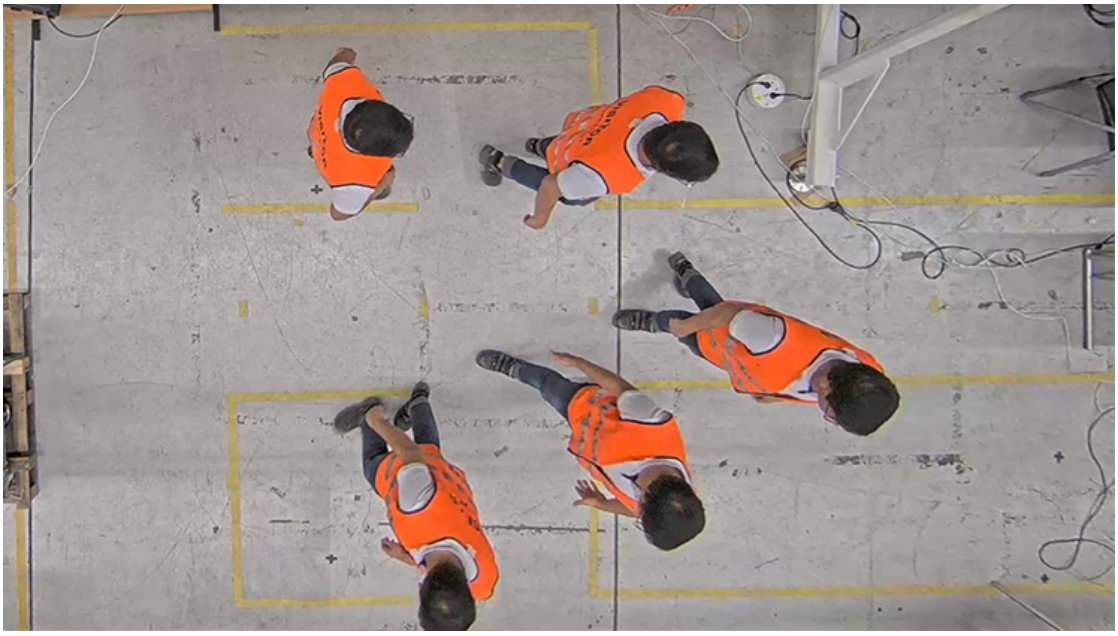

AI/ML Algorithms for Video Data Filtration

A machine learning based video filtration pipeline with a prioritization scheduling algorithm for a multi-camera system

Master's thesis in Computer science and engineering

SIDDHARTH AMIN
DEAN ATWINE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

AI/ML Algorithms for Video Data Filtration

A machine learning based video filtration pipeline with a prioritization scheduling algorithm for a multi-camera system

SIDDHARTH AMIN
DEAN ATWINE

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

AI/ML Algorithms for Video Data Filtration
A machine learning based video filtration pipeline with a
prioritization scheduling algorithm for a multi-camera system
SIDDHARTH AMIN
DEAN ATWINE

Supervisor: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering
Advisor: Harald Gustafsson, Ericsson AB
Examiner: Philippas Tsigas, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Overlay of images of a person walking around a factory floor.

Typeset in LaTeX
Gothenburg, Sweden 2022

AI/ML Algorithms for Video Data Filtration
A machine learning based video filtration pipeline with a
prioritization scheduling algorithm for a multi-camera system
SIDDHARTH AMIN
DEAN ATWINE
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Video cameras are ubiquitous in today's society, with cities and organizations steadily increasing the size and scope of their deployments. These applications have benefited from cloud computing's large-scale computing and storage capabilities over the last two decades. The massive amounts of data generated by these high-definition cameras are proving too large to transport and process in real-time in the cloud. Many critical applications, such as public safety, surveillance, and traffic control, rely heavily on video cameras. Filtering out frames that do not contain relevant information for the query at hand is a common (and natural) strategy used by systems to improve efficiency. However, this necessitates that the filtering algorithm can contextually decide on if the frame is relevant or not. This research looks into the creation of a video analytics pipeline that uses computer vision tasks, object classification models, and a prioritization algorithm to effectively filter frames from multiple cameras while dealing with the over subscription of streams on a processing node and sending only relevant frames for further processing. In this thesis, we examine multiple light-weight computer vision and classification models that can be used to classify if a frame has a contextually interesting object. We then design a pipeline where we use techniques such as frame-differencing, light-weight Deep-Neural Networks(DNNs), and a frame prioritization algorithm to decide on which frames would be processed in the case of overprovisioning and in what order. Our results show that our framework can accommodate up to 85% more streams than running with out the framework.

Keywords: Computer Vision, Object Classification, Video Filtration Pipeline, Machine Learning, Prioritization Scheduling.

Acknowledgements

We want to thank Ahmed Hassan & Harald Gustafsson for the great support and guidance they've provided throughout the project. We also want to thank AI Sweden for providing us with an office space from where we could work and carry out our research.

Siddharth Amin & Dean Atwine, Gothenburg, June 2022

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Overview . . . 1
1.1.1 Aim . . . 2
1.2 Related Work . . . 2

2 Theory **5**
2.1 Image Processing . . . 5
2.1.1 Convolutional Neural Networks . . . 5
2.1.2 Image Processing Tools . . . 7
2.1.3 Image Filtration . . . 8
2.1.3.1 Background Subtraction . . . 8
2.2 OpenCV . . . 8
2.2.1 OpenCV CUDA Module . . . 9
2.3 CUDA & Graphical Processing Unit . . . 9
2.4 Datasets . . . 10
2.5 Modern Streaming Infrastructure . . . 11
2.5.1 Docker . . . 11
2.5.2 Kubernetes . . . 12
2.5.3 GStreamer . . . 13

3 Methods **15**
3.1 System Architecture . . . 15
3.2 Cascade Approach . . . 16
3.3 Streaming Pipeline . . . 17
3.3.1 Single Stream Pipeline . . . 17
3.3.2 Multi Stream Pipeline . . . 18
3.4 Frame Differencing . . . 19
3.4.1 Frame Subtraction . . . 19
3.4.2 Background Subtraction . . . 21
3.5 Object Classification . . . 22
3.5.1 Expanding the Dataset . . . 23
3.5.2 MobileNetV2 Model . . . 24

4 Results **27**
4.1 Streaming 27
4.2 Frame Differencing 29
4.3 Object Classification 29
4.4 Optimization 32
4.4.1 Frame Differencing with CUDA 32
4.4.2 Optimizing Inference Time with TensorRT 32
4.5 Pipeline performance 33

5 Conclusion **35**

Bibliography **37**

A Appendix 1 **I**

List of Figures

2.1 Example of a CNN Network 6
2.2 TensorRT model optimization pipeline[17] 8
2.3 GPU Architecture representation of Kernels, Grids & Blocks 9
2.4 Images from the COCO dataset 10
2.5 Kubernetes Architecture Components[27] 12
2.6 Gstreamer linked elements[31] 13

3.1 Cascade approach to the filtration process 16
3.2 Gstreamer pipeline of elements to receive data from a single stream . 17
3.3 Gstreamer pipeline of elements to receive data from 3 streams 18
3.4 Frame Subtraction performed on a frame and empty floor a) Empty floor b) Current frame c) Difference in frames d) Threshold performed on the difference frame 19
3.5 Frame Subtraction performed on a frame and a previous frame a) Empty floor b) Current frame c) Difference in frames d) Thresholding performed on the difference frame 20
3.6 Frame examples from running Background Subtraction 21
3.7 Background Subtraction performed on frames and subtraction between the frames 21
3.8 Images from the dataset of ceiling mounted camera angle at the Volvo factory 24

4.1 Graph comparing the difference in decoder usage for video inputs of a)10fps and b)30fps 28
4.2 Graph depicting the latency distribution for a) Frame Subtraction and b) Background Subtraction running on the CPU 29
4.3 Graph depicting the training loss and validation accuracy of the custom model across 20 epochs. 30
4.4 Graph depicting the training loss and validation accuracy of the custom model on training it with the new dataset across 10 epochs. . . . 30
4.5 Graph depicting MobileNetV2 training loss and validation accuracy. . 31
4.6 Graph depicting the latency distribution for a) base model and b) mobilenetv2 31
4.7 Graph depicting the latency distribution for a) Frame Subtraction and b) Background Subtraction running on the GPU using openCV CUDA module 32

4.8 Graph depicting the latency distribution for a) Base model without any optimization and b) Base model optimized with tensorRT to improve inference time. 33
4.9 Graph comparing the overall run time of the pipeline handling frames from multiple streams. 34
4.10 Graph depicting the latency distribution for the pipeline handling multiple streams. 34

A.1 Detailed Gstreamer pipeline for a single stream I
A.2 Detailed Gstreamer pipeline for a multiple streams I
A.3 Sequence of images resulting from the frame subtraction method highlighting use of a previous frame that triggers the threshold I

List of Tables

4.1 Table of results depicting the percentage usage of the hardware decoder available on the GPU 27
4.2 The frame difference classification and total run time of the pipeline while handling frames from multiple streams. 33

1

Introduction

1.1 Overview

In today's society, video cameras are ubiquitous, with cities and organizations steadily expanding the size and scope of their deployments. Many critical applications rely heavily on video cameras, including public safety, surveillance, and traffic control. In the last two decades, these applications have benefited from cloud computing's large-scale computing and storage capabilities. According to a multi-phased market analysis[1] of the visual technology ecosystem by LDV Capital, an estimated 45 billion surveillance cameras were predicted to be installed around the world by 2022 and most of these cameras would require human operation. The ever-increasing number of cameras installed around the world, on the other hand, poses a serious challenge to the current cloud-based infrastructures if all this data was to be processed. The massive amounts of data generated by these high-definition cameras are proving too large to transport to the cloud and process in real-time.

Moving objects, motion tracking, and human detection are commonly used in real-time surveillance systems that can automatically detect and track the presence of objects in an environment with few moving objects, and are primarily applied to computer vision systems that can replace human roles in the field of safety surveillance[2]. Machine Learning (ML) is increasingly being used in video analytics to complete various tasks by applying computer vision and deep learning to video footage or live video streams. Video analytics has evolved from old algorithms based solely on Computer Vision to adding strong Deep Learning approaches as a result of advances in Deep Learning research and increased availability of video data with the rise of global video camera networks. Deep learning is a class of machine learning techniques that are used to extract features from data and Convolutional Neural Networks have been found suitable for computer vision tasks [3].

Deep learning algorithms serve as the foundation for object recognition, a key task in video content analysis. They are used to train the system to detect and classify people and objects in video. Object recognition is a computer vision technique that allows you to identify objects in photos or videos. Image categorization and object recognition are challenging tasks, especially when dealing with complex photos that contain multiple items. Despite their appealing advantages and the relative

efficiency of their local architecture, CNNs have remained prohibitively expensive to use on a large scale for high-resolution images. Fortunately, today's Graphical Processing Units (GPUs) are powerful enough to train huge CNNs when combined with a well-optimized implementation of 2D convolution. [3]

Video analytics is an emerging technology, current video analytics applications face challenges in perceiving complex events in real-time on a large scale [4]. To begin, latency is a critical requirement for many video analytics applications including traffic and accident prediction[5], self-driving cars[6], Virtual Reality(VR)[7] and Augmented Reality(AR)[7]. Second, high definition video data consumes a lot of bandwidth[8]. The requirement for real-time processing on edge devices with low computation power, on the other hand, presents a challenge for such applications. Many approaches to optimizing the latency and bandwidth consumption of Edge-Cloud video processing have been proposed, particularly for the increasingly complex Neural Networks (NN)-based methods. Significant efforts have been made to improve the efficiency of video analytics pipelines[9][10].

