MASTER

Scheduler for dynamic processing of a video pipeline on an embedded GPU platform

Sivamurugan, Viknesh B.

Award date:
2020

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems Group

Scheduler for dynamic processing of a video pipeline on an embedded GPU platform

Master Thesis

V.B. SIVAMURUGAN
Student ID: 1336975

University Supervisor:
Dr. Ir. Sander Stuijk (Electronic Systems, TU/e)
Mentor:
Ph.D. Candidate Berk Ulker (Electronic Systems, TU/e)
Company Supervisor:
Dr. Ir. Rob Wijnhoven (C.T.O, ViNotion B.V)

External Committee:
Dr. Ir. Gijs Dubbelman (Video Coding & Architecture, TU/e)
Dr. Ir. Marc Geilen (Electronic Systems, TU/e)

First version

Eindhoven, November 2020

Abstract

In the past few years, there has been a rapid growth in the domain of video analytics. It has attracted increasing interest from both industry and the academic world. Thanks to the enormous advances made in deep learning, video analytics has introduced the automation of tasks that were once the exclusive expertise of humans. From smart cities to security controls in hospitals and airports to people tracking for retail and shopping centers, the field of video analytics enables processes that are simultaneously more effective and less tedious for humans, and less expensive for companies.

The next-generation video surveillance system will be a networked, intelligent, multi-camera cooperative system with integrated situation awareness of dynamic and complex scenes. A powerful hardware platform is necessary for a real-time performance of such systems. However, the computational resources on the platform are limited and heterogeneous in nature. Inefficient mapping of tasks on computation resources stalls the system. Executing concurrent tasks leads to severe resource constraints and increased timing. An increase in execution time is not desirable as it results in a low throughput performance. There arises a need for a software scheduler that can allocate tasks efficiently to the platform while maximising the throughput and GPU utilization.

A conventional static scheduler has a constant processing rate and has been found to have a low GPU utilization. For instance, it is not possible to adapt the processing rate on a stream where no or less execution of tasks is sufficient. Dynamic processing is a technique that adapts the rate of computation. Each stream can be processed based on the activity level monitored in every stream. However, the dynamic scheduling must ensure fairness during task allocation from each stream as it has a larger impact in determining the system accuracy. A weighted mean selection algorithm is proposed for this purpose.

In this work, we propose a scheduler for the dynamic processing of multiple video streams. The scheduler efficiently allocate tasks based on GPU capacity and activity level in video streams. The scheduler is intended to be operable for two types of application modes. The two performance modes considered are maximum throughput mode and minimum latency mode. We quantify the performance of the static and dynamic scheduler by accuracy evaluation on UA-DETRAC data set. Through the proposed design flow, the application delivered a higher throughput and better accuracy performance than a standard static scheduler.

Acknowledgement

I take this opportunity to thank everyone who had helped me for the successful completion of the project. This project gave me great experience, and I am grateful for the wonderful opportunity to work in collaboration with ViNotion and TU/e.

I would like to thank my supervisor, Dr. Ir. Sander Stuijk for his guidance and valuable feedback during the course of the project. I would also like to thank my mentor, PhD student Berk Ulker for his guidance and positive mindset towards addressing multiple issues during my thesis work. Dr. Ir. Rob Wijnhoven, ViNotion supervisor, I would like to thank you for your efforts to make this project possible and your constructive feedback on any topic related to my work and your willingness to sit down and discuss anything at any time, I sincerely appreciate your guidance. I would also like to thank my other thesis committee members Dr. Ir. Gijs Dubbelman and Dr. Ir. Marc Geilen for their feedback during preparation discussions and time to review my work.

I want to express gratitude to my friends and colleagues for their support. Finally, I would like to give my thanks and love to my parents for their unconditional support and care during the whole master program and thesis. I have made many strides to develop my academic career in the past nine months and learned a great deal from every colleague and supervisor, and I am especially thankful for the gained knowledge in deep learning research domain being cross disciplinary subject to my program.

