Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Measurment

Bachelor's Project

Aplication of PREM model on parallel implementation of KCF tracker algorithm

Vít Karafiát

Supervisor: Ing. Michal Sojka, Ph.D.

Study Programme: Open Informatics, Bachelor

Field of Study: Computer systems

May 24, 2018

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Karafiát** Jméno: **Vít** Osobní číslo: **457000**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Aplikace PREM modelu na paralelní implementaci algoritmu KCF trackeru

Název bakalářské práce anglicky:

Application of PREM Model to Parallel Implementation of KCF Tracker

Pokyny pro vypracování:

1. Seznamte se s algoritmem 'Kernelized Correlation Filters' a jeho použitím pro sledování objektů ve video sekvencích. Dále se seznamte s modelem aplikací navrhovaných pro deterministický běh real-time, nazývaným 'Predictable Execution Model' (PREM).
2. Paralelizujte existující implementaci KCF trackeru a její stěžejní část - výpočet FFT - pro běh jak na CPU, tak GPU. Použijte a porovnejte různé technologie, zejména CUDA a OpenMP, případně i OpenCL.
3. Proveďte analýzu (profilování) KCF algoritmu z hlediska četnosti přístupů do paměti a nejkritičtější části algoritmu upravte podle požadavků PREM modelu (rozdělení na prefetch, compute a write-back fáze).
4. Porovnejte výkonnost, úroveň dosaženého determinismu a náročnost manuální konverze pro PREM u různých variant algoritmu. Dále porovnejte výsledky manuální konverze s výsledky dosaženými experimentálním automatickým PREM kompilátorem.
5. Výsledky pečlivě zdokumentujte.

Seznam doporučené literatury:

[1] Joao F. Henriques, Rui Caseiro, Pedro Martins, Jorge Batista: High-Speed Tracking with Kernelized Correlation Filters. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2015
[2] R. Pellizzoni et al.: A Predictable Execution Model for COTS-Based Embedded Systems. 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, Chicago, IL, 2011, pp. 269-279.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Michal Sojka, Ph.D., katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **09.01.2018** Termín odevzdání bakalářské práce: **25.05.2018**

Platnost zadání bakalářské práce:
do konce letního semestru 2018/2019

Ing. Michal Sojka, Ph.D.	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc.
podpis vedoucí(ho) práce		podpis děkana(ky)

Aknowledgements

I would like to express my deepest gratitude to my thesis adviser Ing. Michal Sojka Phd., because without his guidance and persistent help this thesis would not have been possible.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague day 15. 5. 2018 .....

Abstract

In recent years many real-time embedded systems are being built using the Commercial-Off-The-Shelf (COTS) components because of their price. COTS components overall performance is often much higher than specialized custom-made systems used in real-time systems. However, COTS components are typically designed for average case scenario, and little or no attention is put into worst-case timing guarantees required by real-time systems. In this thesis, we implement various parallel and extended versions of the KCF tracker for both CPU and GPU and try to test out the prototype HERCULES compiler, which allows converting automatically parts of the program to conform to PRedictable Execution Model (PREM), which should provide stronger worst-case timing guarantees.

Keywords: KCF tracker, PREM, HERCULES compiler, Real-time, Commercial-Off-The-Shelf (COTS)

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Redefined thesis goals . . . 2
- 1.3 Thesis structure . . . 2

2 Background **3**
- 2.1 Predictable Execution Model . . . 3
  - 2.1.1 Motivation . . . 3
  - 2.1.2 Execution model . . . 3
    - 2.1.2.1 Compatible interval . . . 4
    - 2.1.2.2 Predictable interval . . . 4
  - 2.1.3 HERCULES PREM compiler . . . 4
    - 2.1.3.1 OpenMP . . . 5
    - 2.1.3.2 Compilation process . . . 6
    - 2.1.3.3 Supported loops . . . 6
- 2.2 Visual object tracking . . . 7
  - 2.2.1 Correlation Filter-based Trackers . . . 8
  - 2.2.2 Kernelized Correlation Filter tracker . . . 9
    - 2.2.2.1 Building blocks of KCF tracker . . . 9
    - 2.2.2.2 Used implementation of KCF . . . 9

3 Profiling of original KCF implementation **12**
- 3.1 Overview . . . 12
- 3.2 Original implementation . . . 12
  - 3.2.1 OpenCV . . . 12
  - 3.2.2 Classes . . . 13
- 3.3 Profiling . . . 14
  - 3.3.1 PERF . . . 14
  - 3.3.2 Results . . . 14
    - 3.3.2.1 KCF_VOT binary . . . 15
    - 3.3.2.2 Libopencv core binary . . . 18
- 3.4 Summary of the results . . . 18

4 Parallel and extended implementations **20**
4.1 Target platform . . . 20
4.1.0.1 OpenCV4Tegra . . . 20
4.2 Modifications . . . 21
4.3 Fourier Transforms . . . 21
4.3.1 KCF_Tracker::fft2 . . . 21
4.3.2 KCF_Tracker::ifft2 . . . 23
4.3.3 Problems . . . 24
4.4 FFT class . . . 24
4.4.1 FFTW . . . 26
4.4.1.1 Overview . . . 26
4.4.1.2 Usage in KCF tracker . . . 26
4.4.1.3 CuFFTW . . . 29
4.4.2 CuFFT . . . 30
4.4.2.1 Overview . . . 30
4.4.2.2 Usage in KCF tracker . . . 30
4.4.3 Big batch mode . . . 31
4.5 ComplexMat . . . 32
4.5.1 CPU version . . . 32
4.5.2 GPU version . . . 33
4.5.2.1 CUDA error check . . . 34
4.6 CPU parallel options . . . 34
4.7 HERCULES compiler . . . 34

5 Results **36**
5.1 Benchmark results . . . 36
5.1.1 Interpretation of the results . . . 41
5.2 CuFFT version profiling . . . 41
5.3 FFTW version profiling . . . 43
5.3.1 Gaussian correlation . . . 43
5.3.2 Forward Fast Fourier Transform . . . 44

6 Conclusion **45**

Bibliography **46**

A Contents of CD **49**

List of Figures

2.1 Example of predictable interval. $e_{i,j}^{mem}$, $e_{i,j}^{exec}$ represents maximum execution time for memory and execution phase. Source: [14] 4
2.2 Example of OpenMP offloading. 5
2.3 A block diagram of the compilation steps for the Clang OpenMP offloading process and HERCULES version with added PREM transformation passes. Picture was inspired by [9] 7
2.4 Example of sequence of image frames with object of interest (person) in bounding box (blue). Source: [17] 8
2.5 The basic correlation tracker pipeline. Source: [4] 9
2.6 Pixelwise color name example. Source: [27] 11
2.7 *Process of estimating object scale. When a new frame comes, windows with different sizes are cut by scaling pool. These multi-scale image patches are resized to s_0, then feature vectors with the fixed size are extracted to compute response scores.* Source: [29]. 11

3.1 `cv::Mat` class data layout. 13

4.1 Diagram showing the workflow of `KCF_Tracker::fft2`. After `cv::dft`, `CompleMat`'s method `set_channel` is used to copy data from transform result, which is stored in `cv::Mat` to an instance of `ComplexMat`. 22
4.2 Source code of original `KCF_Tracker::fft2` function. 22
4.3 Diagram showing the workflow of `KCF_Tracker::ifft2` for multichannel input. We can see that after the input `ComplexMat` is transformed to a vector of `cv::Mats`, each representing one channel, the Fourier Transforms are performed the same way as in the `KCF_Tracker::fft2`. 23
4.4 Source code of original `KCF_Tracker::ifft2` function. 24
4.5 A diagram of the `Fftw::forward_window` method. In this figure we show the execution of the `plan_fw_all_scales` plan on N scales each consisting of 44 feature channels. 28
4.6 A diagram, showing workflows for all plans in `Fftw::inverse`. The `cv::Mat` stores each channel pixel by pixel after other channels unlike `ComplexMat`, which stores each channels matrix by matrix after other channels. 29

5.1 Datasets used for benchmarking, starting from the top left: Bag, Car2, Car1, Ball1, Pedestrian2, Nature. All screenshots of the datasets were resized to 128×128 so they fit inside one figure. 37

5.2 Pedestrian2 dataset with default scaling. This dataset was not resized, because the initial bounding box was smaller than 100×100. 37
5.3 Ball1 dataset with default scaling. This dataset was not resized, because the initial bounding box was smaller than 100×100. 38
5.4 Car1 dataset with alternative scaling to power of two feature channels. 38
5.5 Car1 dataset with default scaling to power of two feature channels. 39
5.6 Bag dataset with default scaling. 39
5.7 Nature dataset with default scaling. 40
5.8 Nature dataset with alternative scaling to maximum allowed size of feature channel for CuFFT versions. 40
5.9 CuFFT version without big batch mode. 42
5.10 CuFFT version with big batch mode. 42
5.11 CuFFTW version with big batch mode. 42

List of Tables

3.1 Profiling results for `KCF_Tracker::track`. The threshold was set to 5% of total cache misses measured. 15
3.2 Profiling results for `KCF_Tracker::gaussian_correlation`. The threshold was set to 2% of total cache misses measured. 15
3.3 Profiling results for `KCF_Tracker::get_features`. The threshold was set to 2% of total cache misses measured. 16
3.4 Profiling results for `KCF_Tracker::fft2`. The threshold was set to 2% of total cache misses measured. 17
3.5 Profiling results for `libopencv_core.so.3.4.1`.. 18

5.1 Profiling results for `KCF_Tracker::track`. The threshold was set to 5% of total cache misses measured. 43
5.2 Profiling results for `KCF_Tracker::gaussian_correlation`. The threshold was set to 4% of total cache misses measured. Rest of the operation was under 1 %. 44

List of acronyms

COTS Commercial-Off-The-Shelf

PREM PRedictable Execution Model

WCET Worst-Case Execution Time

DRAM Dynamic Random Acces Memory

HERCULES High-Performance Real-time Architectures for Low-Power Embedded Systems

CPU Central Processing Unit

GPU Graphical Processing Unit

CUDA Compute Unified Device Architecture

PC Program Counter

SIMD Single Instruction Multiple Data

ELF Executable and Linkable Format

IR Intermediate Representation

SM Streaming Multiprocessors

API Application Programmable Interface

CFTs Correlation Filter-based Trackers

KCF Kernelized Correlation Filter

CSK Circular Structure Kernel

MOSSE Minimum Output Sum of Squared Error

HOG Histogram of Oriented Gradients

FHOG Felzenszwalb's HOG

CN Color Names

IPP Integrated Performance Primitives

VOT Visual Object Tracking

SoM System-on-Module

CCS Complex-Conjugate Symmetry

DFT Discreate Fourier Transform

FFT Fast Fourier Transform

FFTW Fastest Fourier Transform in The West

PMU Performance Monitoring Unit

Chapter 1

Introduction

1.1 Background

Traditionally real-time systems have strict requirements on response time, within specified time constraints, often referred to as "deadlines". These systems are often constructed from hardware and software components specifically designed for real-time. Development of these components is not only costly but also lengthy and with many real-time systems nowadays making use of media streaming also require significant performance in terms of the bus and network bandwidth, processors, and memory [1].

These make the use of COTS hardware and software components more attractive. They usually have higher performance than their real-time counterparts and because of their mass production are often much cheaper. The problem with these components is that they are built with the focus on the average performance and not worst case scenario and reliability. One of the proposed solutions to improve the Worst-Case Execution Time (WCET) of COTS components is PRedictable Execution Model (PREM). It introduces programming guidelines that allow transforming parts of the code to highly predictable versions in its memory access behavior, improving the overall WCET. This is done by separating the program into different phases based on their use of shared resources, e.g., system DRAM. This allows to construct the schedules in which only single execution unit, i.e., core, is accessing a shared resource at a time, resulting in reduced pessimism in the WCET calculation, because each phase can be calculated in isolation, i.e., interference from parallel accesses do no longer need to be considered [9].

In this thesis, we aim at applying the PREM to visual object tracking algorithm, i.e., Kernelized Correlation Filter (KCF) tracker. We first analyze the tracker's performance and memory access patterns. Then, as PREM is effective primarily for parallel algorithms, we develop a parallel implementation of the tracker using various technologies: CUDA, FFTW. Finally, we apply the PREM model by using an experimental compiler that allows automating the transformation of the code to conform to PREM requirements. This experimental compiler is part of the European project HERCULES[1] on which the Czech Technical University in Prague works together with ETH Zürich, Airbus, UNIMORE, Evidence srl, PITOM and Magneti Marelli. To our knowledge, the compiler was currently only tested on smaller

[1] <https://hercules2020.eu/>

laboratory projects, so there is no information on performance or use with larger projects, such as the KCF tracker.

1.2 Redefined thesis goals

During the work on the thesis, the original goals has been refined. In accordance to adviser's decision, the updated goals are:

1. Profile the KCF tracker to analyze the tracker's performance and memory access patterns.
2. Develop a parallel implementation of the tracker using various technologies: CUDA, FFTW.
3. Use experimental HERCULES compiler to automatically transform the KCF tracker code to comply with the PREM model.
4. Compare the benchmarking results of various versions of the KCF tracker.

1.3 Thesis structure

In chapter two we give the theoretical overview for the PREM, the HERCULES compiler, and the KCF tracker. In chapter three we present the original version of the tracker and its profiling results. Chapter four is dedicated to the various parallel and extended versions that we implemented and also the use of the HERCULES compiler with the KCF tracker. In chapter five we show the benchmarking results for all of the available version of the KCF tracker, and in the last chapter we will sum up our work and discuss possible future improvements.

Chapter 2

Background

2.1 Predictable Execution Model

2.1.1 Motivation

In recent years real-time embedded systems are being built using the Commercial-Off-The-Shelf (COTS) components because of their price. COTS components' overall performance is often much higher than specialized custom-made systems used in real-time systems, e.g., Boeing777 SAFEbus with 60 $Mbit/s$ versus PCI Express with 16 $Gbyte/s$ [15]. However, COTS components are typically designed with average case scenario and little or no attention to worst-case timing guarantees required by real-time systems [14].

Currently, many industrial real-time systems use single-core devices, for which it is much easier to derive solid guarantees about their timing behavior [10]. But with the increase of interest in autonomous vehicles in recent years and their high computational requirements, use of heterogeneous many-cores in these systems is being researched. In contrast to single-core processors, the heterogeneous many-cores share main memory (DRAM) and interconnect between several actors (e.g. CPU, GPU, Ethernet) [10]. This design can result in one core affecting other cores, resulting in worse latencies as the shared resource has a limited amount of requests per time unit. In this case of heavy resource sharing calculation of WCET is significantly complicated [16], with estimations being so pessimistic that all the advantages of parallelism are lost. PREM introduced by Pellizzoni et al. in [14], is promising a way to overcoming all of these issues.

By separating code according to access to the shared resource, e.g., system DRAM. PREM allows system schedules to be constructed in such a way that only one execution unit, i.e., core, is accessing a shared resource at a time. Such a schedule allows the WCET of each phase to be calculated in isolation, without the need to account for parallel access interference, resulting in tighter bound and more optimistic WCET [9].

2.1.2 Execution model

PREM divides program's tasks into sequence of non-preemptive scheduling intervals. These intervals are divided into compatible and predictable intervals. To determine which

part of the task should be the predictable or compatible interval, code profiling should be first performed to find the parts, in which the task performs most of its memory accesses.

2.1.2.1 Compatible interval

Compatible intervals are compiled and executed normally. During these intervals, cache misses can happen at any time. OS system calls are also allowed to be performed, with blocking calls having bounded blocking time [14].