A common (and natural) strategy used by these systems to improve efficiency is to filter out frames that do not contain relevant information for the query at hand [9][10]. However, filtering out a frame conceptually necessitates understanding how that frame would affect a query result. End users can gain situational awareness by configuring rule-based alerts based on video objects and behaviors, and operational intelligence by visualizing video data in dashboards and heatmaps for trend analysis[4].

1.1.1 Aim

This thesis is an industrial thesis done with Ericsson Research and Volvo Trucks GTO. The aim is to filter video streams from multiple cameras to reduce the total number of GPUs required to automate a Volvo Trucks factory. The study develops a video analytics pipeline that employs computer vision tasks, object classification models and a prioritization algorithm designed to effectively handle frame filtering from multiple cameras, while handling oversubscription of streams on a processing node sending only relevant frames for further processing, in this case, pose estimation. In order to find the best fit for the pipeline being developed, multiple computer vision techniques and classification models are reviewed.

1.2 Related Work

This project builds on a long history of data management for multimedia and videos, as well as recent advances in computer vision and machine learning, which are outlined below.

Noscope[10] reduces the cost of neural network video analysis by up to three orders of magnitude through inference-optimized model search. Given an input video, a target object, and a reference network, Noscope searches for and trains a cascade of models, including difference detection and specialized networks, that can reproduce the reference network's binarised outputs with high accuracy but up to three orders of magnitude faster. NoScope achieves two to three orders of magnitude speedups (265-15,500x real-time) on binary classification tasks over fixed-angle webcam and surveillance video, while maintaining accuracy within 1-5 percent of state-of-the-art.[10]

Reducto[9]is a video analytics system that enables efficient real-time querying by utilizing previously unused resources to perform on-camera frame filtering. This paper investigates on-camera filtering, which relocates filtering to the beginning of the pipeline, and develops a system that dynamically adapts filtering decisions based on the time-varying correlation between feature type, filtering threshold, query accuracy, and video content. According to the findings of this paper, Reducto achieved significantly higher filtering benefits (53.4 percent) than Tiny YOLO (24.36 percent) and Filter Forward (27.7 percent), resulting in an improvement of about 54-63 percent and 53-61 percent in network bandwidth used and back end processing costs, respectively.

2

Theory

Computer vision is a rapidly expanding field devoted to image analysis, modification, and high-level understanding. Its goal is to figure out what's going on in front of a camera and use that knowledge to control a computer or robotic system, or to provide people with new images that are more informative or visually appealing than the original camera images. Video surveillance, biometrics, automotive, photography, film production, web search, medicine, augmented reality gaming, new user interfaces, and many other applications are possible with computer vision technology[11]

2.1 Image Processing

The process of enhancing and extracting useful information from images is known as image processing. It is one of the fastest growing technologies and has evolved significantly over the years. Image processing is used in a variety of applications today, including visualization, image information extraction, pattern recognition, classification, segmentation, and many others. It is also important to understand that image processing algorithms exist and play an important role in image processing. A machine-learning algorithm uses data from processed images to map known and unknown feature vectors. These algorithms learn from patterns using training data with specific parameters. Deep learning networks are used in several machine learning image processing techniques.

2.1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are one of the most widely used methods of object recognition. It is widely used, and most cutting-edge neural networks employ it for various object recognition tasks such as image classification. The CNN network takes an image as input and outputs the probability of each class. If the object is present in the image, its output probability is high; otherwise, the output probability of the remaining classes is either negligible or low.

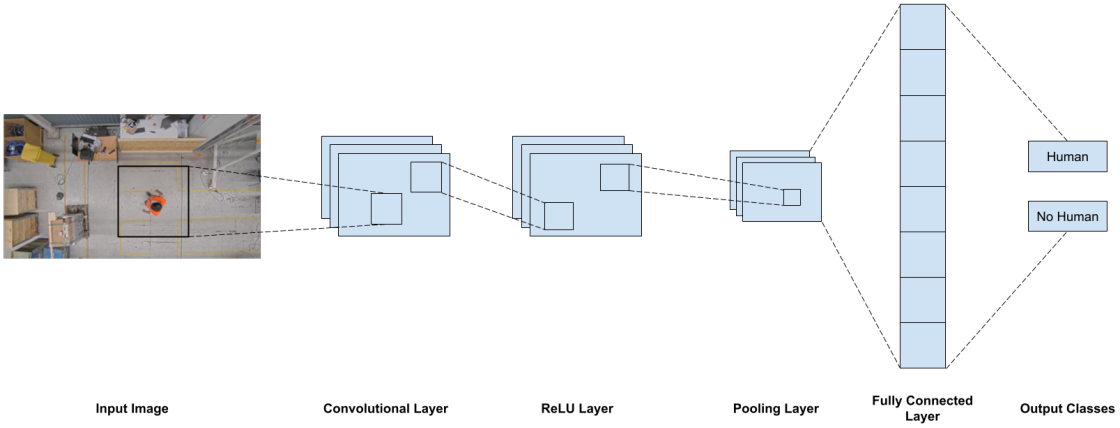

Figure 2.1: Example of a CNN Network

CNN architecture has changed dramatically between 1989 and today. Changes such as structural reformulation, regularization, parameter optimizations, and so on are examples of such modifications[12]. LeCun et al. first introduced CNNs in their paper "Object recognition with gradient-based learning" in 1989. CNNs have made significant progress since then, with CNN-based applications becoming common in 2012 following AlexNet's[13] exemplary performance on the ImageNet[13] dataset, to deeper architectures like ResNet[14] with 20 layers and GoogleNet[15] with 28 layers, as examples.

The CNN architecture consists of a number of layers. Main layers in the CNN architecture, including its function, are described in detail below.

Convolutional Layer:
This is the most important component of the convolutional neural network and is made up of convolutional filters or kernels.The kernel is described by a grid of discrete numbers or values, with each value referred to as the kernel weight. At the start of the CNN training process, random numbers are assigned to act as the kernel's weights, and these weights are adjusted at each training era; thus, the kernel learns to extract significant features. The input images are convolved with these filters to generate the output feature map, which is expressed as N-dimensional metrics. The vector format is the traditional neural network's input, whereas the multi-channeled image is the CNN's input. The gray-scale image format, for example, is single-channel, whereas the RGB image format is three-channeled[12].

Pooling Layer:
The pooling layer's main task is to sub-sample the feature maps, which are generated by shrinking the large-size feature maps to create smaller feature maps and, similarly to the convolution operation, both the stride and the kernel are initially size-assigned before the pooling operation is executed[12].

Activation Function:

The core function of all types of activation functions in all types of neural networks is mapping the input to the output. The input value is computed by taking the weighted sum of the neuron input and its bias (if present). This means that the activation function decides whether or not to fire a neuron in response to a specific input by producing the corresponding output[12].

Fully Connected Layer:
This layer is typically found at the end of each CNN architecture[12]. Each neuron in this layer is connected to all neurons in the previous layer, using the so-called Fully Connected approach. It serves as the CNN classifier. As a feed-forward Artificial Neural Network, it adheres to the basic method of the conventional multiple-layer perceptron neural network. The fully connected layer receives its input from the previous pooling or convolutional layer, which is in the form of a vector created from the feature maps after flattening. The fully connected layer's output represents the final CNN output.

Loss Functions:
The final classification is accomplished by the output layer, which is the final layer of the CNN architecture. In the CNN model, some loss functions are used in the output layer to calculate the predicted error created across the training samples[12]. This error shows the difference between the actual and predicted output. The loss function, on the other hand, uses two parameters to calculate the error. The first parameter is the CNN estimated output (referred to as the prediction), and the second parameter is the actual output (referred to as the label).

Transfer learning involves training and comparing a pre-existing model with a different architecture. The goal is to see if there is an improvement in accuracy as the number of layers increases. This means starting the model with a pre-trained CNN, preferably one trained with a large number of images. This allows for a good convolutional base to be used before adding a dense layered classifier at the end, and using this technique, a very good classifier can be trained for a relatively small dataset of 10,000 images or more.

2.1.2 Image Processing Tools

With the recent surge in interest in deep learning, there has been a proliferation of machine learning tools. Deep learning frameworks such as PyTorch, TensorFlow, Keras, Chainer, and others have been introduced and developed at a rapid pace in recent years[16]. The two most popular machine learning frameworks that support artificial neural network models are TensorFlow and PyTorch. When analyzing large and complex data sets, using artificial neural networks is an important approach for drawing inferences and making predictions[16]. To assemble and train neural network models, these frameworks provide neural network units, cost functions, and optimizers.

NVIDIA TensorRT is a software development kit that enables high-performance machine learning inference. It is intended to work in conjunction with training frameworks. It is designed to run an already-trained network quickly and efficiently on NVIDIA hardware. Figure 2.2 depicts TensorRT's workflow in which the Builder module creates an optimized inference engine with given optimization configuration parameters and network definition. It uses layer-wise optimization and auto-tuning to choose the best algorithm and data format for each layer. Other memory optimization techniques, such as layer fusion and dynamic tensor memory, are also provided[17]. Furthermore, by configuring parameters, a user can employ approximation-based techniques such as mixed precision. During the TensorFlow with TensorRT (TF-TRT) optimization, TensorRT performs several important transformations and optimizations to the neural network graph.

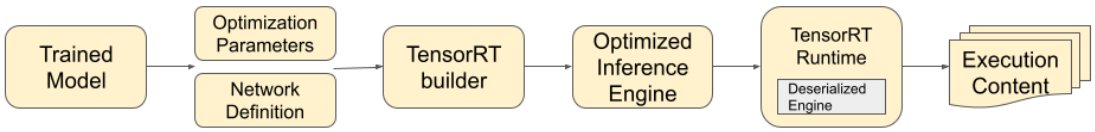

Figure 2.2: TensorRT model optimization pipeline[17]

2.1.3 Image Filtration

2.1.3.1 Background Subtraction

Background Subtraction is a common and widely used preprocessing step in many vision-based applications for generating a foreground mask (that is, a binary image containing the pixels belonging to moving objects in the scene) using static cameras, in which the camera angle does not change and remains fixed in one location. Background Subtraction computes the foreground mask by subtracting the current frame from a background model that contains the static portion of the scene or, more broadly, everything that can be considered background given the characteristics of the observed scene [18][19].

2.2 OpenCV

OpenCV is a free and open source computer vision and machine learning library written in C/C++/Python that is available for Windows, Linux, macOS, and Android. It includes low-level image processing functions as well as high-level algorithms for object recognition, face recognition, and video action classification. OpenCV was created to provide a common infrastructure for computer vision applications and to speed up the incorporation of machine perception into commercial products[20].

2.2.1 OpenCV CUDA Module

The GPU module in OpenCV, covers a significant portion of the library's functionality. It is built with CUDA and thus benefits from the CUDA ecosystem, which includes libraries like NPP (NVIDIA Performance Primitives). The GPU module enables users to benefit from GPU acceleration without prior knowledge of GPU programming. The module is compatible with the CPU version of OpenCV, making it simple to implement. However, there are differences, the most significant of which is the memory model[11].