Contents

Chapter 1

Introduction

Modern-day video analytics allow users to deploy multiple cameras to automatically detect and analyze spatial and temporal objects. Video Content Analysis (VCA) is a branch of computer vision that involves various deep learning-based functionalities such as object detection and tracking. Advancements in VCA technology enabled these functionalities to be implemented as software algorithms on general-purpose machines or specialised hardware platforms. This technical accomplishment is used in a wide range of domains to decrease human effort, including automotive, transport, health-care, home automation, safety, and security.

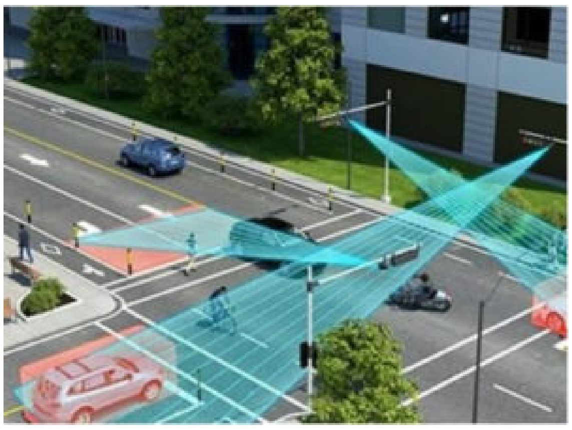
The specialised hardware platforms used for processing deep learning models are heterogeneous systems that contain several processing units. They are either computation or memory bounded. The computation time of a complex deep learning network model is expensive on such platforms [1]. In a multi-camera system, expensive computation profiles affect the whole system to maintain a high throughput and latency performance. Therefore, we see a challenge in obtaining a good system performance with multiple video streams on the same hardware platform.

There are two approaches to address the above challenge. An elementary approach is to adapt the design of the neural network model to suit a specific platform. However, the adaptation comes with a trade-off with the accuracy of the model. Another interesting approach is to adopt an intelligent schedule for the processing of neural networks on the platform. This method investigates the efficient mapping and scheduling of tasks on a heterogeneous platform for concurrent processing. Both approaches help to alleviate the computational problems with limited resources. This project explores the latter approach of finding smart ways to process multiple video streams on the same platform.

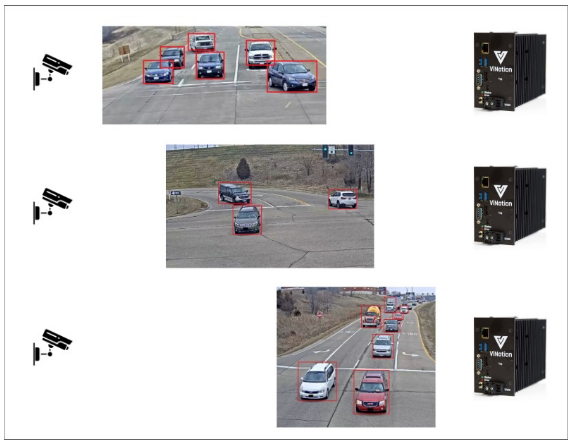
1.1 Problem setting

Multi-camera systems are essential in many applications such as surveillance, automotive safety, robot navigation, industrial manufacturing, and sports. The view of a single camera is finite and limited by scene structures. To monitor a wide area, such as monitoring a traffic intersection or counting people in a large train station, video streams from multiple cameras have to be used. Deploying hardware for each camera unit for processing occupies more space and increases the business investment. This case is illustrated in Figure 1.1.

The scenario considered is traffic surveillance on a T-intersection for vehicle tracking and analysis. The T-intersection consists of three road lanes to be monitored. Therefore, three cameras are required to monitor the scene as depicted in Figure 1.1a. The traditional method of installing three independent hardware units for each camera is illustrated in Figure 1.1b. This solution is expensive and unfit for mass production.