2.1.2.2 Predictable interval

Predictable intervals are divided into two phases: *memory phase* and *execution phase*. During the memory phase, data required for the computation in the predictable interval are brought to local storage from the shared resource, e.g. system DRAM. This results, together with the fact that no system calls are allowed in the execution phase, in no memory accesses.

Another essential property of predictable interval is fixed execution time, which is determined offline by summing the maximum execution time for memory phase and execution phase and forced at run-time. In case that interval finishes before the set execution time, it will busy wait. Example of a predictable interval can be seen in figure 2.1.

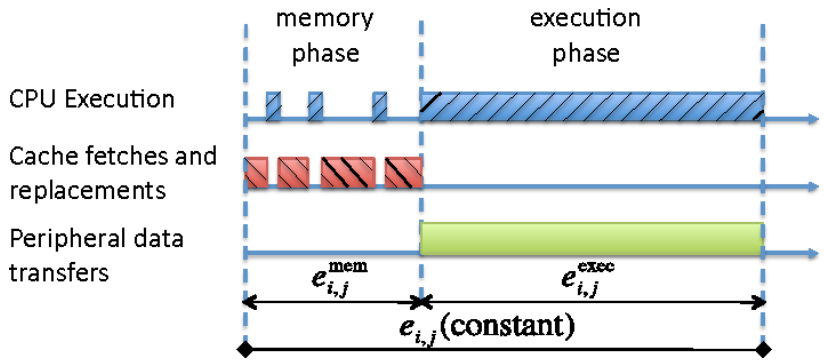

Figure 2.1: Example of predictable interval. $e^{mem}_{i,j}$, $e^{exec}_{i,j}$ represents maximum execution time for memory and execution phase. Source: [14]

2.1.3 HERCULES PREM compiler

The HERCULES compiler is OpenMP compiler, which compiles parts of C/C++ source code annotated by OpenMP for accelerator offloading into separate CPU and accelerator parts, with predictability transformation for accelerator part added to the compiling process, for the code to conform to the requirements of the PREM. Currently, the prototype can provide basic support for the PREM on the NVIDIA platform. Development focus of this compiler is on predictability transformations of the accelerator part of the program and functional prototype on NVIDIA GPUs for testing [9].

The HERCULES compiler is built upon the Clang-YKT fork of LLVM, which focuses on the implementation of OpenMP offloading following the OpenMP standard 4.5. Thanks to this, HERCULES compiler has early access to OpenMP new offloading features.

2.1.3.1 OpenMP

OpenMP allows programmers to turn sequential code to parallel code, with minimal effort, through its directives, given as compiler pragmas. By annotating parts of the code with OpenMP pragmas that programmer wishes to run in parallel. The compiler can transform them without explicit management by the programmer. This level of abstraction allows easy porting of the code to other platforms as long as the specific platform provides the OpenMP runtime environment. In case the OpenMP is not supported on the platform original sequential version of the code is used. Programmer productivity also increases because the management of parallelism is abstracted away.

OpenMP offloading

In version 4.0 accelerator offloading was introduced to the OpenMP standard. This enables to create a program which can automatically be parallelized for different architectures. If the program executes on a system without accelerator, the CPU version of the code executes instead.

The code specified for accelerator offloading is annotated by `target` directive. The compiler then creates both CPU and accelerator version of the annotated code. The `target` directive itself does not introduce parallelism; additional OpenMP directives have to be used together with it to express parallelism. With `target` directive, directives for data movement were also introduced because the OpenMP assumes that the CPU and the accelerator have two distinct memory spaces. For NVIDIA GPUs OpenMP runtime library uses calls to CUDA's Application Programmable Interface (API), for data movement. One thing that should be noted is that accessing CUDA runtime environment has high impact on performance and predictability and should be avoided if possible [9].

```
//Host device
#pragma omp target
        #pragma omp parallel for private(i)
        // Target device
        for(int i = 0; i < n;++i){
                y[i] += a * x[i]
        }
//Host device
```

Figure 2.2: Example of OpenMP offloading.

The one thing programmer has to keep in mind when using OpenMP offloading directive is that depending on the accelerator used in the system, different constructs will perform differently well. In the case of HERCULES, which focuses on GPUs, the construct that

maps well are loops, as the parallel execution of many loop iterations fit well with the Single Instruction Multiple Data (SIMD) model used on GPUs [9]. GPUs use SIMD model because they consist of multiple simple cores, which share Program Counter (PC), resulting in same instruction on all cores but executed on different data.

2.1.3.2 Compilation process

Compiling process of OpenMP's compiler (Clang) for code parts selected for offloading by `target` directive, compiles the code both for the host and all accelerator targets specified. Before the code generation in the LLVM backend, the remaining OpenMP directives are expanded for both host and accelerator processes. For the host code call to OpenMP's runtime library `target()` is added, which tries to find available accelerator for offloading during the execution of the code. If no available accelerator is found, host code version executes instead.

For the separate device code, its compilation follows cross-compilation tool-chain specific to the device, with the device binary being added to host ELF as data in the end. This data is then launched by OpenMP runtime library when the host program reaches the specific code [9].

The HERCULES adds to this compilation the PREM transformation of the accelerator code. The transformation is implemented as LLVM passes invoked by Clang before the device Intermediate Representation (IR) is passed to the backend compiler [9].

2.1.3.3 Supported loops

Currently, the HERCULES compiler adds PREM-enabling transformation to loops with `#pragma omp target teams distribute parallel for` directive. As described in 2.1.3.1 `target` directive is used to specify code region for offloading. `teams distribute` distributes the offload over several accelerator clusters, in the case of CUDA architecture these clusters are Streaming Multiprocessors (SM). Lastly, the `parallel for` directive performs the parallelization of the loop itself.

Loops specified in the way above, are marked by HERCULES's *MarkLoopsToTransform* pass and transformed by following passes according to PREM. In case that no such loops are found remaining passes will do nothing [9].

OpenMP's pragma mapping to CUDA

Starting with `target` directive, its CUDA equivalent `__global__` decorator is used, when defining a CUDA kernel. With only `target`, the code is not yet parallel and only offloaded as one-threaded sequential version similar to a kernel with only one block and one thread. To create more threads per block and blocks `parallel for` with `teams distribute` directives are used. The number of blocks and threads in blocks are specified with `num_teams()` and `num_threads()` clauses.

OpenMP considers host and device memory to be discrete, and as such, all data required in offloaded code has to be explicitly copied to device memory. The required data are specified by `map()` clause, which for its data movement uses `cuMemcpy()` function from

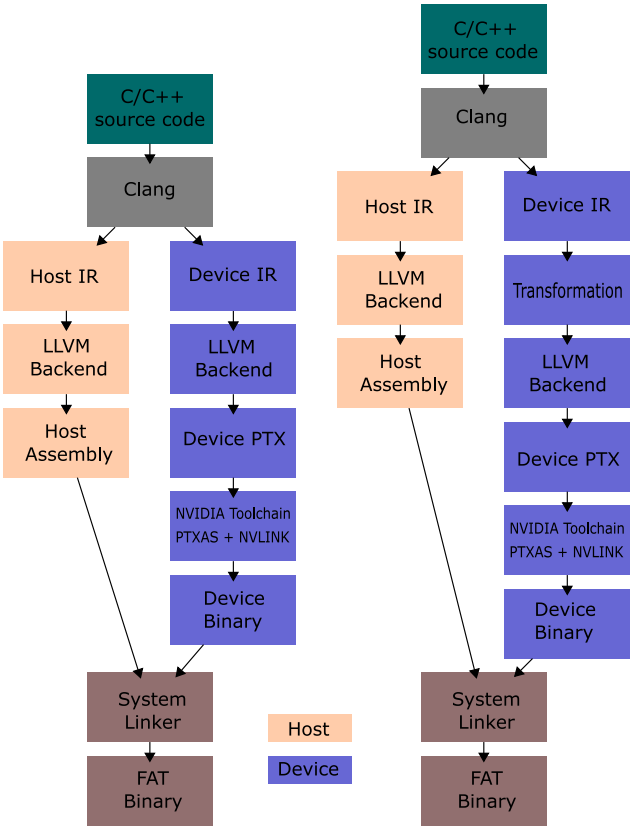

Figure 2.3: A block diagram of the compilation steps for the Clang OpenMP offloading process and HERCULES version with added PREM transformation passes. Picture was inspired by [9]

CUDA runtime library. Calls to CUDA runtime environment and data movement from host to device memory are unfortunately costly and should be avoided if possible. These host-initiated data transfers with CUDA API are scheduled to be addressed by HERCULES's CPU PREM support [9].

2.2 Visual object tracking

The goal of visual tracking can be described as follows. Given a sequence of image frames, that are taken from changing world, wherein the first frame is the object of interest (target) demarcated by bounding box, approximate the trajectory of the target in the following frames. An external source, e.g., user, creates the bounding box defining the target in the first frame. The target position is usually denoted more generally as a target pose, which may represent additional geometrical properties such as target's size or rotation [28]. Image tracking algorithms also have to be able to overcome various tracking challenges, e.g., extreme global and local illumination changes.

The use of visual object tracking ranges from games (e.g., Kinect), augmented reality,

robotics to pedestrian tracking in the airport [1] and the car industry, many examples of visual object tracking exist [2][3][4].

Figure 2.4: Example of sequence of image frames with object of interest (person) in bounding box (blue). Source: [17]

.

There are multiple tracking methods some of the most influential are Gradient-based Tracking, Part-based Tracking, Tracking by Detection, Tracking by Segmentation and Deep Neural Networks. For more information about these methods, we refer to [28]. In this thesis, we will only focus on Correlation Filter-based trackers, which KCF tracker is part of.

2.2.1 Correlation Filter-based Trackers

Correlation Filter-based Trackers (CFTs) are a group of visual object trackers that found a massive increase in interest over the recent years mainly thanks to the Minimum Output Sum of Squared Error (MOSSE) tracker [4, 28]. General CFT work-flow according to the present correlation filter-based tracking methods, as described in [4, 28] can be summarized like this:

From the first frame, where the bounding box defines the object of interest a patch is extracted, on which the initial filter is learned. The filter should, when convolved with the target's location produce correlation peaks for the object of interest and low response for the background. After this in each subsequent frame, the patch at the previous predicted position is cropped for detection. Following this various features are extracted from the patch's raw input data (e.g., Histogram of Oriented Gradients (HOG) and Color Names (CN) features). Afterward, cosine window is usually applied for smoothing the boundary effects. After this features are transformed using the Discreate Fourier Transform (DFT) to the frequency domain, where they are element-wise multiplied with the learned filter. The result is then transformed back to the real domain using inverse DFT to produce spatial confidence map (response map). The position with the max value in the response map is the new position of the target and is then used for the update of the correlation filter. Because

[1] <http://www.fujitsu.com/uk/solutions/industry/transport/aviation/>
[2] <https://www.google.com/selfdrivingcar/>
[3] <http://www.lexus.com/models/LS/safety>
[4] <https://www.mobileye.com/>

only the DFT of correlation filter is required for detection, training and updating procedures are all performed in frequency domain. This whole workflow is shown in figure 2.5.

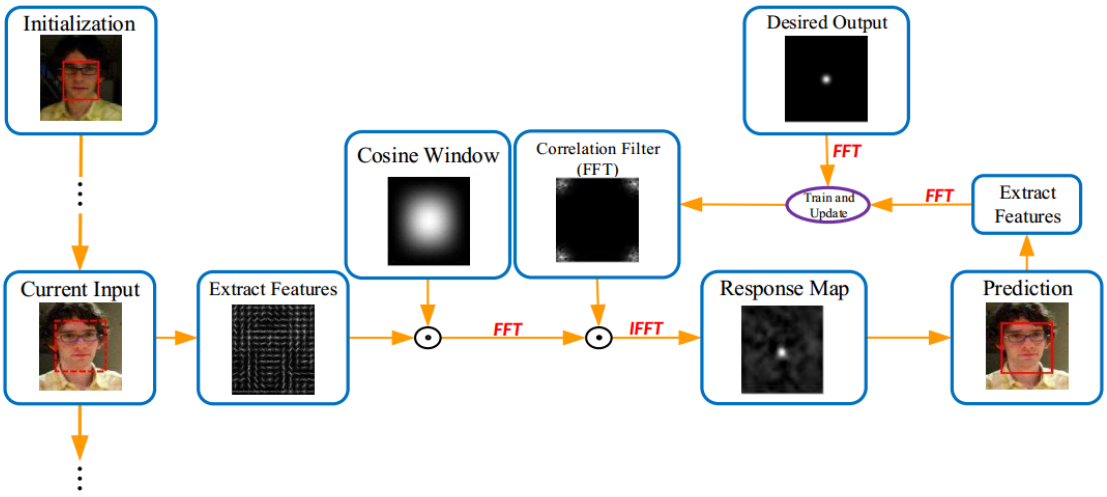

Figure 2.5: The basic correlation tracker pipeline. Source: [4]

.

2.2.2 Kernelized Correlation Filter tracker

KCF tracker is part of CFTs and based on Circular Structure Kernel (CSK) tracker. In this section, we briefly describe main features of the KCF tracker, followed by a description of our used KCF tracker implementation .

2.2.2.1 Building blocks of KCF tracker

The KCF tracker itself is based on CSK tracker [12], which addresses the drawback of the MOSSE tracker (i.e., lack of training examples and support of complex features or multiple channels), by introducing circulant matrix structure and Ridge Regression problem to kernelize the correlation filters and also the support for multi-channel features. For more details on the use of Ridge Regression and circulant matrix structures in the KCF tracker, we refer to [11, 28], especially to [29], which describes similar KCF implementations to one used in our thesis.

2.2.2.2 Used implementation of KCF

We use C/C++ implementation of KCF tracker developed by Tomáš Vojíř[5], which uses together with multichannel features (HOG, RGB channels, and CN) also adaptive scale tracking with the scale pool, improving the ability of tracker to respond to scale changes of the object. In the following sections, we introduce these techniques.

[5] <https://github.com/vojirt/kcf>

Histogram of oriented gradients

The following description of HOG is of the original variant proposed in [5]. The HOG is the primary feature descriptor in our KCF tracker. The version used in our program is Felzenszwalb's HOG (FHOG) [8], which was shown to achieve better performance [8] over the Dalal and Triggs version [5]. The following description of how HOG works is heavily based on [18].

HOG is feature extraction method for images. It tries to capture the shape of structure in the region by extracting information about the gradients. In this description, we assume that the image is a grayscale image.

Firstly for each pixel (x, y), we calculate its gradient vector along the x-direction and y-direction using the finite difference filters, $[-1, 0, +1]$ and its transpose. From the resulting gradient vector, we can calculate its angle $\theta(x, y)$ and magnitude $r(x, y)$. The angle is constrained between 0 and 180 degrees (In FHOG angle is not constrained.).

The image is then divided into non-overlapping cells of size $C \times C$ pixels. For each cell, we compute a histogram of oriented gradients with B bins. Gradient vector contribution is equal to its magnitude. The vector contribution is split between two closest bins to reduce aliasing. Bins are number 0 to $B-1$ with width of $w = \frac{180}{B}$. Bin in position i has boundaries $[w \times i, w \times (i+1)]$ and center $c_i = w(i + \frac{1}{2})$. For a pixel (x, y) with magnitude $r(x, y)$ and angle $\theta(x, y)$ votes to bins B_j and B_{j+1} are:

$$\text{Contribution to } B_j = r(x,y)\frac{c_{j+1} - \theta(x,y)}{w} \quad \text{where } j = \left\lfloor \frac{\theta(x,y)}{w} - \frac{1}{2} \right\rfloor \mod B$$

$$\text{Contribution to } B_{j+1} = r(x,y)\frac{\theta(x,y) - c_j}{w} \quad \text{where } j+1 = (j+1) \mod B$$

The resulting histogram of the cell is a vector with B nonnegative entries.