2.3 CUDA & Graphical Processing Unit

The CUDA parallel programming model prioritizes two key design objectives. First, it aims to provide a minimal set of abstractions for expressing parallelism in a standard sequential programming language, specifically C/C++. In general, this allows the focus to be on the important issues of parallelismhow to create efficient parallel algorithmsrather than the mechanics of an unfamiliar and complicated language. Second, it is intended for the development of highly scalable parallel code that can run on tens of thousands of concurrent threads and hundreds of processor cores. This is critical because current NVIDIA GPUs have physical parallelism ranging from eight processor cores and 768 thread contexts to 240 processor cores and 30,720 thread contexts.

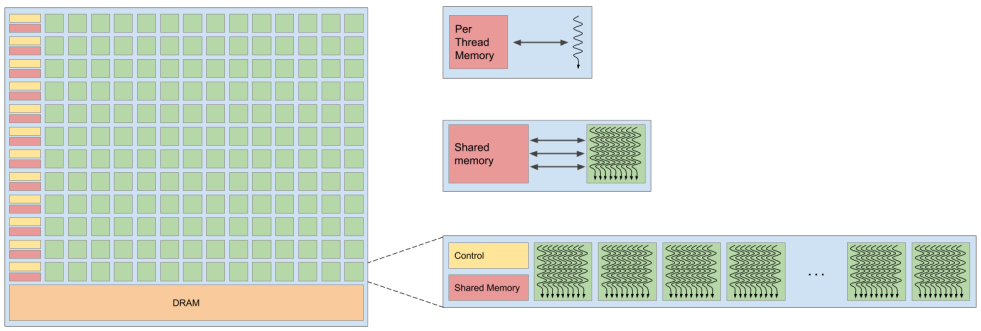

Figure 2.3: GPU Architecture representation of Kernels, Grids & Blocks

A CUDA program is divided into a host program, which includes one or more sequential threads running on the host CPU and one or more parallel kernels suitable for execution on a parallel processing device such as the GPU, shown in Figure 2.3. On a set of parallel threads, a kernel executes a scalar sequential program. The threads are organized into thread blocks in a grid. Threads within a single thread block can synchronize with one another via barriers and have access to a high-speed, per-block shared on-chip memory for inter-thread communication. Threads from different grid blocks can only communicate via operations in a shared global memory space that is visible to all threads.

2.4 Datasets

The majority of classification models are trained supervised. As the name suggests, supervised learning is defined by the use of labeled datasets to train algorithms to accurately classify data or predict outcomes. As input data is fed into the model, the weights are adjusted until the model is properly fitted; this occurs as part of the cross validation process.

In general, the data is divided into two parts. The first is training data, which is used to determine the parameters of a data point which produce model results that are closest to the associated outcome. The other portion, the test data, is used primarily to evaluate the model's performance but should not be used to influence model parameters. In some cases, a portion of the dataset is set aside as validation data, which is then used to select the best performer from a group of models that may use entirely different algorithms or training methods[21].

A classifier model's typical output is a vector that represents the probability that the output belongs to a specific class, and the values sum to 1. Models that can perform binary classification, with the output indicating whether or not a specific class exists. The model produces a single probability between 0 and 1, with a value around 0.5 indicating "positive" and less than 0.5 indicating "negative"[21].

Figure 2.4: Images from the COCO dataset

Tsung-Yi Lin et. al. present a new dataset with the goal of advancing the state-

of-the-art in object recognition by placing the question of object recognition in the context of the broader question of scene understanding[22]. The Microsoft Common Objects in COntext (MS COCO) dataset contains 91 common object categories with 82 of them having more than 5,000 labeled instances, Fig. 2.4 displays a few samples from the dataset. In total the dataset has 2,500,000 labeled instances in 328,000 images. In contrast to the popular ImageNet dataset [23], COCO has fewer categories but more instances per category[22].

2.5 Modern Streaming Infrastructure

The increasing complexity of modern streaming applications poses new challenges in system design today. For example, applications have evolved to the point where hard-real-time execution on multiprocessor platforms is required in many cases to meet the applications' timing requirements. Furthermore, in some cases, it is necessary to run a number of such applications concurrently on the same platform, with support for accepting new incoming applications at run-time. Dealing with all of these new challenges increases the complexity of system design significantly.[24]

Edge computing brings compute resources, dependable network infrastructure, and real-time capabilities closer to devices. Container technology is primarily used to provide resources and workloads at the edge. The proper placement of containerized workloads in terms of when, where, and how to provide them is still a problem domain. Containerization and orchestration are becoming a common place in this context because they enable better information flow between network levels as well as increased modularity in the use of software components.

2.5.1 Docker

Docker is an open platform for app development, shipping, and running. Docker allows for the separation of applications from infrastructure, allowing for faster software delivery. Docker manages infrastructure in the same way that it manages applications[25]. Docker enables the packaging and execution of applications within containers, which are loosely isolated environments. By default, a container is relatively well isolated from other containers and its host machine[25], and how well the network, storage, or other underlying subsystems are isolated from other containers or the host machine can be specified. Many containers can run on a single host at the same time and share resources because of the isolation and security[25]. Containers are small packages that contain all of the files and libraries that an application requires to function properly. Docker spawns containers using images. An image can be used to create multiple containers.

An image is a read-only template containing Docker container creation instructions. An image is typically based on another image, with some additional customization[25]. A container is a runnable image instance. A container can be linked to one or more networks, connected to storage, or even used to generate a new image

based on its current state. A container's image, as well as any configuration options provided to it when it is created or started, define it[25].

2.5.2 Kubernetes

Kubernetes is all about orchestration of containers. It is an open source platform for managing containerized workloads and services that allows for declarative configuration as well as automation[26].

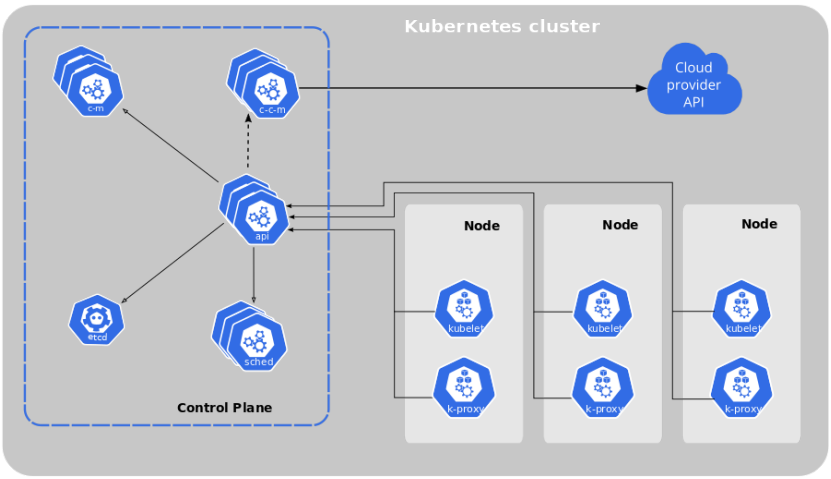

Figure 2.5: Kubernetes Architecture Components[27]

A Kubernetes cluster is made up of a collection of worker machines known as nodes that run containerized applications. Each cluster contains at least one worker node. The worker node(s) host the Pods that make up the application workload. Kubernetes manages the workload by dividing it into Pods that run on Nodes. Depending on the cluster, a node can be a virtual or physical machine. The control plane manages the cluster's worker nodes and Pods. In production environments, the control plane is typically distributed across multiple computers, and a cluster is distributed across multiple nodes to provide fault tolerance and high availability. The control plane manages each node, which contains the services required to run Pods[27][26].

Due to its rich set of APIs, reliability, scalability, and performance features, Kubernetes has quickly become the platform of choice for deploying complex applications built on top of numerous microservices. The device plugin framework in Kubernetes allows access to special hardware resources such as NVIDIA GPUs, NICs, Infiniband adapters, and other devices. Configuring and managing nodes with these hardware resources, on the other hand, necessitates the configuration of multiple software components such as drivers, container runtimes, or other libraries, which is difficult and error-prone[28]. The Kubernetes Operator Framework takes operational business logic and creates an automated framework for the deployment of applications within Kubernetes using standard Kubernetes APIs and kubectl. Based on

the operator framework, the NVIDIA GPU Operator introduced here automates the management of all NVIDIA software components required to provision GPUs within Kubernetes[28].

2.5.3 GStreamer

GStreamer is a framework that includes components for creating a media player that supports a wide range of formats such as MP3, Ogg/Vorbis, MPEG-1/2, AVI, Quicktime, mod, and others. GStreamer, on the other hand, is much more than just a media player. The main benefits are that the pluggable components can be mixed and matched into arbitrary pipelines, allowing for full-fledged video or audio editing applications to be written.

The framework is built on plugins that provide various codecs and other functionality [29] and has a compact core library, resulting in low latency and very high performance[30]. The plugins work together to manage multimedia flows and media travels from source (the producers) to sink (the consumers), shown in Figure A.3.

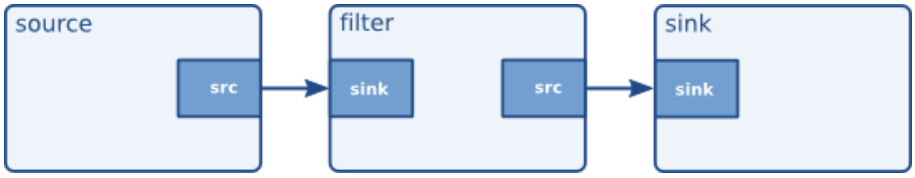

Figure 2.6: Gstreamer linked elements[31]

3

Methods

During the course of this project, multiple computer vision techniques were applied and evaluated. Furthermore, object classification models were trained and predictions were tested to evaluate their performance. The techniques and models that performed well were there tested as part of the video filtration pipeline to assess it's performance and document these results.

3.1 System Architecture

The application makes use of one or more cameras or capture hardware devices that provide video input to the processing pipeline. For this implementation, the streaming task is performed using the Real Time Protocol (RTP) and encoded using the H.264 advanced video coding standards. Multiple pods within a Kubernetes cluster are used to simulate these streams. The video input stream used to represent the camera feed is looped and sent to a predefined UDP port on the pod running the filtration pipeline within the cluster.

The node is linked to an Nvidia Tesla V100 GPU with 16GB RAM, integrated via the Nvidia GPU operator. Multiple pods are deployed to stream the video file, while a separate pod receives the streams and runs the pipeline application. All pods run the same docker container assigned using the YAML configuration file.

The primary goal of this study, using this architecture, is to determine whether or not the objects of interest in a specific environment are in the frame of a camera used to monitor that environment. Humans and robots play critical roles on the factory floor in modern factories, frequently collaborating to complete tasks. It is critical to locate them in order to avoid conflicts and ensure efficiency and safety. To track them, the pose estimation technique MoveNet[32][33] is used to locate these objects of interest within the given frame of the factory floor. Pose estimation, on the other hand, is a fairly heavy algorithm in that it requires initial frame filtering, for which we design and implement a video analytics pipeline.

3.2 Cascade Approach

In this thesis we have designed a processing pipeline Cascade as shown in Figure 3.1. This cascade determines if a frame needs further processing or not. The first stage, called the stream-level ranking stage, determines the priority ranking of the frames in the batch to be processed, by calculating the rank using values from previously processed frames. The frames from high ranking streams are sent to the pose estimation task to determine whetherthe position of a human or robot in the frame, if any.

