalNoSync() however does not return data to the host at this point, as this completed later during synchronisation with the host – illustrated in Figure A.3, line 309.