Following this cells are grouped into blocks of cells, in the case of [5] 2×2 are used cells so that each block has $2C \times 2C$ pixels. Each block overlaps with its horizontal and vertical neighborhood block by two cells resulting in 50% overlap. We concatenate cells histograms together resulting in a block feature vector $\mathbf{b}$ of size $4 \times B$, which we normalize by its Euclidean norm.

Resulting block features are concatenated into a single HOG feature vector $\mathbf{h}$, which is normalized in three steps:

$$\mathbf{h} \leftarrow \frac{\mathbf{h}}{\sqrt{\|\mathbf{h}\|^2 + \epsilon}}$$

$$\text{For each entry } h_i \leftarrow \min(h_i, \tau) \text{ where } \tau = 0.2$$

$$\mathbf{h} \leftarrow \frac{\mathbf{h}}{\sqrt{\|\mathbf{h}\|^2 + \epsilon}}$$

The second step ensures that very large gradients do not influence the resulting histogram too much, which would destroy image detail.

Difference between the original HOG and FHOG is that in each block vectors are not concatenated but the four cell's histograms are averaged together, and their norms are added as additional entries to the feature vector, resulting in a smaller vector. For more details, we refer to [2, 8, 18].

Color names

CN are linguistic labels which humans use to communicate color. Computational color naming tries to learn a mapping from pixel value to color name labels. Most computer vision works consider eleven basic terms of English language: black, blue, brown, gray, green, orange, pink, purple, red, white, and yellow. Example of this mapping can be seen 2.6. We refer for more details to [27]

Figure 2.6: Pixelwise color name example. Source: [27]

Scale pool

Following description of scale pool was taken from [29]:

Suppose the original template size is s0, and then we define a scaling pool $S = \{a_1, a_2, \cdots, a_N\}$ *of* N *scale parameters. In a new frame,* N *windows with different sizes* $s_i = a_i s_0$, $i = 1 \cdots N$ *are cropped around the previous position; then the sample patches are resized to* s_0 *and correlated with the learned correlation filter, thereby a maintained response vector is used to estimate the new state. 2.7 shows this scale searching strategy.*

In our case $N = 7$. For more details on the scale pool and its use in KCF tracker we refer to [29].

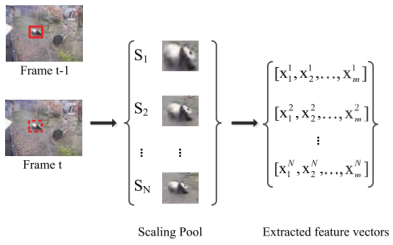

Figure 2.7: *Process of estimating object scale. When a new frame comes, windows with different sizes are cut by scaling pool. These multi-scale image patches are resized to* s_0*, then feature vectors with the fixed size are extracted to compute response scores.* Source: [29].

Chapter 3

Profiling of original KCF implementation

3.1 Overview

In this chapter, we present the original implementation of the of the KCF tracker. Followed by it are profiling results. We performed the profiling on Intel(R) Core(TM) i7- 2620M CPU @ 4M cache, up to 3.4 GHz. Original single-thread implementation runs only on CPU, so no GPU profiling was necessary. For profiling Performance Events for Linux (PERF) tool was used, together with Hotspot[1], which is visualizer for perf results. It also enables to run perf directly from its GUI.

3.2 Original implementation

The original implementation of the KCF tracker was developed to compete in Visual Object Tracking (VOT) challenges[2]. VOT challenges is a collection of datasets that allows a precisely defined and repeatable way of comparing short-term[3] trackers.

3.2.1 OpenCV

The KCF tracker implementation we used is based on OpenCV library. OpenCV is an open source library, used for Computer Vision applications development, featuring over 2500 low- and mid-level image processing and computer vision functions [26]. Its developments began by Intel in 1999; it was designed with a focus on computational efficiency and real-time application [3]. OpenCV is primarily written in C/C++, but also has Python, Java and Matlab interfaces and runs under Windows, Linux, and Mac OS X.

The primary focus of OpenCV is to process and manipulate images, which on digital devices are stored as numerical matrices containing the pixel information of the image. It

[1] <https://github.com/KDAB/hotspot>

[2] <http://www.votchallenge.net/>

[3] Short-term tracking is focused on precise estimation of the object of interest over a smaller frame sequence (usually max. 1000 frames), with no necessarily required recovery of the tracker, when it fails.

is therefore important that we introduce the basic image container called `cv::Mat` that OpenCV library uses to store and handle images [24].

The `cv::Mat` class is separated into two data parts: the matrix header and a pointer to the matrix containing the pixel values. The matrix header contains the information about the matrix, e.g., its size, the address where it is stored, the method used for storing. The header sizes are constant. However, the size of the matrix may vary from image to image. The `cv::Mat` stores the underlying data matrix as one-dimensional array pixel by pixel. For multichannel images (e.g., RGB = three channels, grayscale = one channel)) the channel's pixel values are for each pixel stored behind each other. This data layout can be seen in figure 3.1.

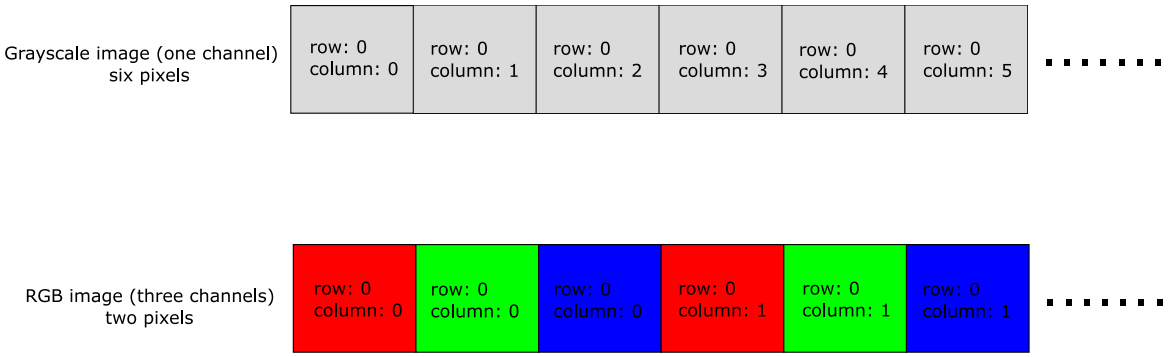

Figure 3.1: `cv::Mat` class data layout.

3.2.2 Classes

In this section, we describe the classes in the original implementation of the KCF tracker and what they are used for. When we in the following profiling sections talk about class functions and methods from `kcf_vot` binary, which is the compiled KCF tracker. We refer to class functions and methods from the following classes.

- **VOT**

  Class used for parsing VOT inputs and providing an interface for image loading and storing output.

- **KCF_Tracker**

  Primary class of the tracker, which performs the tracking itself.

- **ComplexMat_**

  Template class used as basic image container in the frequency domain.

- **FHoG**

  C/C++ wrapper class for FHOG from Piotr Matlab toolbox[4].

- **CNFeat**

  Class used for extracting the CN from the patch.

[4] <https://pdollar.github.io/toolbox/>

3.3 Profiling

3.3.1 PERF

PERF provides a kernel SYSCALL (`perf_event_open()`) and tools to help analyze program performance. The user has an option to either use perf command or implements his tools on top of the `perf_event_open()` system call for profiling.

Perf supports monitoring of multiple types of measurable events. There are two categories of events: software events and hardware events. Software events are pure kernel counters (e.g., context-switches), for hardware events CPU's Performance Monitoring Unit (PMU) is used. PMU provides a list of events measuring micro-architectural events (e.g., cycles, cache misses). The available list of hardware events depends on the CPU type and model.

Perf is based on event-based sampling. A sample is recorded when the sampling counter overflows. When the counter overflows, kernel records information, i.e., a sample, about the program execution. The information that gets recorded depends on the event specified by the user and the tool. For all events type the instruction pointer is also recorded, i.e., where was the program when it was interrupted. One thing to note is that the place of the interrupt and the place of the counter overflow can be several dozen instructions apart on the modern processors and should be remembered when interpreting the results [20].

3.3.2 Results

Selected profiling events were CPU cycles (cycles), and cache misses. On Intel CPUs, perf maps the cycles event to `UNHALTED_CORE_CYCLES`. This event in the presence of CPU frequency scaling, as was in our case, does not maintain a constant correlation to time [20]. The cache miss event in perf is counted only if all cache levels had missed.

We ran the following command: `perf record -event cycles:u,cache-misses:u --call-graph dwarf`, where `event` flag is used to specify which events we want to record. `cycles` and `cache-misses` specify that we want to record CPU cycles and cache misses. `u` specify that we want to record events only on user level and ignore kernel level events. Finally, the *–call-graph dwarf* flag outputs the perf data in a Dwarf format, which Hotspot needs to be able to visualize the perf data. The Dwarf is a standardized debugging format, which was originally designed along with Executable and Linkable Format (ELF), for more information about Dwarf we refer to [6].

Used OpenCV version in profiling was 3.4.1[5]. Following results in tables are displayed in a top down view, ordered by inclusive cache misses in descending order. Inclusive meaning that the values displayed are function's self-cost summed with the cost of children functions called from that function. We first look at `kcf_vot` binary profiling results, where 60.4% of all cycles and 87.6% of all cache misses measured happened. Later we will also present results from OpenCV's `libopencv_core.so.3.4.1.` binary, where the majority of remaining event counts happened.

[5] <https://opencv.org/opencv-3-4-1.html>

3.3.2.1 KCF_VOT binary

For `kcf_vot` binary we will focus on `KCF_Tracker::track` function, where the process of finding a new position of the target and update of the filter takes place. The table 3.1 presents profiling results for `KCF_Tracker::track` function.

Symbol	Cache misses (inclusive)	Cycles (inclusive)
`KCF_Tracker::track`	83.7%	58 %
├─ `KCF_Tracker::gaussian_correlation`	├─ 42.6%	├─ 28.8 %
├─ `KCF_Tracker::get_features`	├─ 26.5%	├─ 15.6 %
└─ `KCF_Tracker::fft2`	└─ 5.37%	└─ 10.5 %

Table 3.1: Profiling results for `KCF_Tracker::track`. The threshold was set to 5% of total cache misses measured.

As we can see most of the cache misses happen in `KCF_Tracker::gaussian_correlation`. Here the Gaussian kernel from extracted features is cross-correlated with interpolation model from the previous frame. This function is also called during initialization of correlation filter in `KCF_Tracker::init`, with the exception that autocorrelation happens and not cross-correlation. For more information about cross-correlation and autocorrelation, we refer to [11]. In `KCF_Tracker::get_features`, HOG features are calculated together with extraction of CN and RGB channels from the patch that is obtained from the bounding box. Resizing of the patch with the object of interest also takes place here depending on the current scale. `KCF_Tracker::fft2` function perform the forward DFT on extracted features that is performed after their extraction from the sub-window. In the following sections, we will take a closer look at profiling results for each of these functions.

Gaussian Correlation

Symbol	Cache misses (inclusive)	Cycles (inclusive)
`KCF_Tracker::gaussian_correlation`	42.6%	28.8 %
├─ `KCF_Tracker::ifft2`	├─ 22.1%	├─ 8.67 %
├─ `ComplexMat_<float>::mat_mat_operator`	├─ 6.35%	├─ 12.2 %
└─ `ComplexMat_<float>::conj`	└─ 2.14%	└─ 3.26 %

Table 3.2: Profiling results for `KCF_Tracker::gaussian_correlation`. The threshold was set to 2% of total cache misses measured.

First important operation, which happens in `KCF_Tracker::gaussian_correlation` is the calculation of complex conjugate of extracted features in the frequency domain, represented by `ComplexMat_<float>::conj`, followed by element-wise multiplication of ex-

tracted features with interpolation model in frequency domain, which is represented by `ComplexMat_ <float>::mat_mat_operator` class function. As we can see in the table 3.2, 12.2 % of all cycles were spent in `ComplexMat_<float>::mat_mat_operator` function. This function implements all matrix operations for matrices with the same number of channels. In this case, it has to perform element-wise multiplication over all feature channels (44 channels if HOG, CN, and RGB channels are used). `ComplexMat_<float>::conj` internally uses `ComplexMat_<float>::mat_const_operator`, which implements matrix operation with constants, e.g., an addition of constant to a matrix and works similarly to the `ComplexMat_<float>::mat_mat_operator`. For both operator functions, a new instance of `ComplexMat_` is created, with copied data from left-hand operand. Here also most of the cache misses were measured for both functions. This new instance is then returned as a result of the matrix operation.

Second significant operation of `KCF_Tracker::gaussian_correlation` is inverse DFT, represented by `KCF_Tracker::ifft2`. With its 22.1% of total cache misses it is the biggest hotspot in the whole program. Two functions in `KCF_Tracker::ifft2`, where most of the cache misses happen, are `cv::dft` and `cv::merge`. `cv::dft` function implements OpenCV's version of DFT. `cv::merge` is here used for merging of all feature channels after inverse DFT into one `cv::Mat`.

In the `KCF_Tracker::gaussian_correlation` itself 8.7% of cache misses were measured, which can be accounted to the summation of feature channels over third dimension and the calculation of the correlation of the Gaussian kernel with the interpolation model.

Extraction of feature descriptors

Before the extraction of the features begins the subwindow is taken from the image from the current location of the bounding box. Following this HOG is calculated on the grayscale version of the subwindow, and depending on the options, RGB channels together with CN are added to HOG results. The final vector of feature channels equals to 44 matrices of size equal to dimensions of the subwindow divided by the size of the cell in the HOG. The size of 44 matrices is only true if all available feature descriptors are used, i.e., HOG, CN, and RGB. The result of profiling is in table 3.3.

Symbol	Cache misses (inclusive)	Cycles (inclusive)
`KCF_Tracker::get_features`	26.5%	15.6%
├─ `KCF_Tracker::get_subwindow`	├─ 8.09%	├─ 0.44%
├─ `gradMag`	├─ 5.05%	├─ 1.32%
├─ `cv::resize`	├─ 3.85%	├─ 3.47%
└─ `fhog`	└─ 2.81%	└─ 6.57%

Table 3.3: Profiling results for `KCF_Tracker::get_features`. The threshold was set to 2% of total cache misses measured.

`KCF_Tracker::get_subwindow` creates subwindow of image input centered around coordinates from input parameters. First, the fitting of the image to target coordinates and

setting of border extensions takes place, followed by a call to `cv::copyMakeBorder`, which returns subwindow from image input. If any pixels are outside of the image, they will replicate the values at the borders. If the input parameters are out of an image or if four corners of calculated subwindow create zero dimension subwindow, the function will return `cv::Mat` set to the desired size, with all values set to zero. Most of the cache misses in `KCF_Tracker::get_subwindow` function happens, during the copying of image input to the middle of destination image, which internally uses `__memset_sse2_unaligned_erms`. It accounts for 6.23% of total cache misses measured in `KCF_Tracker::get_subwindow`.

`fhog` function computes Felzenszwalb's HOG (FHOG) features. FHOG is fast variant of HOG used by Felzenszwalb in [7]. Original function comes from Piotr toolbox and requires SSE2[6] instruction support to compile and run. Both `fhog` and `gradMag` are called from `FHog::execute`, which is C/C++ wrapper function created for Piotr toolbox. Perf has some margin of error in its results and incorrectly showed in the call graph, that both of these functions were directly called from `KCF_Tracker::get_features`. In `gradMag` gradient magnitude and orientation is computed at each location. Most of the cache misses were measured in the `gradMag` function itself. For `fhog` main hotspot was again `__memset_sse2_unaligned_erms`. Here it is used in `wrCalloc`, which is C/C++ wrapper function for `calloc`. `wrCalloc` is called when allocating memory for histograms.