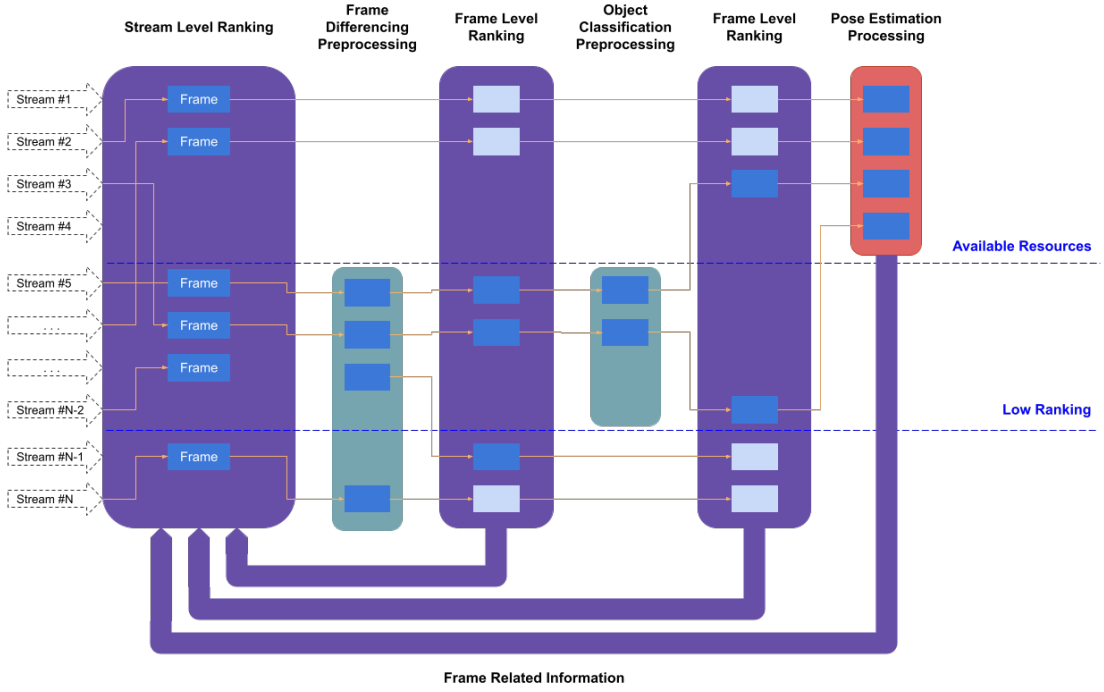

Figure 3.1: Cascade approach to the filtration process

If there are still available resources, the remaining frames are sent to the next stage called frame-differencing pre-processing module to further assess the priority of the frame and the stream it belongs to. The list of frames is then sorted based on the results of this pre-processing stage. The higher-ranked frames with a high degree of activity are then passed on to the pose estimation task, given that resources are available. Frames are further processed if there are available resources by sending them to the Object Classification pre-processing module to detect classes of objects of interest, and the results of this stage enable frames to be further sorted and sent to the pose estimation task.

The cascade approach allows for processing of frames that are of relevance - that is, they have a high degree of activity and have objects of relevance frequently appearing

within the frame. The priority for each stream and its frames are calculated using equation 3.1.

$$R_{stream} = O_{present} \times (C_{previous} + P_{previous}) + (C_{average} + P_{average}) + elapsed\_time \quad (3.1)$$

where,
$O_{present}$ = Boolean value representing whether an object of interest was in the previous frame processed.
$C_{previous}$ = value returned from the frame-differencing module
$P_{previous}$ = confidence value returned from the object classification module
$C_{average}$ = historic average of the activity value maintained for the stream
$P_{average}$ = historic average of the confidence value maintained for the stream
$elapsed\_time$ = time elapsed between the last time a frame was processed and the time the rank was calculated.

The time aspect of the rank formula allows for fair scheduling. Any low ranking stream, not having regular activity or objects of interest within usually would rank low. The *elapsed_time* value increases the rank of the stream to ensure the frames from the stream get processed after certain cycles of processing.

3.3 Streaming Pipeline

3.3.1 Single Stream Pipeline

As depicted in Figure 3.2, the multimedia passes through a series of intermediate elements organized in a pipeline to complete all tasks [29]. This pipeline defines the data flow and includes the modules listed below and highlighted in Figure 3.2 for the implementation in this study.

Figure 3.2: Gstreamer pipeline of elements to receive data from a single stream

The pipeline's first element is the udpsrc, which reads UDP packets from the network. To implement RTP streaming, the udpsrc element can be used in conjunction with RTP depayloaders.The udpsrc element is followed by the rtpbin element, which includes the rtpssrcdemux element, which acts as a demuxer for RTP packets based on their Synchronization Source(SSRC). Its primary function is to enable an application to easily receive and decode an RTP stream with multiple SSRCs. Data received on the requested sink pad will be processed in the manager and forwarded on the element after validation. Each RTP stream is demuxed based on its SSRC,

and a source pad for that stream is created, which is identified by its session number, SSRC, and payload type [34]. The demuxed streams then send the data to the h264depay element, which extracts the h264 video from the RTP packets and sends it to the h264parse element, which then sends it on to the decoder.

Most Nvidia dGPUs come with a dedicated hardware decoder which allows faster decoding using software decoders. The Deepstream SDK[35] by Nvidia is used to plug the necessary elements into the pipeline, allowing the Gstreamer pipeline to leverage the GPU to decode the frames/data received. DeepStream extends the open source V4L2 codec plugins to support hardware-accelerated codecs. The Nvv4l2decoder plugin leverages the hardware decoding engines on Jetson and DGPU platforms. The module/element accepts H264 stream, but outputs it in the NV12 format. To be able to use these frames it is converted to a relevant format using the nvvideoconvert element. The Nvvideoconvert plugin converts video color formats, and also allows the frames to accessed from the GPU after converting them to the desired format with the memory:NVMM capability.

3.3.2 Multi Stream Pipeline

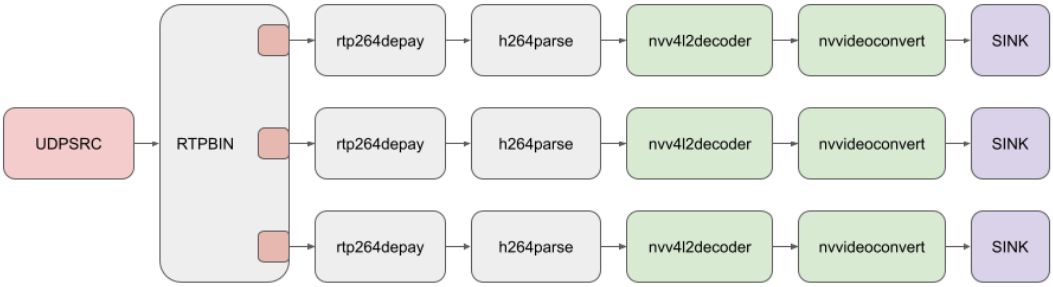

Figure 3.3: Gstreamer pipeline of elements to receive data from 3 streams

When working with multiple streams, each stream is given its own chain made up of the h264depay, h264parse, nvv4l2decoder, nvvideoconvert, and sink elements. The frames for each stream are extracted from their respective sink buffers. Figure 3.3 depicts an example of a multistream pipeline receiving data from 3 streams. These buffered frames are in the RGBA format and are collected in batches and only the most recent frames are sent for further processing, i.e., those that fall within a certain window. These frames are then passed to the stream-ranking module to be sorted based on rank and processed accordingly.

3.4 Frame Differencing

The Frame Differencing module's main goal is to detect motion within a stream's data using low latency computer vision techniques. Image subtraction and background subtraction methods are evaluated in this task to determine motion within the frame, while in this chapter, any movement within the frame is defined as activity. The degree of activity within the stream is calculated using image subtraction operations performed on different sets of images.

3.4.1 Frame Subtraction

The subtraction method consists of two steps. The first step is to subtract the current frame from one with an empty floor, as illustrated in figure 3.4. The second step is to calculate frame movement by subtracting the current frame from previous frames. This enables the system to determine stream activity, as shown in Figure 3.5.

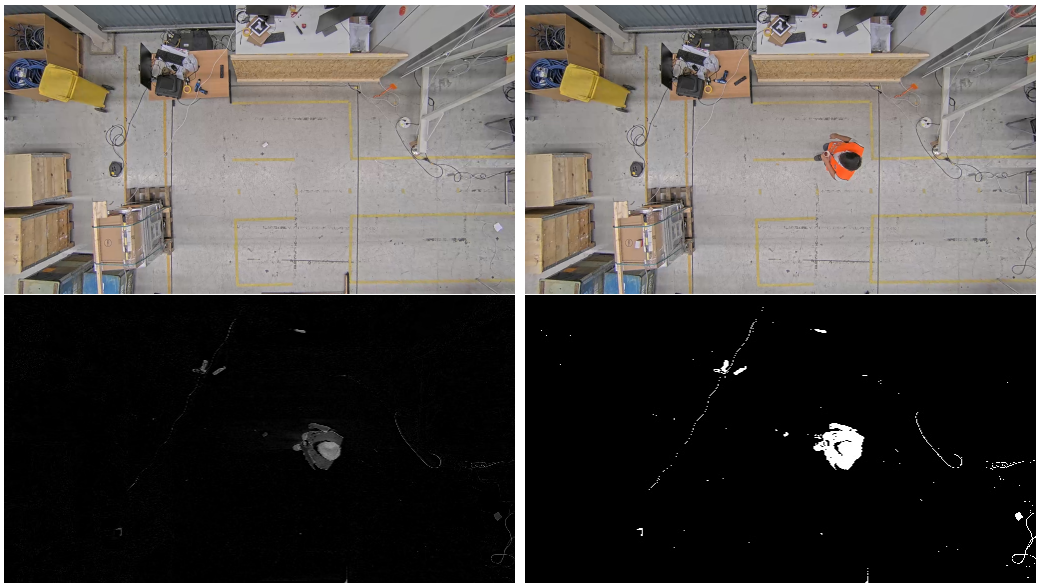

Figure 3.4: Frame Subtraction performed on a frame and empty floor a) Empty floor b) Current frame c) Difference in frames d) Threshold performed on the difference frame

Figure 3.5: Frame Subtraction performed on a frame and a previous frame a) Empty floor b) Current frame c) Difference in frames d) Thresholding performed on the difference frame

The difference between the frames is calculated using the image subtraction method to determine a threshold of activity defined as A_{low} to determine whether further processing is required within this module and after. The activity is determined by calculating the area of difference between frames, denoted as A; if $A > A_{low}$, it is determined that there is enough activity within the frame to continue processing. The equation 3.2 is used to calculate the activity percentage at each step.

$$A = (Sum\ of\ pixel\ values/(height * width)) * 100 \tag{3.2}$$

OpenCV allows certain functions to use GPU resources to perform image processing functions. Within this module, these functions are used to reduce the latency of the pipeline. The OpenCV GPU functions get a new class called cv::gpu::GpuMat (or cv2.cuda GpuMat in Python), which acts as a primary data container for input and output arguments[11].

Listing 3.1 depicts the processing flow, wherein the data is uploaded to the GPU from CPU memory and then processed to calculate the activity metric.

```
cv::cuda::gpuMat gpu_frame, emptyframe_gpu, empty_diff
cv::cuda::resize(gpu_frame, (512,288))
cv::cuda::cvtColor(gpu_frame, cv::COLOR_RGBA2GRAY)
cv::cuda::subtract(emptyframe_gpu, gpu_frame, empty_diff)
cv::cuda:threshold(empty_diff, 50 , 255, cv::THRESH_BINARY)
```

Listing 3.1: Code snippet for the frame subtraction process

The cv::gpuMat output is converted to a CuPy array using the cuda array interface, that was created to allow different implementations of GPU array-like objects to interact with one another. Converting to a compatible CUDA array using this interface, allows the necessary CuPy functions to accept it as input[36].