`cv::resize` is OpenCV function used for resizing of the matrices. In KCF tracker depending if either we are downsampling the frame or not, pixel area relation is used for the former or bilinear interpolation for the latter. It is called once for both RBG and grayscale frame. Most cache misses were measured in OpenCV library function `ippcviResizeLinear`, used during the bilinear interpolation.

Forward Fast Fourier Transform

In this section, we will look at `KCF_Tracker::fft2` function. This function is used after obtaining the vector of features from `KCF_Tracker::get_features`. Two primary operations performed in the function are matrix multiplication between each feature channel and cosine window and forward DFT performed on the product of the multiplication. Perf results are in the following table:

Symbol	Cache misses (inclusive)	Cycles (inclusive)
`KCF_Tracker::fft2`	5.37%	10.5%
└ `cv::dft`	└ 2.52%	└ 7.11%

Table 3.4: Profiling results for `KCF_Tracker::fft2`. The threshold was set to 2% of total cache misses measured.

`cv::dft` is, as already mentioned OpenCV's function for calculation of DFT. In original KCF tracker implementation `cv::dft` is used for all DFTs, it, however, does not support execution of multiple transforms of same size together, which results in function call for

[6] <http://softpixel.com/~cwright/programming/simd/sse2.php>

every feature channel. Cache misses here are mainly divided between `cv::dft` and for cycle in `KCF_Tracker::fft2` over feature channels in the `std::vector`. Matrix multiplication between cosine window and channels has only 0.246% of total cache misses.

3.3.2.2 Libopencv core binary

In this section, we discuss profiling results for `libopencv_core.so.3.4.1.` binary, where the majority of remaining event counts happened, especially Central Processing Unit (CPU) cycles. We will present two main hotspots. During profiling, perf was able to create only partial call-graph. This is the reason, why event counts for some OpenCV core library functions were not part of KCF tracker functions. To create complete call-graph, to interpret some of the results of perf, we used Valgrind's profiling tool Callgrind[7]. It introduces substantial overhead during profiling, because it uses extra instructions that record activity and keep counters for every function [22] but it can create very precise call-graph. With it, we checked that most of the major hotspots in OpenCV core library were part of `cv::dft` function used in all DFTs in KCF tracker.

In the following table, we present the major hotspots of OpenCV core library, ordered by CPU cycles:

Symbol	Cache misses (inclusive)	Cycles (inclusive)
`icv_y8_ownsrDftInv_Dir_32f`	0.0649%	18%
`icv_y8_ownsrDftFwd_Dir_32f`	0.568%	16.2%

Table 3.5: Profiling results for `libopencv_core.so.3.4.1.`.

From table 3.5, we can see that more than third of all cpu-cycles were spent during calculation of forward and inverse FFT in OpenCV core library. Because so much processing time was spend in these two functions, it might be the reason why perf was not able to create complete call graph for them.

Both of these functions are part of Intel Integrated Performance Primitives (IPP)[8] free-of-charge subset, nicknamed IPPICV. This subset is part of OpenCV (Open Computer Vision) from version 3.0. [3]. On Intel CPUs, OpenCV uses by default IPPICV for some functions, if it is available on the architecture. IPPICV's Fourier Transform algorithm depends on the magnitude of the input , if the input is of the power of 2 IPP uses Fast Fourier Transform (FFT), for other sizes DFT is used [13], as is the case here.

3.4 Summary of the results

If we add results from table 3.5 to results we obtained from `kcv_vot` binary for functions related to Fourier Transform, more than half of all CPU cycles are related to the calculation of DFT, and almost 30% of all cache misses. As such it was selected as the primary part

[7] <http://valgrind.org/docs/manual/cl-manual.html>
[8] <https://software.intel.com/en-us/intel-ipp>

of the program for optimization. Second spot for possible optimization is `ComplexMat_` template class. Primarily its methods for matrix operations.

Chapter 4

Parallel and extended implementations

In this chapter we describe our implemented parallel and extended versions of the tracker, with a description of the used libraries (e.g., FFTW and CuFFT) and changes made to KCF tracker to acquire as much performance as possible. The primary focus of parallel and extended versions was on Fourier transforms in the tracker, as mentioned in section 3.4. The source files for our modified KCF tracker can be found at: <https://github.com/Shanigen/kcf>.

4.1 Target platform

The target platform of development was NVIDIA Jetson Tegra X2[1]. It is NVIDIA's embedded System-on-Module (SoM), with focus on Computer vision and Artificial intelligence. It features dual-core NVIDIA Denver2 + quad-core ARM Cortex-A57, 8GB 128-bit LPDDR4 and integrated 256-core Pascal GPU.

The ARM CPU introduced problem during porting of the original implementation of KCF tracker. As described in section 3.3.2.1 functions for the computation of HOG, requires SSE2 SIMD instructions to run. Most of the modern Intel and AMD processors support SSE2. The ARM processors does not but has similar instructions set called NEON. For compatibility SSE2NEON[2] header file was used, which automatically converts SSE2 instructions to NEON instructions.

4.1.0.1 OpenCV4Tegra

The normal OpenCV library is optimized for x86 architecture. For Jetson TX2 we used OpenCV4Tegra, which is a free library provided by NVIDIA, which contains optimization for NVIDIA's Tegra CPUs and NVIDIA's GPUs. It is a closed-source binary replacement for the public OpenCV, so the programmer can write regular OpenCV code, which will automatically take advantage of OpenCV4Tegra's optimization. The performance gain over

[1] <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/>
[2] <https://github.com/jratcliff63367/sse2neon>

regular OpenCV is typically between 2x-5x on Tegra 3, Tegra 4 or Tegra K1 [19]. Between optimized functions, unfortunately, isn't `cv::dft`, which implements DFT.

For Jetson TX2, OpenCV4Tegra is part of NVIDIA's JetPack installer[3], used for flashing Jetson Developer Kits with the latest OS image. On our Jetson TX2, we had used Jetpack version 3.0, which contains OpenCV4Tegra 2.4.13.

4.2 Modifications

This section contains brief overview of all of the modification made to the original version of the KCF tracker. All of them are described in more detail in following sections.

First modification of the tracker is the creation of the `FFT` class which provides same interface for all of the different implementations of the Fourier transforms (i.e., Fastest Fourier Transform in The West (FFTW), CuFFT, OpenCV) available for the tracker.

For the CPU versions of the tracker we made modification to the `ComplexMat` template class to make it compatible with the FFTW plans. Also, we added OpenMP support for the tracker, which parallelizes the calculations of the response maps for individual scales.

For the Graphical Processing Unit (GPU) version of the tracker using the CuFFT library we added a GPU version of `ComplexMat` class and `KCF_Tracker::gaussian_correlation` to improve the performance. New mode for the tracker was also added, called the big batch mode, which tries to modify the workflow of the tracker to better suit GPU. This mode is, however, also compatible with FFTW version.

4.3 Fourier Transforms

This section is dedicated to detailed description of the different functions in the original version of the KCF tracker that performs the DFT. The reason for this description of the original implementation is to show the main problems with them.

4.3.1 KCF_Tracker::fft2

The first DFT that we will focus on is `KCF_Tracker::fft2` function, which takes a vector of feature channels as input and cosine window. It is used right after the extraction of the feature descriptors from the patch to transform them into the frequency domain, where they are then correlated with the filter. In this function on every feature channel matrix multiplication, with cosine window is performed, followed by the forward DFT. The result of the transform is then copied to an instance of `ComplexMat`, with `set_channel`. This whole workflow and the code of the function can be seen in figures 4.1 and 4.2.

OpenCV's implementation of DFT does not support transform over multiple data, so function call has to be made every iteration. Also if the input array is not the power of 2 the performance of the transform is decreased, though for arrays, whose size is a product of 2s,3s, and 5s (e.g., $60 = 5 * 3 * 2 * 2$) are also processed quite efficiently [25]. The horizontal and vertical dimensions of the window used for tracking in the KCF tracker, are calculated

[3] <https://developer.nvidia.com/embedded/jetpack>

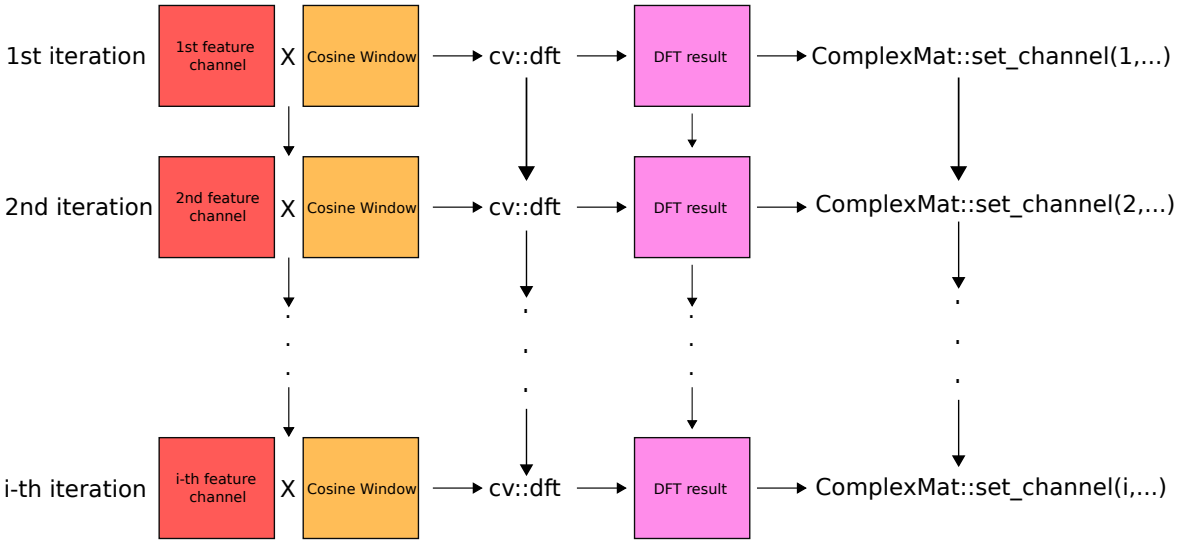

Figure 4.1: Diagram showing the workflow of `KCF_Tracker::fft2`. After `cv::dft`, `CompleMat`'s method `set_channel` is used to copy data from transform result, which is stored in `cv::Mat` to an instance of `ComplexMat`.

```
int n_channels = input.size();
ComplexMat result(input[0].rows, input[0].cols,
   n_channels);

for (int i = 0; i < n_channels; ++i) {
cv::Mat complex_result;
cv::dft(input[i].mul(m_window), complex_result, cv::
   DFT_COMPLEX_OUTPUT);
result.set_channel(i, complex_result);
}
return result;
```

Figure 4.2: Source code of original `KCF_Tracker::fft2` function.

by the following equation: $[X * (1 + \text{padding}) \div \text{cell size}] \div \text{cell size}$. The X represents either horizontal or vertical dimension size of the input bounding box; these dimensions will also be downscaled by the factor of 2 if the bounding box is larger than 100×100. Padding stands for additional padding added around the target and finally cell size, which is the size of the cell in HOG, as described in section 2.2.2.2. This equation takes into account only the size of the cell in HOG, so optimal size is not guaranteed. This was the case for all tested datasets, where the resulting dimensions of the window were never optimal.

For forward Fourier transforms `cv::DFT_COMPLEX_OUTPUT` flag is passed to `cv::dft`, to generates a full complex output, this allows more straightforward spectrum analysis, but sacrifices performance [25]. This flag, however, overrides the default option, which takes advantage of Complex-Conjugate Symmetry (CCS) that appears in complex output for real

data input, which allows packing the complex output to the array of the same size as the real input. As we discovered during the development of the FFTW version of the tracker the full complex output is completely unnecessary. The tracker with modifications to indexing in the `KCF_Tracker::gaussian_correlation` can work with the half complex output effectively halving the data sizes in the frequency domain and making the tracker and the DFTs faster as a result.

4.3.2 KCF_Tracker::ifft2

The second Fourier Transform, present in the original code is inverse DFT performed in `KCF_Tracker::ifft2`. There are two usages of this function in the tracker first is during the final step of obtaining the response map on the product of correlation filter and the result of Gaussian correlation. The second is in the `KCF_Tracker::gaussian_correlation` function when we transform the cross-correlated or autocorrelated feature channels back into the spatial domain. The input in the former is one channel, and in the later it is multichannel `ComplexMat`.

This function's workflow is similar to `KCF_Tracker::fft2` function, if the input `ComplexMat` has multiple channels. The difference here is that channels from input `ComplexMat` are first copied to a vector of `cv::Mats`, with the `to_cv_mat_vector` method. Following this Fourier Transform is performed for every channel, with results saved to another vector. After all channels are transformed back to the spatial domain, the vector of results is merged into single `cv::Mat` with `cv::merge` function. For one channel input `ComplexMat`, `to_cv_mat_vector` method is called first, to transform the input to `cv::Mat`. Following this single `cv::dft` call is performed and the result `cv::Mat` is returned. `cv::DFT_SCALE` flag is also passed to the `cv::dft` to scale the output of the inverse DFT, which is otherwise multiplied by the size of the array. The workflow diagram for multichannel input is in figure 4.3 and the source code of the function in figure 4.4.

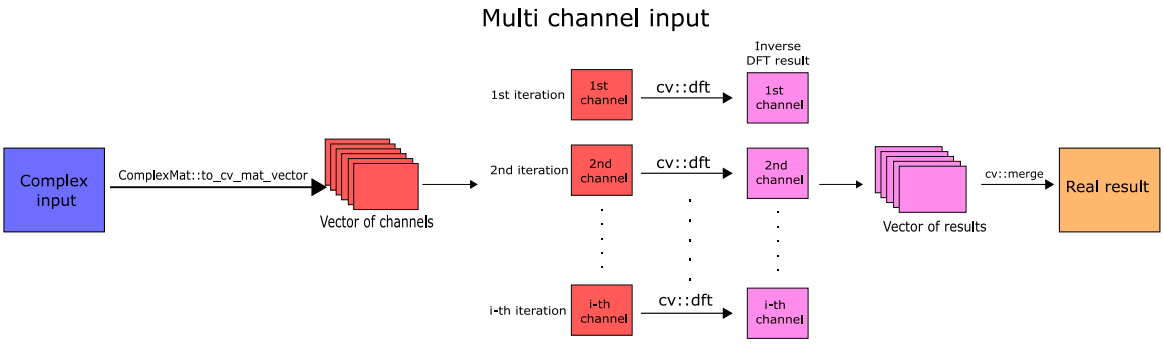

Figure 4.3: Diagram showing the workflow of `KCF_Tracker::ifft2` for multichannel input. We can see that after the input `ComplexMat` is transformed to a vector of `cv::Mats`, each representing one channel, the Fourier Transforms are performed the same way as in the `KCF_Tracker::fft2`.

Here to `cv::dft`, `cv::DFT_INVERSE`,`cv::DFT_REAL_OUTPUT` and `cv::DFT_SCALE` flags are passed. The `cv::DFT_INVERSE` flag changes the default behavior of `cv::dft` from forward Fourier Transform to inverse. The normal output of the inverse transformation is a

```
cv::Mat real_result;
if (input.n_channels == 1) {
cv::dft(input.to_cv_mat(), real_result, cv::
   DFT_INVERSE | cv::DFT_REAL_OUTPUT | cv::DFT_SCALE);
} else {
std::vector<cv::Mat> mat_channels =
   input.to_cv_mat_vector();
std::vector<cv::Mat> ifft_mats(input.n_channels);
for (int i = 0; i < input.n_channels; ++i) {
cv::dft(mat_channels[i], ifft_mats[i], cv::DFT_INVERSE
   | cv::DFT_REAL_OUTPUT | cv::DFT_SCALE);
}
cv::merge(ifft_mats, real_result);
}
return real_result;
```

Figure 4.4: Source code of original `KCF_Tracker::ifft2` function.

complex array of the same size as the input, but if `cv::DFT_REAL_OUTPUT` flag is passed to the function, it will assume that the input array has CCS and produce real output array. This again makes it unnecessary to use the full complex output in the forward DFTs.

4.3.3 Problems

Principal problems, with both of these workflows, are that when we tried to port them directly to GPU, it resulted in frequent data movement between GPU and CPU, which have a negative impact on the performance. This was evident with our first GPU version of the KCF tracker, which was performed with OpenCV's 'gpu' module, which allows modifying the original OpenCV's code with minimal changes. It internally uses calls to CUDA's runtime environment but shares the same API as normal CPU version OpenCV function. It required copying data from CPU to GPU each time we wanted to perform the transform, because both the CPU version and GPU version, which internally uses CuFFT, of OpenCV's DFT only support single transform execution. This resulted in a massive increase of the execution time added to this was also the problem that the data sizes for the transforms were too small to exploit GPU performance in the DFT properly. For these reasons not only the 'gpu' module version of the tracker was dropped, but the whole workflow had to be changed to better support the GPU. These changes were performed for both the FFTW and CuFFT implementations of the DFT.

4.4 FFT class

The first significant change to the tracker was the creation of a new class called `FFT`. This class provides virtual member functions, for all versions of Fourier Transform to have same function calls and also to improve readability of the code. Initially, we used only

C++ define directives in original functions (e.g., fft2) to decide, which part of the code will compile, this however quickly made the code very bloated and unreadable.

The new interface for Fourier Transform contains following functions:

- `void init(unsigned width, unsigned height, usigned num_of_feats, unsigned num_of_scales, bool big_batch_mode)`

  This method performs creation of the plans for FFTW and CuFFT version. For all versions, it also saves the dimensions of the transform. The `num_of_feats` stands for the number of features. When HOG, RGB channels and CN are all used, it equals to 44. The `num_of_scales` represents the size of the scale pool. Its default size is seven scales. Finally, the `big_batch_mode` is special mode created primarily for CuFFT version, but with works FFTW too, which performs response map calculation for all scales simultaneously. It is described in more detail in section 4.4.3. Depending on whether the boolean is true or not, additional plans will be created.

- `void set_window( cv::Mat &window)`

  `set_window` takes as an input result of `cosine_window_function`, which creates Cosine Window used in Fourier Transformation. Same as `init` it is called only once during initialization of the tracker.

- `ComplexMat forward(const cv::Mat &input)`

  Replaces `fft2(cv::Mat &input)` function, that performs FFT on the single channel input. For FFTW and CuFFT version it also adds support for the big batch mode.

- `ComplexMat forward_raw(float *input, bool all_scales)`

  Method used only in CuFFT version, It is intended to be used together with `cuda_gaussian_correlation`, which is wrapper function for custom CUDA kernel used for calculating the correlation of feature channels with a Gaussian kernel. It will be described in more detail in CuFFT section.

- `ComplexMat forward_window(const std::vector<cv::Mat> &input)`

  This method replaces the forward Fourier Transform `fft2` function described in section 4.3.1. It has support for the big batch mode in both FFTW and CuFFT versions, same as in `forward` and takes advantage of the advanced interfaces for both regular and big batch mode.

- `cv::Mat inverse(const ComplexMat &input)`

  Implements the inverse Fourier transform. In the FFTW and CuFFT version support for big batch mode is added, for current CuFFT version this method is not used and instead is replaced by `inverse_raw`.

- `float* inverse_raw(const ComplexMat &input)`

  Same as `forward_raw`, this is a special method used only with CuFFT version and is intended to be used together with `cuda_gaussian_correlation`.

In the following sections, we will describe the FFTW and CuFFT versions of the `FFT` class and how they are used to improve the performance of the Fourier Transform. The OpenCV version was only copied from the original version of the KCF tracker to this new interface.

4.4.1 FFTW

4.4.1.1 Overview

The FFTW library is a comprehensive collection of fast C routines for computing the discrete Fourier Transform and various individual cases thereof [21]. It supports DFT for both real or complex-valued arrays and for all arbitrary sizes and dimension of the input data it employs $O(n \log n)$ algorithms but works most efficiently if the size can be factored into small primes (2, 3, 5 and 7). The FFTW library comes in two variants, either double precision variant, for which all functions have `fftw` prefix or single precision variant with the `fftwf` prefix. In the KCF tracker, we used single precision FFTW variant, because rest of the tracker also uses only single precision arithmetics. In version 3.3.4, FFTW added support for the NEON instruction set, improving the speed on ARM CPUs. The FFTW also allows creating parallel plans for the shared-memory parallel hardware (e.g., multi-core CPU). These plans support both POSIX threads and OpenMP threads.

The FFTW uses plans to perform the DFT. The reason for this is that FFTW does not have fixed algorithm for computing the DFT, but instead it adapts the algorithm to the underlying hardware to maximize the performance. This approach splits the FFTW into two phases. In the first phase, the creation of the plans takes place, in which the planner finds the fastest way to execute the transform on the current hardware. This information is then stored in the data structure called plan. After the optimal plan is found, it can be executed as many as needed on the input array as dictated by the plan. This approach is reasoned by the authors [21]: *In typical high-performance applications, many transforms of the same size are computed and, consequently, a relatively expensive initialization of this sort is acceptable.* For the cases, where only single transformation is needed FFTW also allows using only heuristics, when creating the plan, lowering the cost of initialization.

4.4.1.2 Usage in KCF tracker

In our case the KCF tracker exactly carter the many transforms of the same size case. One problem that is however in the basic interface of the FFTW is that a plan initializes with input and output arrays on which the transform is to be performed. In KCF tracker the arrays on which the transforms are to be performed, are not available during the initialization phase of FFTW. This was a problem at first because it meant, we had to copy data to right arrays whenever we wanted to execute the corresponding plan.

FFTW thankfully has "new-array-execute" functions, which provides a way to perform the plan on a different array, than the one used during the creation of the plan. There are however few conditions, which have to be met to use a different array with the plan. The size of the new array has to be the same as the old one if the plan is out-of-place the two input arrays have to be different and for in-place, they have to be the same. Also, the authors state that the alignment of the input/output arrays have to be the same as the

original one, and recommends to use the `fftw_malloc`. Nonetheless, in the tracker, we use only ordinary `malloc` without experiencing any problem on multiple different platforms.

As mentioned in section 4.3 one of the problems, with OpenCV's algorithm for DFT, is that it can only perform a transform on a single array at one time. In the KCF tracker, this was primarily problem in `KCF_Tracker::fft2` and `KCF_Tracker::ifft2` functions. With advanced interface of FFTW, we were able to avoid this problem altogether. This interface allows performing a transform of multiple arrays simultaneously, resulting in better performance than if the FFTW was called for every array separately. During the initialization of the plan, the programmer has to specify the size of the transform, the number of transforms, rank and the stride to indicate, where to look for another transform. For more details, we refer to [21].

In KCF tracker we used the real data version for both the basic and advanced interface, which saves space and computation time. It uses the fact that the output of a DFT of real data possesses the "Hermitian" symmetry, which as a result makes half of the output redundant (being the complex conjugate of the other half). The output size of FFTW's two-dimensional real data transform of size $X \times Y$ is only $X \times Y/2 + 1$. This symmetry is also taken into account when performing the real data inverse DFT. We will now look more closely at how `Fftw::forward_window` and `Fftw::inverse` methods are implemented. The `Fftw::forward` method will not be described in more detail because it is implemented very similarly to `Fftw::forward_window`.

Fftw::forward_window

The whole method starts with the creation of a new instance of one channel `cv::Mat` of size $X \times (Y * \textit{number of channels})$, where X represents the width of single feature channel, Y height of single feature channel and number of channels equals the size of the input vector. The matrix is then filled in, with results of matrix multiplications of individual feature channels with cosine window. This matrix is used to align the input array for the FFTW's plan from the advanced interface. The `ComplexMat` class had to be modified to be compatible with the advanced interface. This modification will be discussed in more detail in 4.5 section. The only important thing that has to be known now is that `ComplexMat` has changed data storage from a vector of vectors to single vector, allowing it to be the output array of the FFTW plans and so no copying is needed as is the case in OpenCV version.

After the creation of both input and output arrays, the corresponding plan is executed. There are two plans used in `forward_window` both of which are part of the advanced interface. The first plan called `plan_fw` is used for computing DFT for feature channels of a single scale if the big batch mode is used it is executed during the initialization of the KCF tracker and after that during the update of the correlation filter. The second plan `plan_fw_all_scales` computes, as the name suggests, DFTs for all feature channels of all scales and is used only with the big batch mode. Otherwise, this plan is not created at all as mentioned in 4.4.

Fftw::inverse

Similar to the `Fftw::forward_window` this method also starts with the creation of

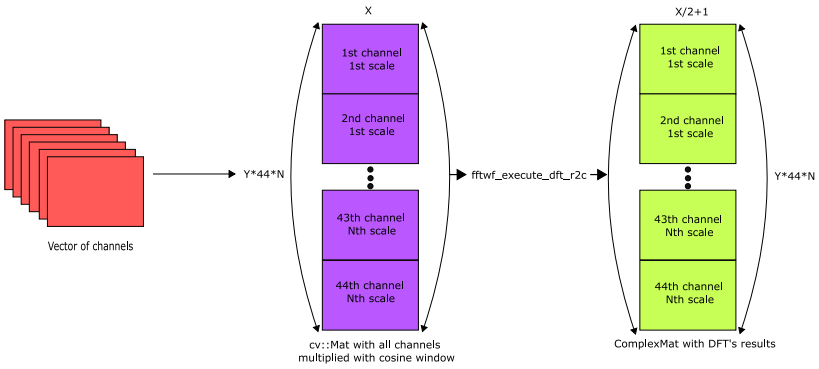

Figure 4.5: A diagram of the `Fftw::forward_window` method. In this figure we show the execution of the `plan_fw_all_scales` plan on N scales each consisting of 44 feature channels.

`cv::Mat`, however, here it is not one channel matrix but has the same number of channels as the `ComplexMat` from the input argument. After the creation of the output array, one of the four different FFTW's plans available executes, with two being available only in big batch mode.

- `plan_i_1ch`

  Used during the final step in calculating the response map for the scale, performs only single DFT on the result of matrix multiplication between correlated feature channels with the Gaussian kernel. This plan in the case of the big batch mode is not initialized at all and `plan_i_1ch_all_scales` takes its place instead.

- `plan_i_1ch_all_scales`

  The Big batch mode plan, same as the plan above, is executed during the final step of obtaining the response map. For this plan, the number of transform equals to the number of scales in the scale pool.

- `plan_i_features`

  Executed during the correlation of extracted features with Gaussian kernel in `KCF_Tracker::gaussian_correlation`. If the big batch mode is used it is used, during the initialization of the tracker and after the scale with the maximal response when updating the tracker's filter.

- `plan_i_features_all_scales`

  This plan is executed, when calculating the correlation of extracted features with Gaussian kernel with all scales. The number of transforms performed equals to the number of feature channels times number of scales in the scale pool.

After the inverse DFT is calculated the output has to be explicitly scaled back, because the FFTW's plans scale the output array by the size of the array on which the transform was performed.

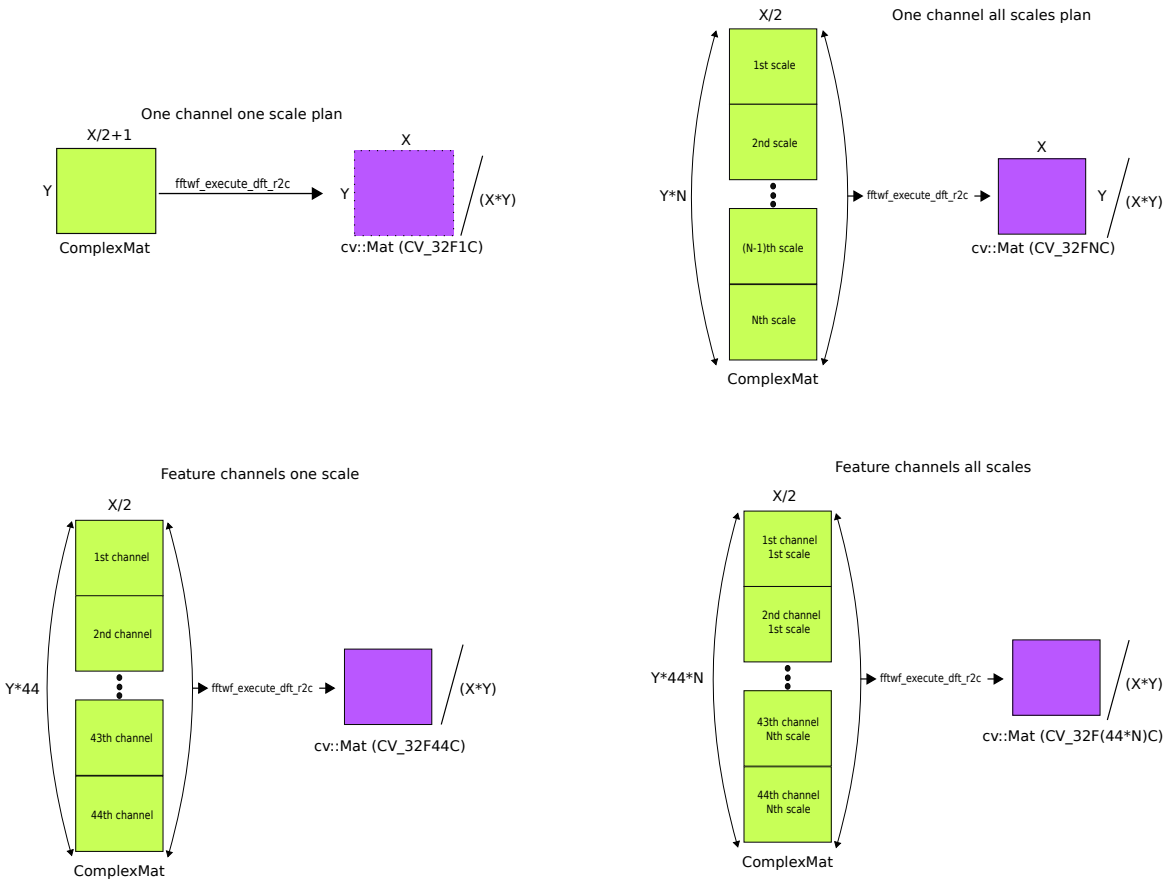

Figure 4.6: A diagram, showing workflows for all plans in `Fftw::inverse`. The `cv::Mat` stores each channel pixel by pixel after other channels unlike `ComplexMat`, which stores each channels matrix by matrix after other channels.

4.4.1.3 CuFFTW

With the FFTW version, we also tested the NVIDIA's CuFFTW library. This library provides FFTW interface for the CuFFT library, allowing the programmer to use NVIDIA's GPU in FFTW with minimal modifications to program source code [23]. There are three steps required before the interface can be used. First, the include file for FFTW (`fftw3.h`) should be replaced with CuFFT include file (`cufftw.h`). The second modification is during linking, where instead of linking with double/single precision libraries such as `fftw3/fftw3f`, the program links with both the CuFFT and CuFFTW libraries [23]. The last requirement is that the search path contains the directory, where `cuda_runtime_api.h` is located. The last step is done because the CuFFTW library requires data to be on GPU, but also allows the user to use data located on the CPU, which it will copy to the device memory automatically using the CUDA runtime library. In the case of the FFTW plan from the advanced interface, that maps to `cufftPlanMany` with CuFFTW, data are assumed to be in CPU memory.