3.4.2 Background Subtraction

Background subtraction methods in OpenCV include BackgroundSubtractorMOG()[37], BackgroundSubtractorGMG()[38] and the more recent BackgroundSubtractorMOG2()[39][40]. The MOG2() method provides a significant improvement for the task of Background Subtraction by using an automatic selection scheme for the number of Gaussian kernels used for each pixel instead of the original MOG's constant number of distribution kernels. As a result, MOG2 is more adaptable to changing lighting conditions in scenes. Figure 3.6 displays the masked frames generated by the MOG2() method with varying parameters.

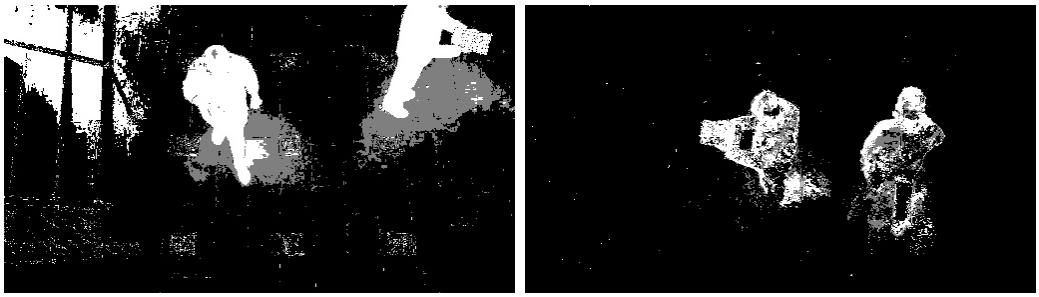

Figure 3.6: Frame examples from running Background Subtraction

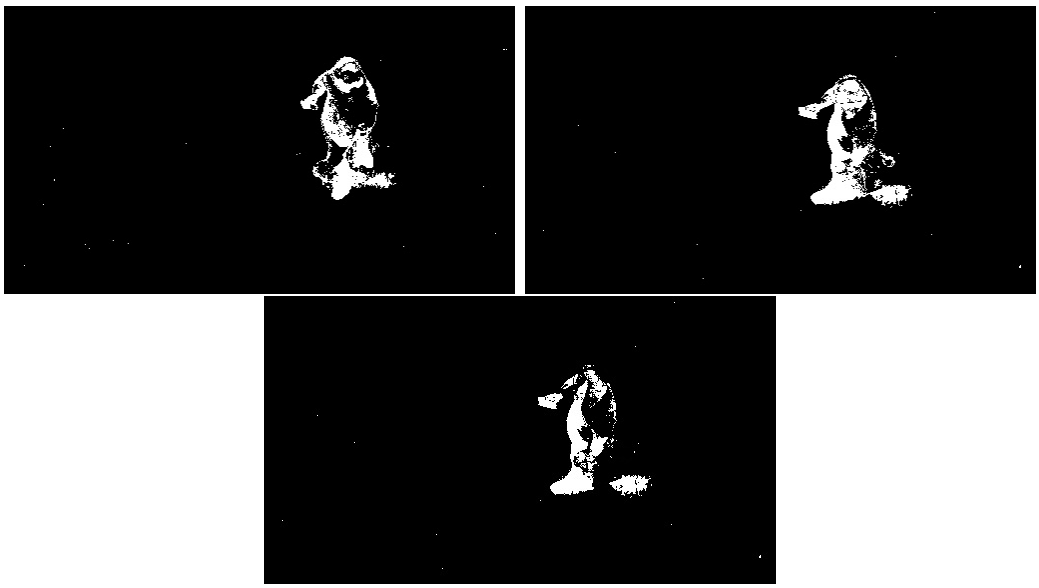

Figure 3.7: Background Subtraction performed on frames and subtraction between the frames

Figure 3.7.a and 3.7.b displays the masked frames generated. Figure 3.7c displays the difference between the two frames. From this difference, the activity A is calculated and compared with the threshold. The background subtraction is also evaluated with CUDA capabilities.

3.5 Object Classification

If a frame from the Frame Differencing module is determined to have sufficient activity, it is sent to the Object Classification module for further processing, assuming resources are available. While calculating activity assists in determining which streams need to be processed, it is also critical to understand the cause of the motion or activity within that frame. The object classification model is used to identify specific objects of interest and only process frames that contain these object classes.

While working on the classification model, the dataset is used a single unit for analytic and prediction purposes, in this case, the images from the COCO dataset introduced in section 2.4 are used to train the classification models to be implemented later in the pipeline. The dataset is divided in to 2 classes namely 'Humans' and 'NoHumans'. The dataset is further prepared into subsets for training, validation and testing. Before feeding the data in to the model, it is preprocessed by resizing the images to 128*128 pixels, while also rescaling the pixels within the image from values between 0 - 255 to relative values between 0 - 1.

To obtain the prediction values for each class, the model used, identified as the base model ahead, is a simple classification architecture consisting of three convolutional layers, three Pooling layers, and a densely Connected layer. Over the input data, a 3x3 kernel is used, and the Rectified linear activation function (relu) is applied to the output of each convolutional action. The maxpooling operation is carried out with a 2x2 sample and a stride of 2. The frequency of the filters is increased from 32 to 64 while the data shrinks as it passes through the layers.The Dense layers extract features from the data to be classified by the model, and the flattened layer changes the shape of the data so that it can be fed to the 64-node dense layer, which is followed by the final output layer of two neurons representing each class. Listing 3.2 depicts the model's architecture. Before training, the model is compiled with the necessary settings, such as using the SparseCategoricalCrossentropy as the loss function to measure how accurate the model is during training. The Adam Optimizer is also enabled in the model. Training the model takes a series of steps that involves feeding the data to the model, associating the images and the labels, and making predictions about the test set and finally verifying that the predictions match the labels.

Layer (type)	Output Shape	Param #
sequential (Sequential)	(None, 128, 128, 3)	0

```
conv2d (Conv2D)              (None, 124, 124, 32)      2432

max_pooling2d (MaxPooling2D  (None, 62, 62, 32)        0
)

conv2d_1 (Conv2D)            (None, 60, 60, 32)        9248

max_pooling2d_1 (MaxPooling  (None, 30, 30, 32)        0
2D)

conv2d_2 (Conv2D)            (None, 30, 30, 64)        18496

max_pooling2d_2 (MaxPooling  (None, 15, 15, 64)        0
2D)

dropout (Dropout)            (None, 15, 15, 64)        0

flatten (Flatten)            (None, 14400)             0

dense (Dense)                (None, 128)               1843328

dense_1 (Dense)              (None, 2)                 258

=================================================================
Total params: 1,873,762
Trainable params: 1,873,762
Non-trainable params: 0
_________________________________________________________________
```

Listing 3.2: Model summary for the classification model

3.5.1 Expanding the Dataset

The dataset used for the object classification models is further expanded to include images from the Volvo factory setup, which were combined and classified as Human and No-Human class categories, examples of which are shown in Figure 3.8.

Figure 3.8: Images from the dataset of ceiling mounted camera angle at the Volvo factory

3.5.2 MobileNetV2 Model

While the dataset contained images of varying sizes due to their origins, the images were resized and scaled, with each pixel converted to a float32 value between 0 and 1. These preprocessing techniques are applied to all images in the dataset. The initial fully convolution layer with 32 filters in the MobileNet V2 [41][42] CNN architecture is followed by 19 residual bottleneck layers. It was created by Google and trained on over 1.4 million images from 1000 different classes. Because a separate classifier is used, only the convolutional base of this model is used for the transfer learning use case. The $includetop = True$ parameter is specified after loading the model to include the top convolutional layers. Listing 3.3 highlights the summary of the model.

Retraining the model required retraining of all weights and biases, as well as approximately 2,257,187 parameters. Since these weights had already been defined and set, and they had been used in the classification of 1000 classes, this seemed unnecessary. By freezing the base, the trainable attributes of the layer are disabled. The resulting trainable parameters are shown to be zero rather than the previous value of 2,257,984. With the addition of a global average pooling layer and a prediction layer, the output from the frozen layers is then classified as Human or NoHuman. The total number of parameters is 2,260,546, with only 2562 of these being trainable.

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_160 (Functional)	(None, 5, 5, 1280)	2257984
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense (Dense)	(None, 2)	2562

Total params: 2,260,546
Trainable params: 2,562
Non-trainable params: 2,257,984

Listing 3.3: Code snippet showing the added classification layers

4

Results

This chapter presents the results of evaluating the performance of the various parts of the filtration pipeline. The primary metrics considered in the evaluation are latency and accuracy. Several optimization techniques were implemented and tested to compare the performance gains or losses obtained with these optimization techniques. Finally, the performance of the filtration pipeline in a multi-stream environment was assessed.

4.1 Streaming

An initial evaluation was carried out to estimate the number of streams that the GPU's hardware decoder could handle. The experiment used two videos with different frame rates. The first video had a frame rate of 30 frames per second, while the second video had a frame rate of 10 frames per second. Table 4.1 shows the decoder usage for each video sample streamed to the pipeline from multiple input streams.

FPS	No. of streams	Decoder %	Estimated no. of streams
30	4	11	30 - 50
	8	23	
	10	26	
	15	31	
	20	35	
10	4	5	70 - 90
	8	8	
	10	10	
	15	16	
	20	22	

Table 4.1: Table of results depicting the percentage usage of the hardware decoder available on the GPU

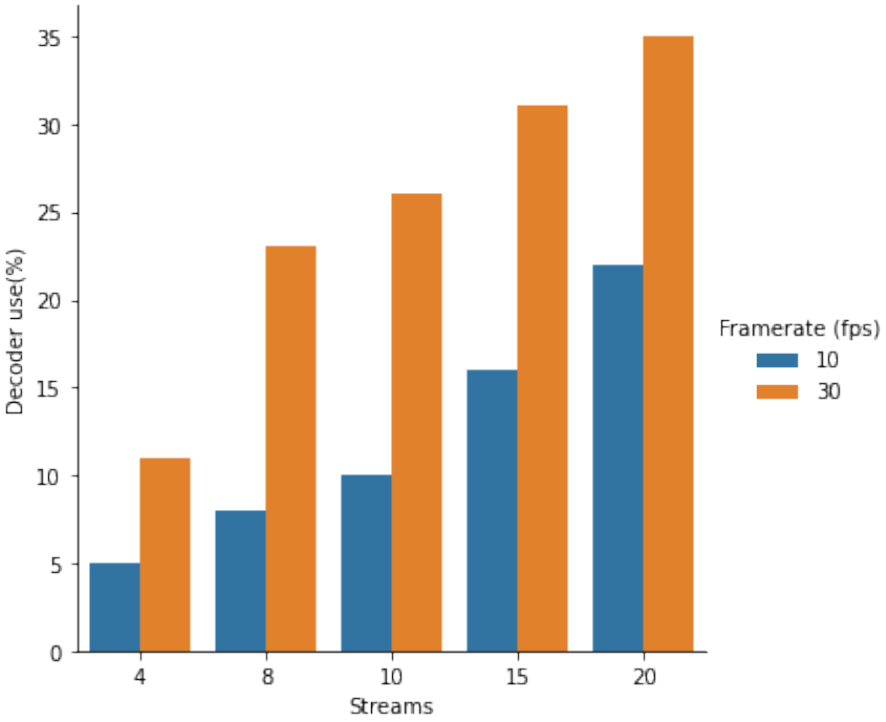

Figure 4.1: Graph comparing the difference in decoder usage for video inputs of a)10fps and b)30fps

While the number of streams does not directly increase the decoder usage, the rate at which frames are passed does, as shown in Figure 4.1. The 30 FPS video input results in an 11% utilization for 4 streams, increasing to 35% utilization when receiving frames from 20 streams. Based on these findings, the estimated number of streams that can be decoded on the Tesla V100 Nvidia GPU for a 30FPS input is around 30-40 streams.