The modifications to the FFTW version were indeed minimal. However, the main problem with CuFFTW library is that the programmer cannot explicitly manage data movement

between device and host memory. Also, the CuFFT library even though based on the FFTW library works slightly different, which results in a number of unnecessary operations (e.g., input and output arrays during initialization of plans). Both of these reasons is why we moved from exploring this option in more detail and decided to use native CuFFT library instead.

4.4.2 CuFFT

4.4.2.1 Overview

The NVIDIA's CuFFT library is a high-performance library for computing the FFT on NVIDIA GPUs. Its API is modeled after the FFTW library, and same as FFTW is also based around plans, which use internal building blocks to optimize the transform for the given configuration and the particular GPU hardware selected [23]. The main difference between the FFTW and CuFFT is the way how they work with plans. The FFTW library has many different types of plans, but only one execute function for all of them. The CuFFT, on the other hand, has few plans and many execute functions, which determines the precision and type of the input. In KCF tracker we used only the single precision execute functions for the same reason as with FFTW[4]. The rest of the library is used the same way as the FFTW. However, input and output arrays have to be located in the device memory.

The CuFFT library also supports the use of CUDA streams, which allows CUDA operation in different streams to run concurrently improving the performance. The main advantage with the use of the CUDA streams is that we can overlap kernel executions with the data copying. However, the CuFFT does not allow to execute its kernel in different streams until the previous plan was completed. This limitation is not a problem if the datasets are large enough that the copying overlaps the full execution of the plan. This was regrettably not the case in KCF tracker, where data sizes are too small and in our testing of the CUDA streams with CuFFT, there were no performance gains, on the contrary, the performance lowered because of the additional CUDA API calls. These testing results are the reason why the use of streams with CuFFT was dropped.

4.4.2.2 Usage in KCF tracker

The CuFFT version of the KCF tracker is not only modification of the Fourier Transforms, but of all operations in the frequency domain. A significant part of this modification is the addition of a GPU version of `ComplexMat`, which will be discussed in the following section. This together with a GPU version of `KCF_tracker::gaussian correlation` function and means that all operations in the frequency domain are performed on GPU. We also took advantage of the Jetson TX2's shared memory between CPU and GPU to minimize the number of data movements between host and device memory. We will describe here the whole process of the CuFFT version when it is used with the GPU version of `KCF_tracker::gaussian correlation`. Normal `KCF_tracker::gaussian correlation` can also be used, but in this case, the work-flow is almost the same as the FFTW version, except for the data copying between the host and the device, so we will not describe it in more detail here. Also,

[4]KCF tracker performs all of its operations in single precision arithmetics.

all used CuFFT plans work the same way as the ones described in FFTW version, so they will not be described in detail too. Currently, the CuFFT version only supports Gaussian kernel for filter learning and tracking.

The central GPU part of the tracker starts after the extraction of all feature descriptors into a vector when the `cuFFT::forward_window` method is called. The `cv::Mat` used to store the whole feature vector, uses NVIDIA's Zero Copy memory[5]. It allows to entirely avoid memory copying in the case of Jetson TX2, by allowing GPU threads to access CPU memory. The rest of the method work-flow is the same as in the case of `cuFFT::forward_window`.

Following the call to `KCF_tracker::gaussian correlation` function, and calculation of a complex conjugate for extracted features and their element-wise multiplication with interpolation model in it. Call to `cuFFT::inverse_raw` method is performed. As already mentioned in 4.4, the change here, over regular method, is that return value here is array allocated in device memory, that is passed to the `cuda_gaussian_correlation` function. In it is located custom CUDA kernel, that performs rest of the operations in `KCF_tracker::gaussian correlation`, and stores the result in an array that is passed as one of the input arguments. The array for the results is allocated in the device memory during the initialization of the tracker and is stored in global pointer `gauss_corr_res`. The next step is the `cuFFT::forward_raw` function, which unlike `cuFFT::forward` takes an array and boolean as input parameters. The boolean is set to true depending on whether all scales are currently worked on or not.

During the final step in calculating the response map in `cuFFT::inverse`. The output `cv::Mat` uses as data array Zero Copy memory. This, however, introduced problem because all calls to CUDA kernels are asynchronous so we have no guarantee that the `cv::Mat` will contain required data when we try to perform scaling on the DFT result in CPU. This problem does not exist in `cuFFT::inverse_raw` because the scaling is part of the CUDA kernel in `cuda_gaussian_correlation`. For this reason after execution of the CuFFT plan `cudaDeviceSynchronize` is called to block the CPU until all CUDA kernels finish. We tried to put as many operations on the GPU as possible to avoid overusing the synchronization call, so the next possible step is to perform the search for the maximal response also on the GPU.

4.4.3 Big batch mode

The big batch mode is an alternative to the regular mode in KCF tracker; it was created primarily for the CuFFT version of the tracker but also supports FFTW version. This mode is intended to be used with scale pool of size bigger than one.

The regular mode of KCF tracker performs calculations of the response map for each scale separately. Unfortunately, the size of the data for each scale is too small to take advantage of GPU. The big batch mode tries to address this problem, by calculating all scales together as one big data set. The default scale pool as mentioned in 2.2.2.2 contains seven scales, which results in seven times larger datasets for all operations during calculation of the response maps. There are however two parts during this mode that are performed separately for each scale. The first one is the extraction of the feature descriptors from the

[5] <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#zero-copy>

scaled patches and the second one is the searching of the maximal response. This mode also takes advantage of FFTW's and CuFFT's advanced interfaces, which allows execution of multiple DFTs in the same time.

4.5 ComplexMat

There are two versions of `ComplexMat`, where one runs only on CPU and other runs only on GPU. The CPU version of the class is like original template class, but the GPU version is a regular class with the same name, using float precision arithmetics. Both versions share the same interface. However, data in GPU version are stored in only the device memory, so it is currently only compatible with the CuFFT version of the tracker.

4.5.1 CPU version

The first problem with the original implementation of `ComplexMat` class was its data organization. It uses a vector of vectors of complex numbers to store data, where each vector represented one channel. This data organization, however, was not compatible with FFTW or CuFFTW. The new data organization is a single vector of complex numbers. Allowing the `ComplexMat` to be passed as an output of FFTW and CuFFT plans.

Another change made, was an added support for big batch mode, allowing the `ComplexMat` to store data from all scales. Following list contains all of the modified and the newly added methods and class functions. Functions and methods that differ only with indexing, because of the different storing of the data, are not listed:

- `void create(int _rows, int _cols, int _n_channels)`

  This method was added together with its counterpart function `void create(int _rows, int _cols, int _n_channels, int _n_scales)` to add support for the big batch mode in FFTW version. Both methods perform allocation of space for the future data. The first method is used for regular mode. The second one is used during the big batch mode, it adds the fourth parameter, which represents the number of scales overwriting the default value (i.e., one) representing the number of scales in `ComplexMat`.

- `void set_channel(int idx, const cv::Mat & mat)`

  Used only in OpenCV version of the tracker; it converts data from `cv::Mat` on the input to the data vector of `ComplexMat`. The original version called class function `convert`, that performed the conversion of the `cv::Mat`'s data to vector and this resulting vector, representing one channel, was then stored in the primary data vector. Because we changed the way data are stored, the call to the `convert` function was unnecessary, and the conversion now happens in the `set_channel` method itself.

- `void sqr_norm(T *sums_sqr_norms)const`

  A method used for calculating the sum of squared complex norms of all channels. The original method did not have input parameters and returned value, which represented the sum of the complex squared norms. Current version takes an array as

an input that is also used to store the result. This change was done to support big batch mode, for which the size of the input array equals to the number of scales in the scale pool. Before the computation, the number of channels per scales is computed to perform the summation for each scale only on the corresponding channels. After the sum is computed, it is stored in the input array on the index representing a position of the current scale in the scale pool.

- `ComplexMat_<T> sum_over_channels()const`

  Used only if the linear kernel is selected for filter learning and tracking. This method was also modified to support big batch mode. It returns an instance of `ComplexMat`, with the number of channels equal to the number of scales of the caller, where each channel is the sum of all channels of that scale.

- `std::complex<T>* get_p_data()const`

  Return pointer to the beginning of data stored in the data vector. This method was created for compatibility reasons with FFTW, allowing an instance of `ComplexMat` to be passed as the destination of the FFTW plan.

- `ComplexMat_<T> mul2(const ComplexMat_<T> & rhs)const`

  Method created to support big batch mode, based on the `mul` method in the `ComplexMat`. It calls class function `matn_mat2_operator`, which implements operators overloading for matrix operations between multichannel and single channel instances of `ComplexMat`. This method is used only when the linear kernel is selected.

4.5.2 GPU version

The GPU version of `ComplexMat` was created because CuFFT plans execute asynchronously. This asynchronous execution resulted in the need to use synchronization between device and host after every execution of the plan. By moving all operations in the Frequency domain to the GPU, we now perform the synchronization only at the end of `cuFFT::inverse`, when obtaining the response maps. The GPU version of the `ComplexMat` implements all functions as custom CUDA kernels, additionally move assignment operator was implemented to avoid unnecessary copying of data in the device memory. One limitation in the current GPU version of the `ComplexMat` is that the kernels can work only on channels, whose product of dimensions is smaller than 1024. That is the maximal number of threads that can be in one block on Jetson TX2. This limit could be possibly avoided if we used different indexing in kernels. However, we were unable to come up with a solution that would not sacrifice too much performance. For `sqr_norm` method, the indexing is especially tricky, because we use parallel reduction with sequential addressing[6] to minimize the calls to atomic addition, which is required when performing parallel summation.

[6] <http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf>

4.5.2.1 CUDA error check

With the GPU version of the `ComplexMat`, we also added debugging in-line functions for CUDA and CuFFT. There are three functions available:

- `CudaSafeCall( err )`

  Used to check errors that are returned from CUDA API calls.

- `CudaCheckError()`

  Used to check errors from CUDA kernel calls. `cudaDeviceSynchronize()` is called every time it is used to make sure that all kernels finish, which in case that some error happens in any of the kernels will return it.

- `CufftErrorCheck(call)`

  Used to check errors from CuFFT API calls.

These functions are not used in default and are only available if the `CUDA_DEBUG` variable is set during the compilation.

4.6 CPU parallel options

In this section, we describe the available CPU multi-threaded options of the KCF tracker. There two primary options either C++ async directive or OpenMP. The original version of the KCF tracker has a multi-threaded option, which uses the C++ async directive to perform the computation of the response maps for all scales in parallel.

In the regular mode, OpenMP option does the same parallelization as the async option but also parallelizes the search of the maximal response in the response maps obtained for each scale. The OpenMP has alternative usage when used with the big batch mode here it parallelizes the extraction of feature descriptors and the search of the maximal response from the response maps of the scales. For big batch mode, the OpenMP can be used with all of the available versions of the tracker. Additionally, with the FFTW version, it is also used to create multi-threaded plans, where the number of threads used with each plan is obtained from `omp_get_max_threads()`, which returns the maximum number of threads that can be used in the parallel region. If neither the async directive or OpenMP is used, the FFTW versions will use two threads as default for its plans. The parallelization of FFTW plans is however not guaranteed, because we pass `FFTW_PATIENT` flag during the initialization. This flag not only finds the fastest plan possible, in exchange for the time it takes to find the plan but in the case that the problem size is not big enough to take advantage of parallelization, it will only use one thread.

4.7 HERCULES compiler

As mentioned in section 2.1.3.3 the HERCULES compiler is currently only able to perform the PREM transformation on loops specified with `#pragma omp target teams distribute parallel for`. The best place of the KCF tracker that can use this pragma are the `ComplexMat` methods for overloading operators, e.g., `ComplexMat::mat_mat_operator`.

As discussed in section 2.1.3 allows offloading parts of the source code to GPU with timing guarantees. For our testing, we, however, had only available the CPU version of the compiler that performs the transformation of the source code but for the CPU. We used HERCULES compiler version 2017.11.3. In this version, the CPU implementation has no proper driver and instead uses a script that takes C/C++ file to be compiled as an argument. This proved to be a problem because this script is not compatible with CMake build system used in our KCF tracker and also requires that the source file contain the `main` function. This forced us to concatenate all source files from the KCF tracker into one.

When we tested the compilation of this new source file with the regular Clang version available from Ubuntu package repositories no errors were encountered, however, during the HERCULES's LLVM passes the compilation encountered errors immediately with the first pass, which we could not resolve. We contacted the developers of the compiler and discussed possible solutions. Here we were informed that if we would like to use test the compiler with the CPU version of the `ComplexMat` we would have to change it into non-template class. Next problem was also that the current version requires that every variable transformed for predictability must be defined either as a static global variable or in the entry function (main). However, making the variables global or static in `ComplexMat` would be quite difficult for at least three reasons:

- The size of the matrix depends on the size of the tracked object and is different for each video. It could be possible to make the size constant, but we will lose generality.
- The same class is used for "images" with a different number of channels and therefore different sizes of data (all used in one tracking step). We could work around this by having different types for different image depths, but this would probably lead to unnecessary duplication of code.
- The tracker can run some computations in parallel threads, and each thread needs to have its own matrices to work on. At compile time, we do not (want to) know how many CPUs are there at runtime and how many copies of matrices will be needed.

We were also informed that there the current problems with the compilation also seem to be related to the C++ class allocators/destructors in classes. Because of these problems, we were unable to test out the HERCULES compiler with the KCF tracker and show its performance in comparison to other versions of the tracker.

Chapter 5

Results

5.1 Benchmark results

In this section, we present benchmark results for all available versions of the KCF tracker. We tested the benchmarks on six different datasets from VOT Challenge 2016[1], which are shown in figure 5.1. We tried to select datasets with different sizes of the object of interest to better capture differences in execution time on GPU and CPU. However, we were limited by the GPU version of `ComplexMat`, whose CUDA kernels are limited to dimensions of feature channels with product smaller than 1024. The reason for this limit is described in section 4.5.2.

Each dataset was first run with default scaling, which resizes the frames and patch by a factor of two only if the patch from initial bounding box is larger than 100×100. The second run was performed with alternative scaling, which scales bounding box and input frames by a factor, that results in a window of size 128×128. This size of window produces feature channels of dimensions 32×32, with default parameter of HOG, i.e., `cell_size` $= 4$. With this sizes of feature channel, all of the different libraries used for calculating DFT can use the most optimal algorithm, i.e., a power of two FFT. The third run was again with alternative scaling, that scaled the window to dimensions of 256×120, which results in feature channels of dimensions 64×30. That is the maximal size of feature channels that CuFFT versions can work on. Also, both of these dimensions are divisible into small primes (i.e., $64 = 2^6$ and $30 = 5 * 3 * 2$) on which still quite efficient algorithms for calculating the DFT can be used.

Figures from 5.2 to 5.8 with the results of the benchmarks start from smallest data sizes to biggest data sizes. The datasets selected for alternative scaling were those, whose window size in the default scaling was closest to the desired size (i.e., 128×128 and 256×120). During our testing on Jetson TX2, we also turned off the Ubuntu LighDM so that the GPU would not be interrupted by it and influence the results. We used boxplots to show the time for one frame, where the box is extending from the lower to upper quartile with orange line demarcating the median. The whiskers show the lowest value of the 1.5 IQR of the lower quartile and the highest value of the 1.5 IQR of the upper quartile.

[1] <http://www.votchallenge.net/vot2016/dataset.html>

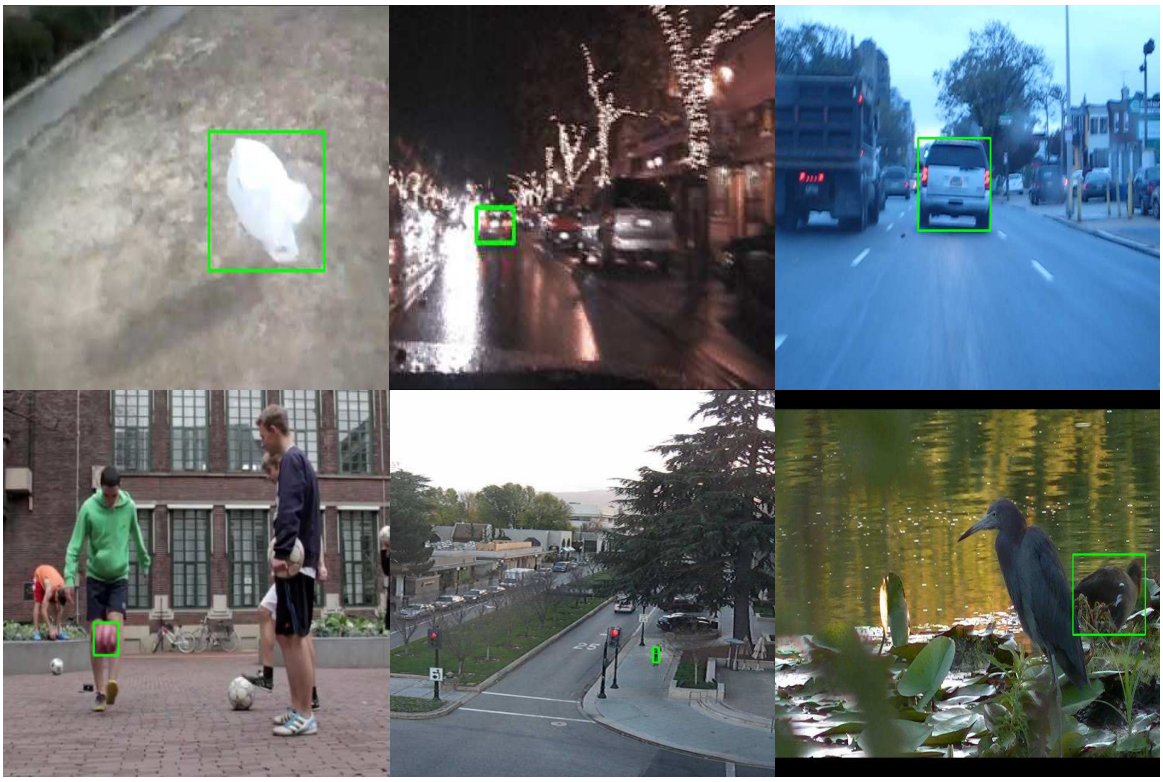

Figure 5.1: Datasets used for benchmarking, starting from the top left: Bag, Car2, Car1, Ball1, Pedestrian2, Nature. All screenshots of the datasets were resized to 128×128 so they fit inside one figure.

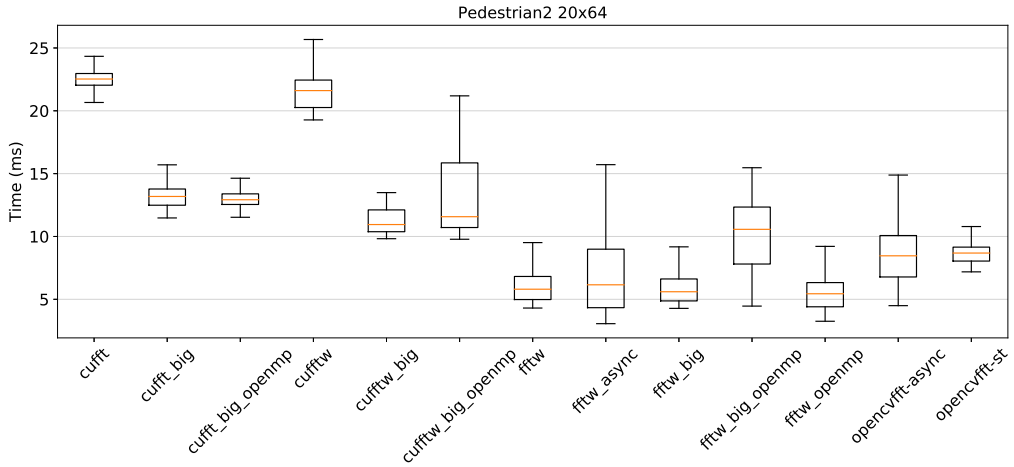

Figure 5.2: Pedestrian2 dataset with default scaling. This dataset was not resized, because the initial bounding box was smaller than 100×100.

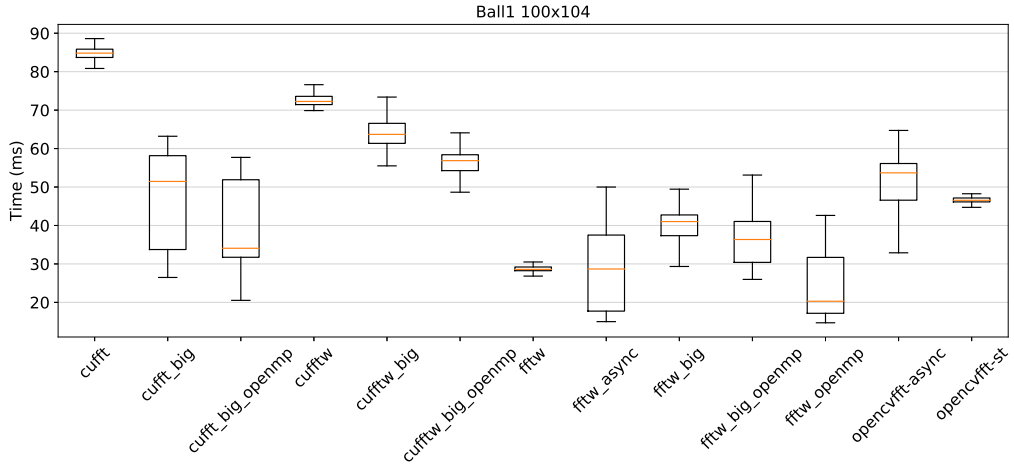

Figure 5.3: Ball1 dataset with default scaling. This dataset was not resized, because the initial bounding box was smaller than 100×100.

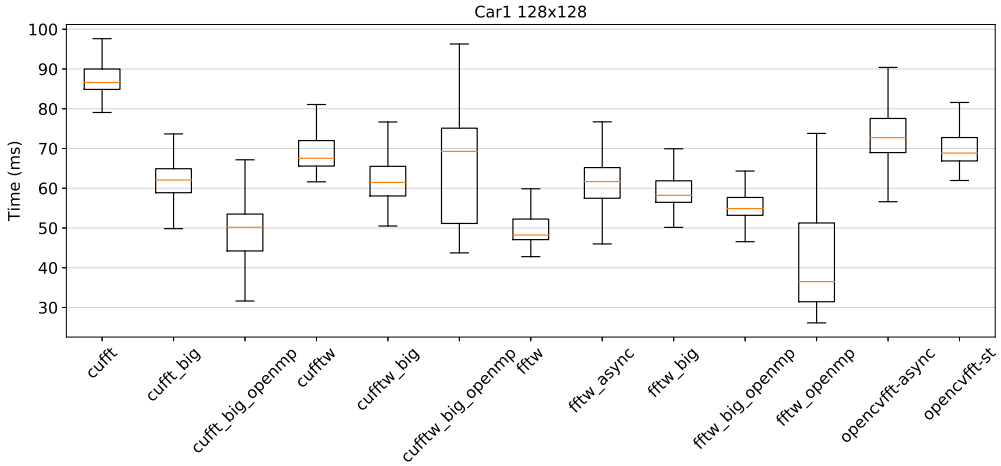

Figure 5.4: Car1 dataset with alternative scaling to power of two feature channels.

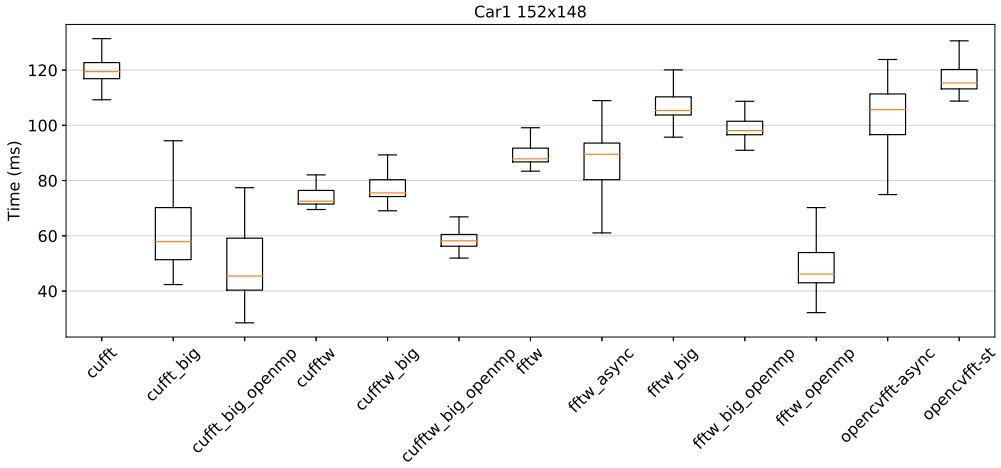

Figure 5.5: Car1 dataset with default scaling to power of two feature channels.

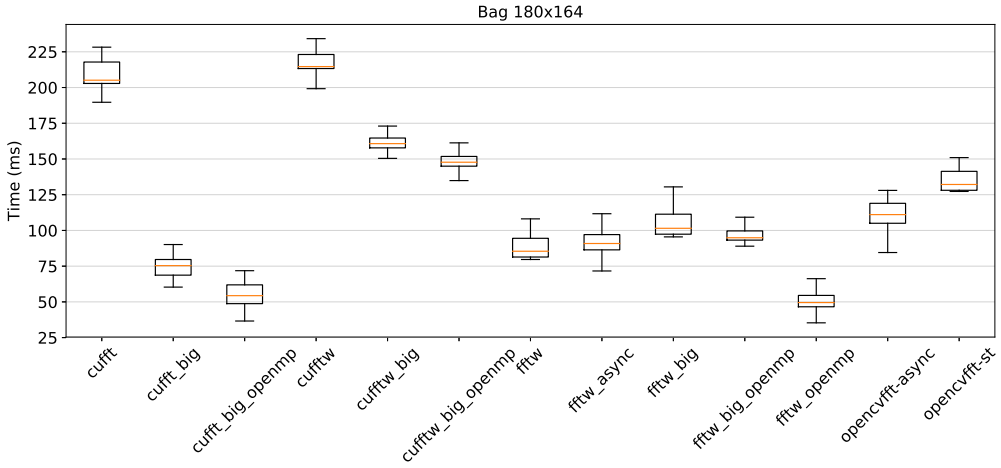

Figure 5.6: Bag dataset with default scaling.

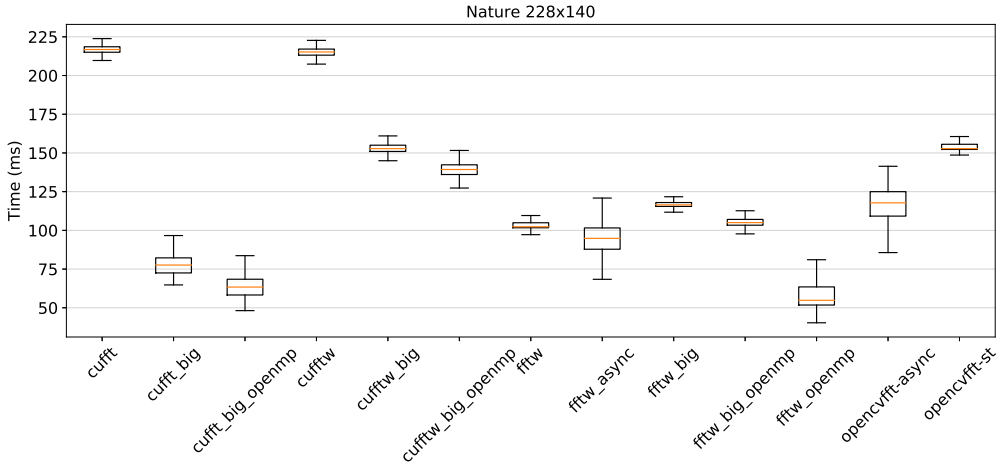

Figure 5.7: Nature dataset with default scaling.

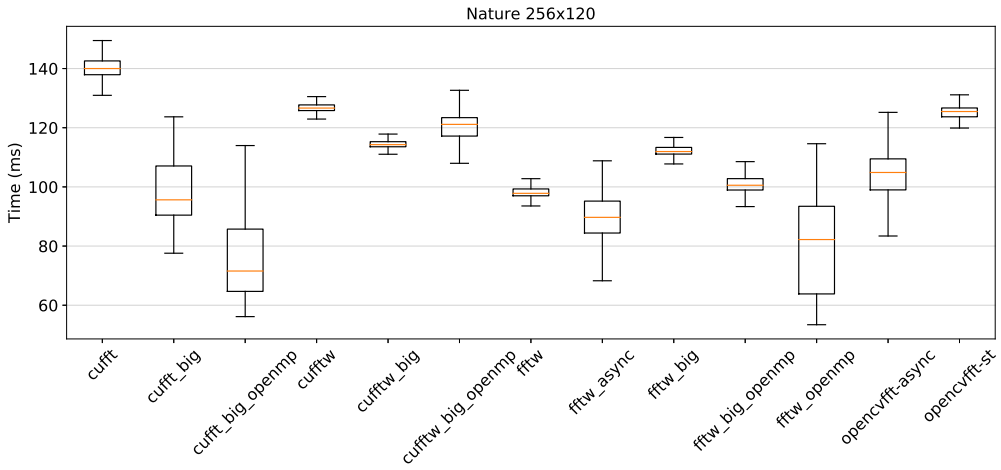

Figure 5.8: Nature dataset with alternative scaling to maximum allowed size of feature channel for CuFFT versions.

5.1.1 Interpretation of the results

Starting from the smallest data sizes, we can see that all of the CPU version of the KCF tracker, perform much better than GPU version, which is logical as the data sizes are too small to benefit from the GPU. However, we can see that how as the data sizes increase the performance of the GPU versions starts to become more similar to the CPU version, especially the CuFFT version with big batch mode and OpenMP starts to outperform CPU versions of the tracker and also perform similarly to the FFTW version with OpenMP. The reason for this is that the overhead introduced with the use of the GPU and CUDA API calls starts to become much less evident because of the performance gains in the operations (e.g., DFT, Gaussian correlation) and efficiency with which the GPU can work on bigger data sizes. These performance gains with bigger data sizes can be best seen when comparing `cufft_big_openmp` with `fftw_big_openmp`.

The most significant problem with the CuFFT and CuFFTW versions is with the execution time of the CUDA API calls and CUDA kernels. Especially significant fluctuation in the execution time was seen in CuFFT kernels, where when tested on Bag dataset for one frame they took only 16 ms and in next jumped to 30 ms, even though the size of the data does not change. Big variance resulting from this fluctuations can be best seen in figures 5.3 and 5.7. Unfortunately, we were unable to find the reason for this fluctuation. For the custom CUDA kernels that we implemented this problem was not seen.

In figure 5.2 we can see that the CuFFTW versions using the big batch mode outperform the CuFFT versions with the big batch mode. The reason for this result is that the CuFFTW versions, unlike CuFFT, performs the operations in the frequency domain on the CPU, which gives it better performance for smaller data sizes, even though it performs memory copying between host and device and vice versa. In the following figure 5.3, where the size of data increases the performance of CuFFTW versions is worse than the CuFFT because the overhead introduced by the data movements starts to be more evident.