Furthermore, the results for 10 FPS indicate the ability to decode more streams. The capacity to decode 60-90 streams is estimated based on results indicating a use of 5% for 4 streams up to 22% for 20 streams. Despite these encouraging figures, the number of streams handled is significantly lower, owing primarily to the frame differencing and classification modules' performance.

4.2 Frame Differencing

The results of the frame differencing section were obtained by running frames from a test video with 2405 frames through the module. The Frame Subtraction method and the BackgrounSubtraction method MOG2() were both tested. To begin, the performance of each method was compared based on latency while running them on the CPU.

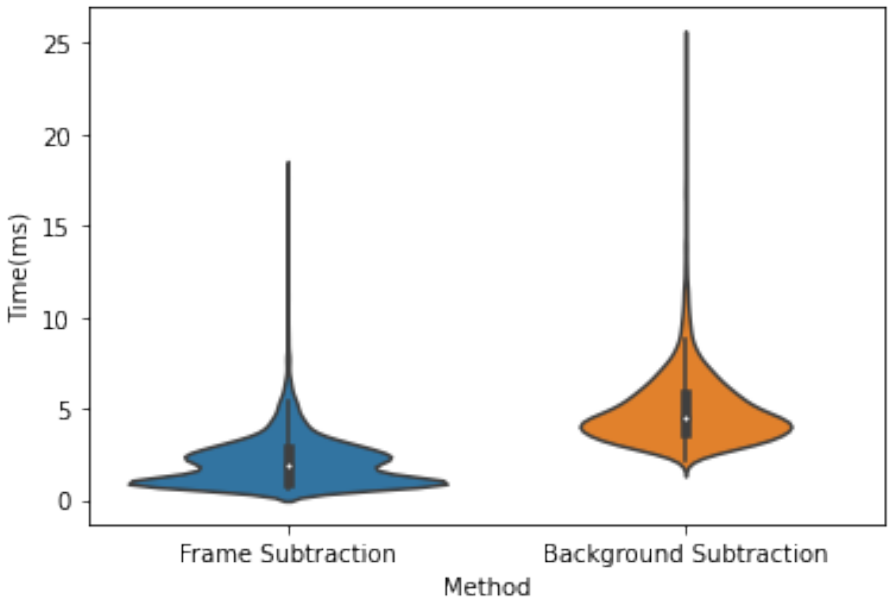

Figure 4.2: Graph depicting the latency distribution for a) Frame Subtraction and b) Background Subtraction running on the CPU

According to the graph in Figure 4.2, the frame subtraction method has lower latency than the MOG2() background subtraction method. In comparison to the MOG2() method's 5.00ms average latency, the frame subtraction method averages around 2.11ms per image.

4.3 Object Classification

We compare the base model to an established model to assess how different object classifiers perform in identifying humans on a set of images that are representative of the factory floor where this system will be used. We chose the MobileNetV2 model for this task because it is a lightweight DNN designed for mobile and embedded vision applications[42]. We compare the accuracy and latency of these two models.

The base model was trained on a dataset of over 20,000 images, from the COCO dataset. The model was trained over 20 epochs with a batch size of 30 frames. The

training yielded a training accuracy of 97.75 percent and a validation accuracy of 99.72 percent. Figure 4.3 depicts the accuracy and loss graph over 20 epochs. On a set of 79 test images with an average latency of 41.61ms, the model predicted with 85% accuracy.

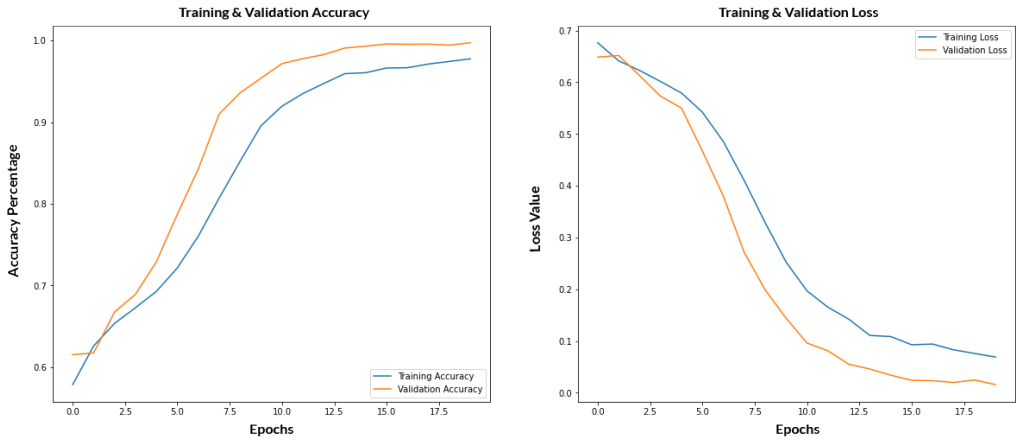

Figure 4.3: Graph depicting the training loss and validation accuracy of the custom model across 20 epochs.

The model was then trained on the image dataset from the Volvo factory introduced in section and the accuracy and loss graphs from this process are shown in Figure 4.4.

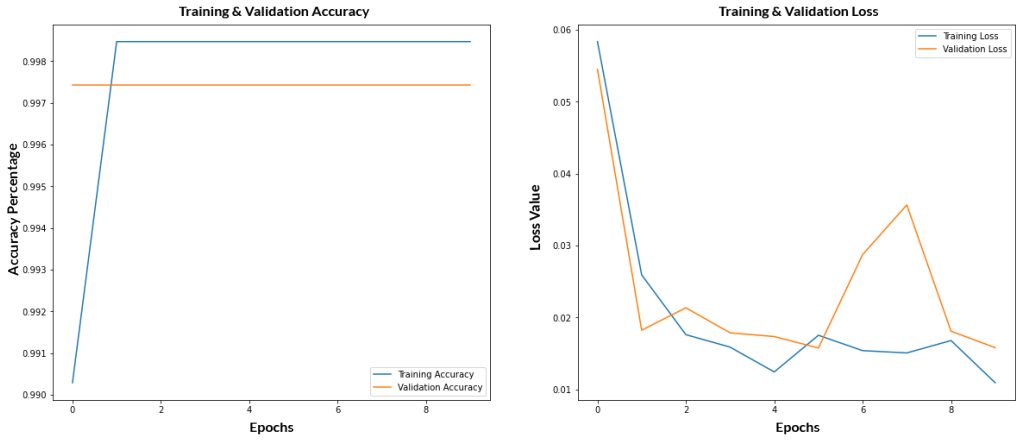

Figure 4.4: Graph depicting the training loss and validation accuracy of the custom model on training it with the new dataset across 10 epochs.

The graph in Figure 4.4 indicate an early peak in the training and validation accuracy, pointing towards little or no learning, due to similarities within them. While the training and validation accuracy were initially high after training with the COCO datset, the model peaks at around 99.85% training accuracy and 99.74% validation accuracy. Reevaluating the model and datasets would aid in optimizing the model's

learning and prediction abilities.

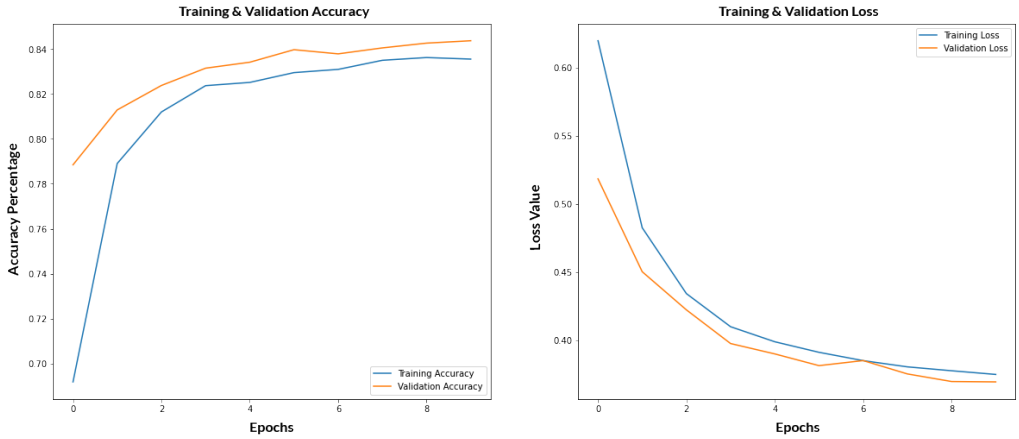

Figure 4.5: Graph depicting MobileNetV2 training loss and validation accuracy.

The MobileNetV2 model was trained on over 23458 images from a mixed dataset consisting of images from the COCO dataset and images from the Volvo dataset over 10 epochs in batches of 32, yielding a training accuracy of 83% and a validation accuracy of 84%. The model runs an inference image in about 46.4638 milliseconds on average. Figure 4.5 depicts the loss and accuracy graphs. When run on the test dataset, the MobileNetV2 model only achieves 49% accuracy.

When comparing latency for both models, the graph in Figure 4.6 shows a lower average of 41.205ms compared to MobileNetV2's 46.463ms, as well as more consistent inference times compared to MobileNetV2.

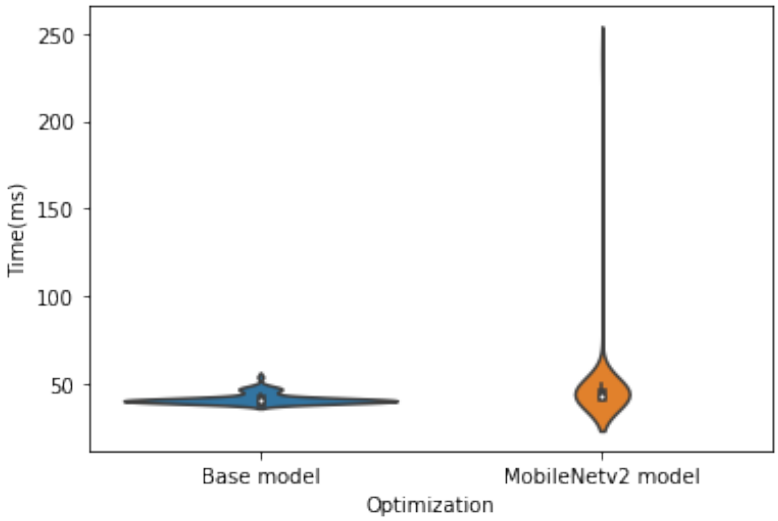

Figure 4.6: Graph depicting the latency distribution for a) base model and b) mobilenetv2

4.4 Optimization

4.4.1 Frame Differencing with CUDA

As mentioned in section 2.2.1, the openCV CUDA enabled functions provide significant performance gains[11]. This module's methods were used to generate the latency distribution graph shown in figure 4.7. The various methods were tested on 2405 images to see how they performed in terms of latency. Frame Subtraction has an average latency of 1.209ms per image, while Background Subtraction has an average latency of around 1.559ms. The observed latency improved by about 42% for Frame Subtraction and 69% for Background Subtraction.