Both the advanced interface in the FFTW library and the CuFFT batch executions, introduced performance gains that we can see when compared with the original OpenCV version on bigger data sizes 5.7. When compared only with only the FFTW this difference can be seen in all tested data sizes.

5.2 CuFFT version profiling

In this section, we present the results of the NVIDIA's profiling tool nvprof[2] for the CuFFT version with and without the big batch mode and also the CuFFTW version with the big batch mode, to show the differences in the size of the kernel computations and data movements between host and device. We used the bag dataset with default scaling for the profiling. The images are taken from the NVIDIA Visual Profiler, which allows visualizing the results of the nvprof. The resolution used in the visual profiler for all versions is the same to better capture the difference.

In the figures 5.9 and 5.10, we can see how all calculation for all scales are performed together in the big batch mode, wherein the normal mode each scale is performed individually. This results in seven times more calls to CUDA runtime API and kernels, which

[2] <https://docs.nvidia.com/cuda/profiler-users-guide/index.html>

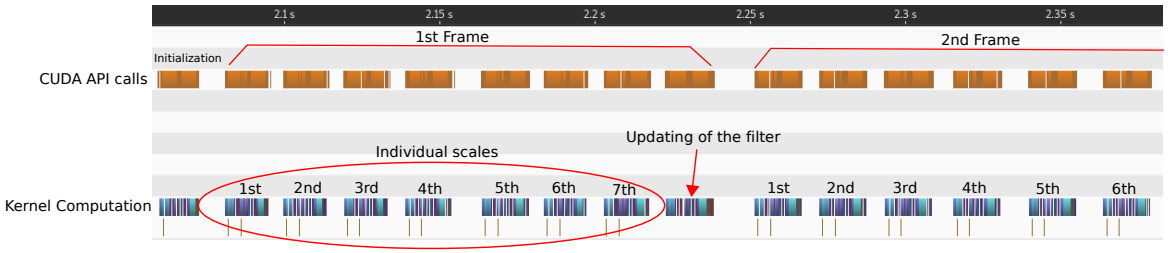

Figure 5.9: CuFFT version without big batch mode.

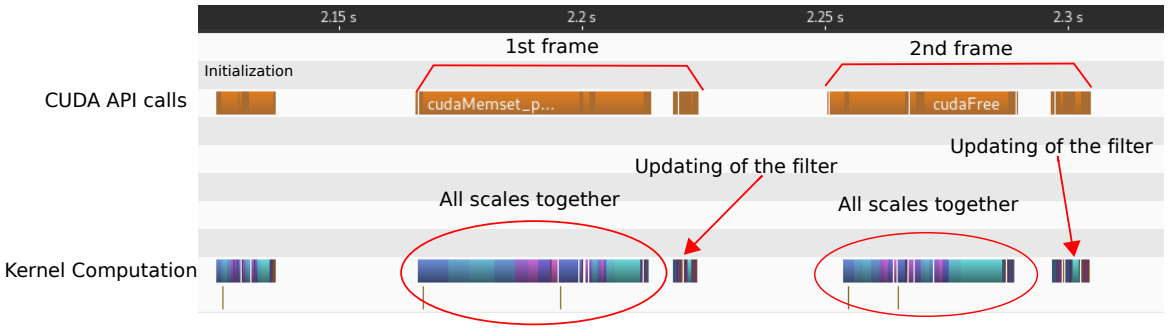

Figure 5.10: CuFFT version with big batch mode.

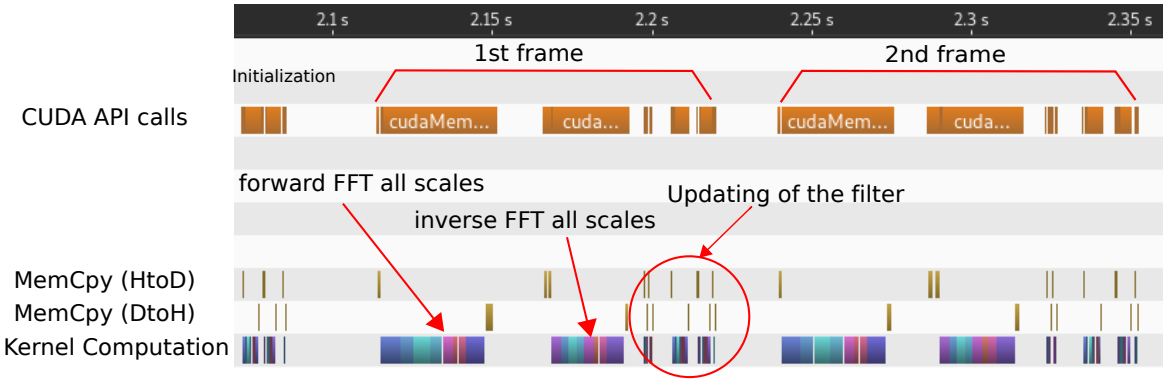

Figure 5.11: CuFFTW version with big batch mode.

have a high impact on performance and predictability as mentioned resulting in significant execution time increase. Because the big batch mode takes all the scales as one big array, it can offset the small data sizes with which the KCF tracker works, making the data more suitable to be executed on the GPU. As demonstrated in section 5.1 this mode significantly improves the performance of CuFFT version, especially with bigger data sizes.

The figure 5.11 shows how in the CuFFTW version there is a considerable gap between the forward and inverse FFT in the kernel computations, which represents the operation in the correlation with the Gaussian kernel in the Fourier domain. The second problem is the number of memory copies from the host to device memory and vice versa, which on Jetson TX2, as mentioned in section 4.4.2.2, can be avoided entirely. The reason for these memory copies is that the CuFFTW batch mode assumes the input and output arrays to

be in the host memory and we used the regular CPU version of the `ComplexMat` version with the CuFFTW. From the figure 5.10, it is apparent that both of the problems are not present in the CuFFT version.

5.3 FFTW version profiling

The following section contains the profiling results for the FFTW version of the KCF tracker. Specifically, the single threaded `fftw` version, to better see the results. We profiled this version in the same way as we profiled the original version of the tracker, using the Linux perf tool with profiled events being the CPU cycles and cache misses. The profiling was also done on the same hardware so that the testing conditions are not different. The overall number of events recorded is the same as in the profiling results from the section 3.4.

Symbol	Cache misses (inclusive)	Cycles (inclusive)
`KCF_Tracker::track`	76.7%	77.3 %
├─ `KCF_Tracker::get_features`	├─ 38.1%	├─ 19.9 %
├─ `KCF_Tracker::gaussian_correlation`	├─ 23.5%	├─ 34.6 %
└─ `Fftw::forward_window`	└─ 6.81%	└─ 19.5 %

Table 5.1: Profiling results for `KCF_Tracker::track`. The threshold was set to 5% of total cache misses measured.

As we can see from the table above one big difference happened in the number of cache misses. In the original version of the tracker, the function with the most cache misses in the `KCF_Tracker::track` was `KCF_Tracker::gaussian_correlation`, with 42.6%. In the table 5.1, we can, however, see that in the FFTW version the cache misses dropped almost by half. In its place is now `KCF_Tracker::get_features`, where the number of cache misses increased from 26.5 % to 38.1%. We will look in detail only to `KCF_Tracker::gaussian_correlation` and `Fftw::forward_window`, which contains modifications that we made. `KCF_Tracker::get_features` was not modified, so its profiling results are the same as in 3.4.

5.3.1 Gaussian correlation

The inverse DFT is still the primary hot-spot here but big change happened with the results for the `ComplexMat` by changing the data layout from vector of vectors to only single vector, we were able to reduce the number of cache misses happening in the class's methods all across the board. The number of cache misses for `ComplexMat_<float>::mat_mat_operator` dropped from 6.35 % to only 0.0917 %. The `ComplexMat_<float>::conj` was not present in the results instead the `ComplexMat_<float>::sqr_norm` was. We concluded that the reason for this are inaccuracies of perf described in section 3.3.1, both of these operation operations are close to each other, which could make the results for both operations show only in one of them because of this we cannot say how the cache misses changed. For the `Fftw::inverse` the main two hot-spots are the execution of the FFTW

Symbol	Cache misses (inclusive)	Cycles (inclusive)
`KCF_Tracker::gaussian_correlation`	23.5%	34.6 %
├─ `Fftw::inverse`	├─ 10.1%	├─ 19.3 %
└─ `ComplexMat_<float>::sqr_norm`	└─ 4%	└─ 0.881 %

Table 5.2: Profiling results for `KCF_Tracker::gaussian_correlation`. The threshold was set to 4% of total cache misses measured. Rest of the operation was under 1 %.

plan and the `cv::Mat::convertTo` function, which even with Callgrind we were unable to locate its caller. Our educated guess would be that it is the part of the creation of the output `cv::Mat`. For the inverse DFT, the overall number of cache misses is now reduced to only half it was, when using the OpenCV library. The main reason for this is that both the `ComplexMat` and `cv::Mat` data arrays can be passed as input and output to the FFTW plan.

5.3.2 Forward Fast Fourier Transform

For `Fftw::forward_window` the results show that the overall number of cache misses is bigger than in the original version. Increasing from 5.37 % to 6.81 %. In the transform itself under 1 % of all cache misses were measured, which is less than the original function with 2.52 % and as such is no longer the central hot-spot here. Instead, the new problematic section is the creation of the one channel `cv::Mat` used as an input for the FFTW plan. The cache misses from the matrix multiplication between feature channels and cosine window and iterations over the input `std::vector` with the channels now equal to 5.66 %. Reason for this is that in each iteration we shift between memory location of the `cv::Mat` and the input vector but their sizes are too large for both of them to be stored in the same time in the cache resulting in the repeated eviction of one by the other.

Chapter 6

Conclusion

Based on the profiling we made to the original version of the KCF tracker, we modified the most demanding sections of the tracker. These modifications resulted in 13 different versions of the tracker using a different combination of technologies (OpenMP, CUDA), libraries (OpenCV, FFTW, CuFFT) and data structures. We then benchmarked all of these versions of the tracker to compare their performance and predictability. The results showed that the acceleration of the tracker with GPU is advantageous only for larger tracking windows and if we limit the number of data transfers between CPU and GPU.

During the implementation of the CuFFT version, we also found out that to obtain better performance on the GPU it was not viable to convert only parts of the tracker, because of the asynchronous execution of the CUDA kernels but to convert most of the tracker to execute on the GPU. In all of the current GPU versions, we perform explicit synchronization between the host thread and the device which negatively impacts the overall performance of the tracker. For this reason, we would like to test out full GPU version of the tracker. Also, we noticed a noticeable slowdown with larger windows which could be accounted for the extraction of features and is one of the possible places that could show improvement if transformed to execute on GPU, especially the HOG, which is well suited for GPU. Another improvement that we would like to make is to change the current CUDA kernels in the GPU `ComplexMat`, so they work on all sizes of feature channels and also make the GPU `ComplexMat` compatible with linear kernel and not only Gaussian kernel.

As for the PREM model instead of the manual modification of the code we decided to use the prototype compiler from HERCULES project, which should make these modifications automatically. However, we discovered that the current version of the compiler could not be easily used for complex codes such as the KCF tracker. In the future, we would like to perform a manual implementation of the PREM model on the KCF tracker and see how the performance and predictability changes compared to currently available versions. We could see in the benchmarking results that the execution time for some datasets (e.g., Car1, Nature) showed large differences in the maximum and minimum time per frame.

Bibliography

[1] Emiliano Betti. *Satisfying hard real-time constraints using COTS components*. PhD thesis, UNIVERSITÁ DI ROMA, 2010.

[2] D. S. Bolme, J. R. Beveridge, B. A. Draper, and Y. M. Lui. Visual object tracking using adaptive correlation filters. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2544–2550, June 2010.

[3] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2018.

[4] Zhe Chen, Zhibin Hong, and Dacheng Tao. An experimental survey on correlation filter-based tracking. *CoRR*, abs/1509.05520, 2015.

[5] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, June 2005.

[6] Michael J. Eager and Eager Consulting. Introduction to the dwarf debugging format, 2012. <http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf> [Accessed: 2018-05-2].

[7] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part based models. *Pattern Analysis and Machine Intelligence*, 32(9), 2010.

[8] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, Sept 2010.

[9] Bjorn Forsberg and Andrea Marongiu. High-performance real-time architectures for low-power embedded systems. Technical Report 3.2, ETHZ, 2017.

[10] Björn Forsberg, Daniele Palossi, Andrea Marongiu, and Luca Benini. Gpu-accelerated real-time path planning and the predictable execution model. *Procedia Computer Science*, 108:2428 – 2432, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[11] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista. High-speed tracking with kernelized correlation filters. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2015.

[12] João F. Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. Exploiting the circulant structure of tracking-by-detection with kernels. In Andrew Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid, editors, *Computer Vision – ECCV 2012*, pages 702–715, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] Jeffrey M. Intel ipp's 1d fourier transform functions. <https://software.intel.com/en-us/articles/how-to-use-intel-ipp-s-1d-fourier-transform-functions>, 2015. Accessed: 2018-05-8.

[14] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

[15] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. <http://pertsserver.cs.uiuc.edu/~mcaccamo/papers/PREM_talk_2011.pdf>, 2011.

[16] J. Reineke and R. Wilhelm. Impact of resource sharing on performance and performance prediction. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–2, March 2014.

[17] Sanjay Singh, Chandra Shekhar, and Anil Vohra. Real-time FPGA-based object tracker with automatic pan-tilt features for smart video surveillance systems. *Journal of Imaging*, 3(2), 2017.

[18] Carlo Tomasi. Histograms of oriented gradients. <https://pdfs.semanticscholar.org/8086/99bacf0cc91a38fa77b6af5a9f91dc25d127.pdf> [Accessed: 2018-04-8].

[19] Unknown. Elinux-jetson/computer vision performance. <https://elinux.org/Jetson/Computer_Vision_Performance>, 2015. Accessed: 2018-05-8.

[20] Unknown. Perf wiki, 2015. <https://perf.wiki.kernel.org> [Accessed: 2018-05-2].

[21] Unknown. Fftw3-manual. <http://www.fftw.org/fftw3.pdf>, 2017. Accessed: 2018-05-10.

[22] Unknown. Callgrind, 2018. <https://web.stanford.edu/class/cs107/guide/callgrind.html> [Accessed: 2018-05-3].

[23] Unknown. CuFFT-manual. <https://docs.nvidia.com/cuda/cufft/>, 2018. Accessed: 2018-05-10.

[24] Unknown. Mat the basic image container, 2018. <https://docs.opencv.org/2.4/doc/tutorials/core/mat_the_basic_image_container/mat_the_basic_image_container.html> [Accessed: 2018-05-2].

[25] Unknown. Opencv 2.4 documentation. <https://docs.opencv.org/2.4/index.html>, 2018. Accessed: 2018-05-8.

[26] Unknown. Opencv4tegra-introduction. <https://docs.nvidia.com/gameworks/content/technologies/mobile/opencv_intro.htm>, 2018. Accessed: 2018-05-8.

[27] Joost van de Weijer and Fahad Shahbaz Khan. An overview of color name applications in computer vision. In *Lecture Notes in Computer Science*, pages 16–22. Springer International Publishing, 2015.

[28] Tomáš Vojíř. *Short-Term Visual Object Tracking in Real-Time*. Phd thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2017.

[29] F. Xu, H. Wang, Y. Song, and J. Liu. A multi-scale kernel correlation filter tracker with feature integration and robust model updater. In *2017 29th Chinese Control And Decision Conference (CCDC)*, pages 1934–1939, May 2017.

Appendix A

Contents of CD