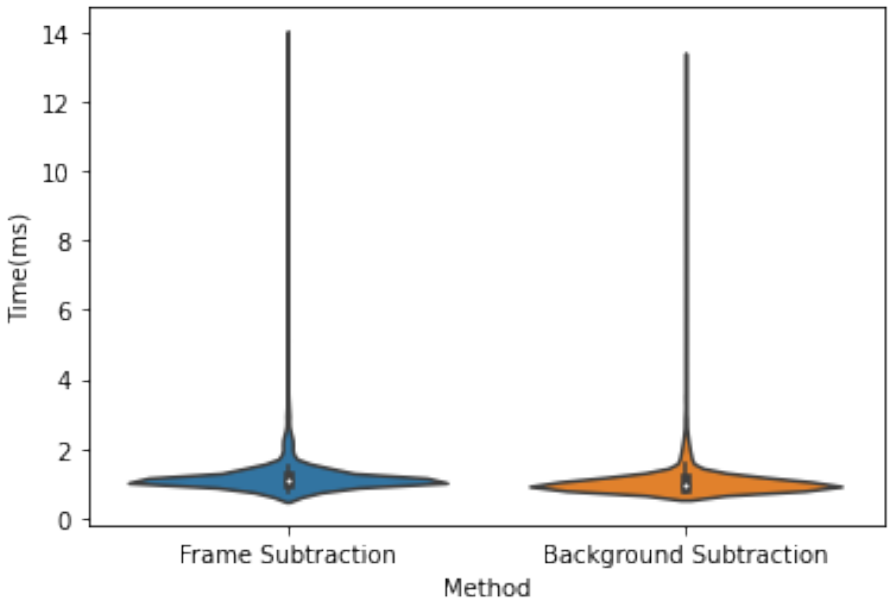

Figure 4.7: Graph depicting the latency distribution for a) Frame Subtraction and b) Background Subtraction running on the GPU using openCV CUDA module

4.4.2 Optimizing Inference Time with TensorRT

TensorRT is used to optimize the inference task, with weights in FP32 format. When running inference on the test video, a faster inference time for the base model is observed, as shown in Figure 4.8, where the inference time for the non-optimized base model is compared. The observed performance gain of around 91%, with an average latency of 3.444ms per image.

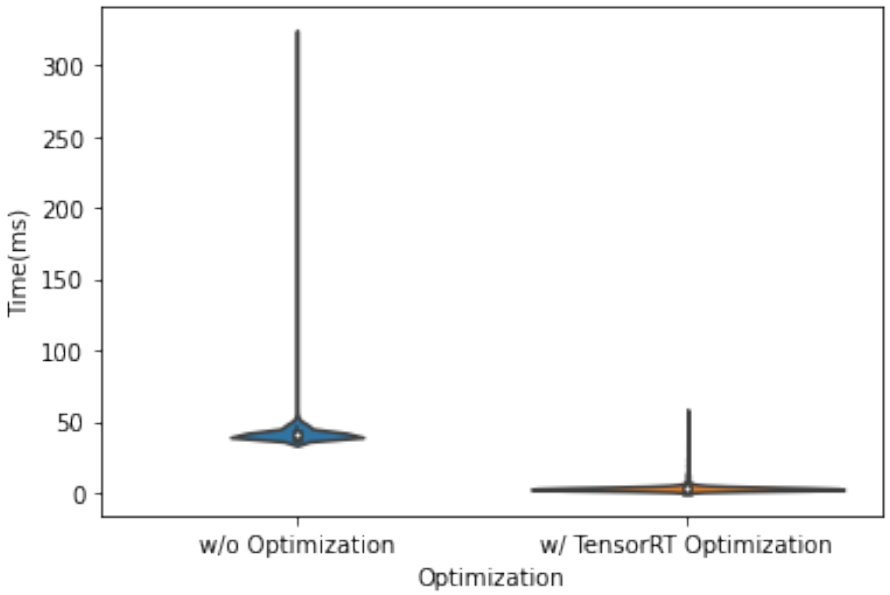

Figure 4.8: Graph depicting the latency distribution for a) Base model without any optimization and b) Base model optimized with tensorRT to improve inference time.

4.5 Pipeline performance

Finally, after testing the pipeline with the chosen methods and optimizing it, an optimistic allocation of 7 streams is assumed. The results in Table 4.2 and the graph in Figure 4.10 depict the runtimes observed while running this pipeline with up to 16 simultaneous streams. The batch frame execution threshold was set at 100ms for a 10 frames per second video input to ensure that no frames were dropped due to a delay in processing the current batch of frames. The results in table 4.2 show that we can run up to 13 streams with the pipeline while keeping the execution time for each batch of frames below the threshold, indicating support for up to 6 more oversubscribed frames.

No. of Frames	Frame differencing (ms)	Classification (ms)	Pipeline Runtime (ms)
8	20.8618	13.7075	69.316
9	23.2267	20.3433	82.8963
10	31.019	16.5219	83.3594
11	32.235	19.8546	93.8209
12	30.4576	20.3152	102.6699
13	29.1474	19.917	92.4365
14	40.3632	17.1825	123.8814
15	32.8545	25.9988	115.7212
16	51.2613	27.8707	172.5544

Table 4.2: The frame difference classification and total run time of the pipeline while handling frames from multiple streams.

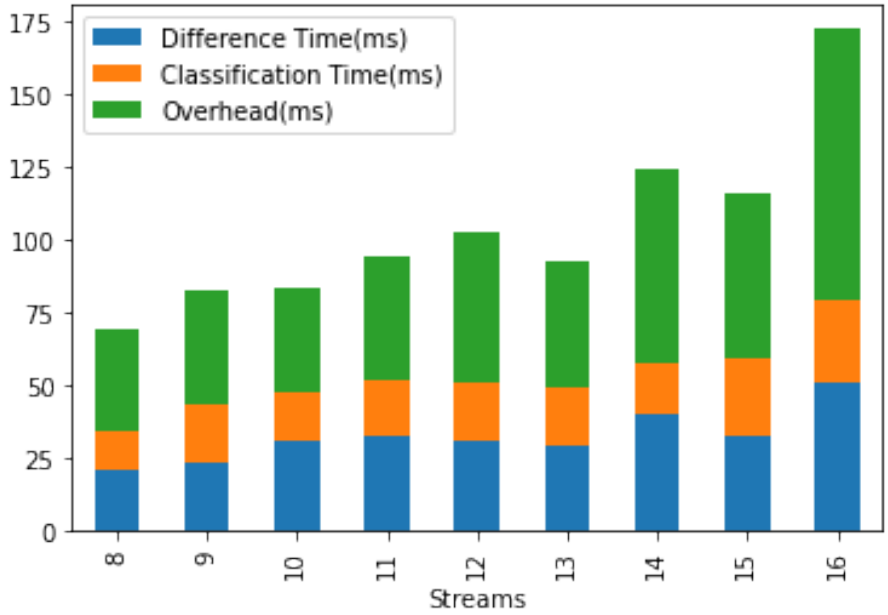

Figure 4.9: Graph comparing the overall run time of the pipeline handling frames from multiple streams.

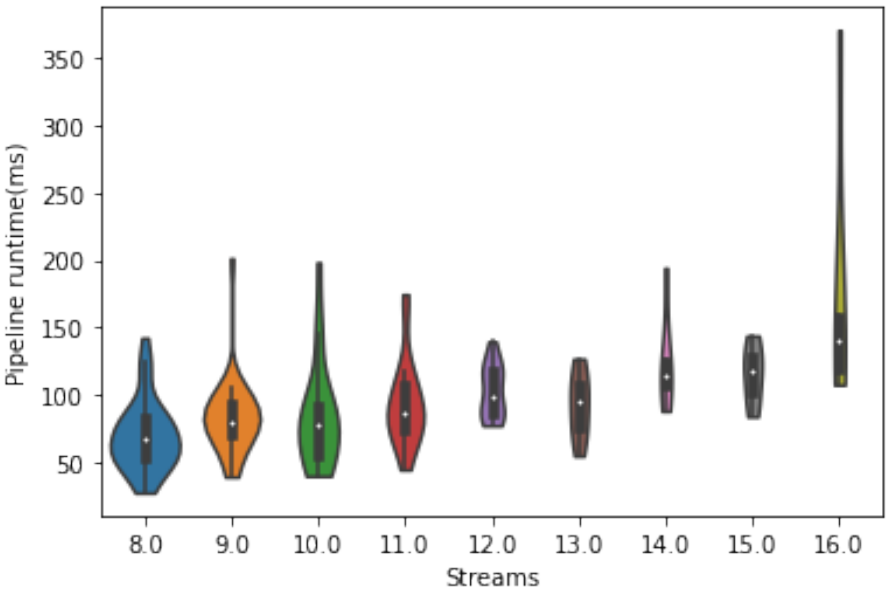

Figure 4.10: Graph depicting the latency distribution for the pipeline handling multiple streams.

5

Conclusion

We investigate the use of a cascaded approach to implementing a video filtration pipeline that employs a rank-based scheduling algorithm to effectively assign resources to relevant streams in this study. On the basis of accuracy and latency, computer vision techniques such as frame subtraction and background subtraction, as well as classification models, were compared and evaluated. The pipeline was also tested for performance while accommodating oversubscription, which occurs when more streams are assigned to process than resources are available.

It was discovered that the pipeline could handle 6 streams in excess of the available resource limit. Each pre-processing step in the pipeline was optimized to run with as low latency as possible. While accuracy was not improved, the current test scenarios produced sufficient results to assume effective performance.

To improve accuracy, the performance of the classification model can be reassessed in relation to the quality of the dataset on which it was trained as well as assessing the layers that make up the model. While the current test scenarios were idealistic, testing the effect of changing lighting conditions becomes an important task. Further analyzing the pipeline on a test setup similar to its real-world counterpart will provide opportunities to correct any overheads or latency observed during the test runs on a virtual system. Another important aspect in such systems is the prediction of False Negative results, which could be detrimental in high-risk situations encountered by the system.

Overall, this study provides insight into the performance of a fitration pipeline with promising results that can be improved further by optimizing various aspects of the real-time implementation that is part of a safety-critical system. While the filtration process is not a stand-alone component of the safety-critical system, it is vital in ensuring proper data analysis in order to perform critical tasks.

Bibliography

[1] *Ldv capital insights: 45 billion cameras by 2022 fuel business opportunities*, https://www.ldv.co/insights/2017, (Accessed on 02/16/2022).

[2] H. Ahn and H.-J. Cho, "Research of multi-object detection and tracking using machine learning based on knowledge for video surveillance system," *Personal and Ubiquitous Computing*, vol. 26, no. 2, pp. 385–394, Apr. 2022, ISSN: 1617-4917. DOI: 10.1007/s00779-019-01296-z. [Online]. Available: https://doi.org/10.1007/s00779-019-01296-z.

[3] E. Nishani and B. Çiço, "Computer vision approaches based on deep learning and neural networks: Deep neural networks for video analysis of human pose estimation," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, 2017, pp. 1–4. DOI: 10.1109/MECO.2017.7977207.

[4] A. Ben Sada, M. A. Bouras, J. Ma, H. Runhe, and H. Ning, "A distributed video analytics architecture based on edge-computing and federated learning," in *2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, 2019, pp. 215–220. DOI: 10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00047.

[5] N. Aung, W. Zhang, S. Dhelim, and Y. Ai, "Accident prediction system based on hidden markov model for vehicular ad-hoc network in urban environments," *Information*, vol. 9, no. 12, 2018, ISSN: 2078-2489. DOI: 10.3390/info9120311. [Online]. Available: https://www.mdpi.com/2078-2489/9/12/311.

[6] A. Gupta, A. Anpalagan, L. Guan, and A. S. Khwaja, "Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues," *Array*, vol. 10, p. 100 057, 2021, ISSN: 2590-0056. DOI: https://doi.org/10.1016/j.array.2021.100057. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2590005621000059.

[7] J. Xiong, E.-L. Hsiang, Z. He, T. Zhan, and S.-T. Wu, "Augmented reality and virtual reality displays: Emerging technologies and future perspectives," *Light: Science & Applications*, vol. 10, no. 1, p. 216, Oct. 2021, ISSN: 2047-7538. DOI: 10.1038/s41377-021-00658-8. [Online]. Available: https://doi.org/10.1038/s41377-021-00658-8.

[8] G. Ananthanarayanan, P. Bahl, P. Bodk, *et al.*, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[9] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali, "Reducto: On-camera filtering for resource-efficient real-time video analyt-

ics," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 359–376, ISBN: 9781450379557. DOI: 10.1145/3387514.3405874. [Online]. Available: https://doi.org/10.1145/3387514.3405874.

[10] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: Optimizing neural network queries over video at scale," *arXiv preprint arXiv:1703.02529*, 2017.

[11] *Realtime computer vision with opencv - acm queue*, https://queue.acm.org/detail.cfm?id=2206309, (Accessed on 06/11/2022).

[12] L. Alzubaidi, J. Zhang, A. J. Humaidi, *et al.*, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, no. 1, p. 53, Mar. 2021, ISSN: 2196-1115. DOI: 10.1186/s40537-021-00444-8. [Online]. Available: https://doi.org/10.1186/s40537-021-00444-8.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385.

[15] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[16] *Pytorch vs tensorflow: A head-to-head comparison - viso.ai*, https://viso.ai/deep-learning/pytorch-vs-tensorflow/?msclkid=c0bb2b97d05611eca52f9f20ffe593 (Accessed on 06/11/2022).

[17] E. Jeong, J. Kim, and S. Ha, "Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards," *ACM Trans. Embed. Comput. Syst.*, Dec. 2022, Just Accepted, ISSN: 1539-9087. DOI: 10.1145/3508391. [Online]. Available: https://doi.org/10.1145/3508391.

[18] *Opencv: Background subtraction*, https://docs.opencv.org/4.x/d8/d38/tutorial_bgsegm_bg_subtraction.html?msclkid=170a7b79d05411eca66c46a8464ffb38, (Accessed on 06/11/2022).

[19] *Opencv: How to use background subtraction methods*, https://docs.opencv.org/4.x/d1/dc5/tutorial_background_subtraction.html?msclkid=e9d93caed11811eca7b0d6544f737958, (Accessed on 06/11/2022).

[20] *About - opencv*, https://opencv.org/about/?msclkid=c4bec039d04111eca01e491a539899b3 (Accessed on 06/11/2022).

[21] J. A. Nichols, H. W. Herbert Chan, and M. A. B. Baker, "Machine learning: Applications of artificial intelligence to imaging and diagnosis," *Biophysical Reviews*, vol. 11, no. 1, pp. 111–118, Feb. 2019, ISSN: 1867-2469. DOI: 10.1007/s12551-018-0449-9. [Online]. Available: https://doi.org/10.1007/s12551-018-0449-9.

[22] T. Lin, M. Maire, S. J. Belongie, *et al.*, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014. arXiv: 1405.0312. [Online]. Available: http://arxiv.org/abs/1405.0312.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[24] M. A. Bamakhrama, J. T. Zhai, H. Nikolov, and T. Stefanov, "A methodology for automated design of hard-real-time embedded streaming systems," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 941–946. DOI: 10.1109/DATE.2012.6176632.

[25] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015, ISSN: 0163-5980. DOI: 10.1145/2723872.2723882. [Online]. Available: https://doi.org/10.1145/2723872.2723882.

[26] *What is kubernetes? | kubernetes*, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, (Accessed on 06/11/2022).

[27] *Kubernetes components | kubernetes*, https://kubernetes.io/docs/concepts/overview/components/, (Accessed on 06/11/2022).

[28] *Nvidia gpu operator: Simplifying gpu management in kubernetes | nvidia technical blog*, https://developer.nvidia.com/blog/nvidia-gpu-operator-simplifying-gpu-management-in-kubernetes/?msclkid=64c4c1f0d06011eca1efc1e112f8 (Accessed on 06/11/2022).

[29] *What is gstreamer?* https://gstreamer.freedesktop.org/documentation/application-development/introduction/gstreamer.html?gi-language=c, (Accessed on 06/11/2022).

[30] *Gstreamer: Open source multimedia framework*, https://gstreamer.freedesktop.org/, (Accessed on 06/11/2022).

[31] *Elements*, https://gstreamer.freedesktop.org/documentation/application-development/basics/elements.html?gi-language=python, (Accessed on 06/11/2022).

[32] *Movenet: Ultra fast and accurate pose detection model. ă/ă tensorflow hub*, https://www.tensorflow.org/hub/tutorials/movenet, (Accessed on 08/31/2022).

[33] *Pose estimation and classification on edge devices with movenet and tensorflow lite the tensorflow blog*, https://blog.tensorflow.org/2021/08/pose-estimation-and-classification-on-edge-devices-with-MoveNet-and-TensorFlow-Lite.html, (Accessed on 08/31/2022).

[34] *Rtpbin*, https://gstreamer.freedesktop.org/documentation/rtpmanager/rtpbin.html?gi-language=python#rtpbin-page, (Accessed on 06/11/2022).

[35] *Nvidia deepstream sdk developer guide deepstream 6.1 release documentation*, https://docs.nvidia.com/metropolis/deepstream/dev-guide/index.html, (Accessed on 06/11/2022).

[36] *Cuda array interface (version 2) numba 0.52.0.dev0+274.g626b40e-py3.7-linux-x86_64.egg documentation*, https://numba.pydata.org/numba-doc/dev/cuda/cuda_array_interface.html, (Accessed on 06/11/2022).

[37] P. KaewTraKulPong and R. Bowden, "An improved adaptive background mixture model for real-time tracking with shadow detection," in *Video-Based Surveillance Systems: Computer Vision and Distributed Processing*, P. Remagnino, G. A. Jones, N. Paragios, and C. S. Regazzoni, Eds. Boston, MA: Springer US, 2002, pp. 135–144, ISBN: 978-1-4615-0913-4. DOI: 10.1007/978-1-4615-0913-4_11. [Online]. Available: https://doi.org/10.1007/978-1-4615-0913-4_11.

[38] A. Godbehere, A. Matsukawa, and K. Goldberg, "Visual tracking of human visitors under variable-lighting conditions for a responsive audio art installation," Jun. 2012, pp. 4305–4312, ISBN: 978-1-4577-1095-7. DOI: 10.1109/ACC.2012.6315174.

[39] Z. Zivkovic and F. van der Heijden, "Efficient adaptive density estimation per image pixel for the task of background subtraction," *Pattern Recognition Letters*, vol. 27, no. 7, pp. 773–780, 2006, ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2005.11.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167865505003521.

[40] Z. Zivkovic, "Improved adaptive gaussian mixture model for background subtraction," in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 2, 2004, 28–31 Vol.2. DOI: 10.1109/ICPR.2004.1333992.

[41] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018. arXiv: 1801.04381. [Online]. Available: http://arxiv.org/abs/1801.04381.

[42] A. G. Howard, M. Zhu, B. Chen, *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. arXiv: 1704.04861. [Online]. Available: http://arxiv.org/abs/1704.04861.

A

Appendix 1

This chapter displays images from the pipeline and a sequence with an increasing degree of activity while performing the frame subtraction method.

Figure A.1: Detailed Gstreamer pipeline for a single stream

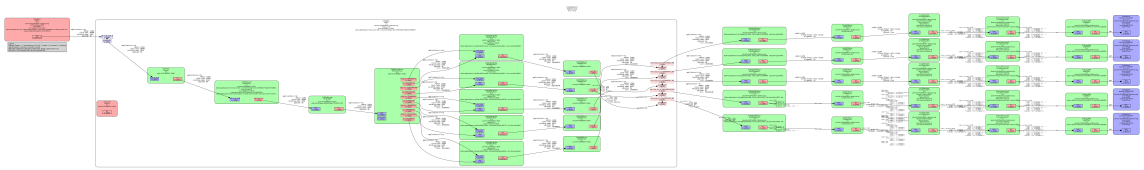

Figure A.2: Detailed Gstreamer pipeline for a multiple streams

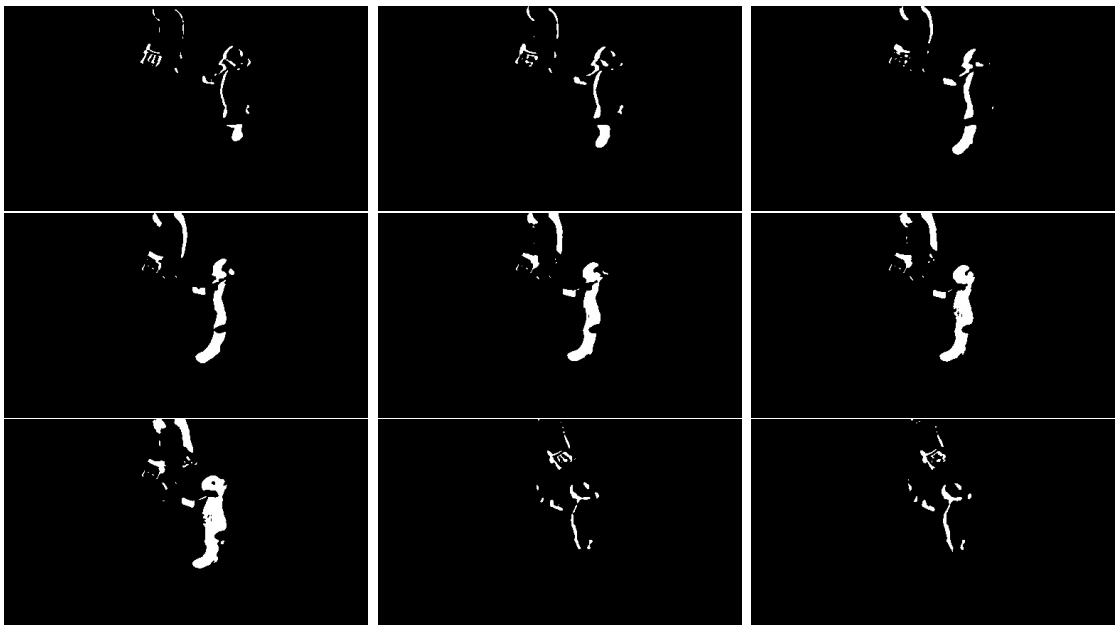

Figure A.3: Sequence of images resulting from the frame subtraction method highlighting use of a previous frame that triggers the threshold