(a) Surveillance in a T-intersection from [2]



(b) Existing multi-camera solution

Figure 1.1: Multi-camera traffic surveillance

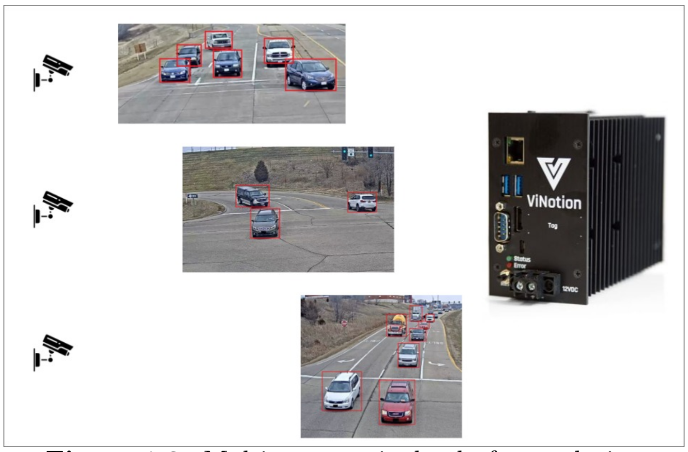


Figure 1.2: Multi-camera single platform solution

The individual hardware units are not often utilized to their fullest potential. Therefore, there is an opportunity to merge computations from multiple cameras on to a single hardware unit as represented in Figure 1.2. This solution is efficient, requires cheap investment and maintenance. In such a solution, a single hardware is responsible to process frames from multiple video streams. The computational demand on the single hardware increases as the number of video streams increases. Overloading computations on a resource-constrained hardware degrades the system throughput and processing latency. Thus, the research is to find a way to manage multi-camera computations on a resource-limited platform resulting in a good system performance.

1.2 Motivation

Object detection and tracking on a video stream is a process of feature extraction and computation over successive frames. To improve this performance in a multi-camera system, the hardware can be replaced with a more powerful platform. However, using a more powerful platform incurs higher investments again. This arises a need for management of computations using the existing resources on the platform. Therefore, it is important to measure the computing capability of existing resources and perform computations below the estimation for a good system performance. Hence, this motivates the need for the system to evaluate the platform, select frames from each stream, and arrange them into an appropriate sequence for computation.

During the concurrent processing of streams in a multi-camera system, a non-deterministic behaviour may be expected in the execution time. These can be attributed to resource contention and synchronization periods. Hence, the scheduling sequence of the video pipeline is expected to be designed to minimize the variations to guarantee predictable behaviour. It is also important to build an efficient pipeline of tasks and computation streams to avoid stalling of hardware. A high application throughput is expected with an efficient pipeline and processing configuration.

The processing schedule plays a vital role in determining the system performance and accuracy. Often frames from each stream are processed at regular intervals. This technique is called fixed frame rate scheduling. Such a scheduling technique ensures that all streams make equal use of the available resources irrespective of the environment scenario [3]. However, this may also become a drawback in certain cases. A fixed frame rate scheduling does not support a flexible frame processing rate. For instance, in the T-intersection where a minor road joins major roads, a video stream from the former could be less busy than the latter. Object detection and tracking algorithm still runs at a constant rate in both streams. In such cases, lowering the processing rate of idle streams could make way for more computation on the busy streams. This results in better detection and tracking performance on busy streams. Hence, it drives the need for content-aware resource provisioning systems for analyzing video streams, where the resource management decisions are based on the content. Therefore, the goal of this project is to set up a multi-camera system with a dynamic scheduling policy based on situation awareness.

In our approach, we define the performance through a set of performance metrics. The metrics chosen are Throughput, Latency, GPU resource utilization, Average precision, and recall for object detection. The project uses an Nvidia GeForce GTX 1050 Ti GPU and Intel(R) Core(TM) i7-2600K quad-core CPU platform to set up a framework for multi-camera video processing. By the end of the project, the goal is to implement a dynamic scheduling technique for efficient processing.

1.3 Problem Statement

A multi-camera video processing system impose additional challenges to extract a high performance on the heterogeneous resource-constrained hardware. Several factors complicate the computations. First, tasks are run concurrently on the same hardware resources that leads to higher resource demand and increased execution time. Second, considering the available resource capacity, it is important to schedule tasks to maximize the combined throughput while satisfying the latency bounds. Third, given the dynamic processing rate of streams, it is important to identify a fair scheduling algorithm to select frames from each stream.

Considering these scenarios, the research aims to **design an efficient scheduler configuration for dynamic processing of the computation pipeline for multi-camera systems to achieve maximum GPU utilization**.

The contribution of this work is to set up a multi-camera system with a dynamic scheduling policy based on the priority of input video streams. The focus is to build a scheduler that satisfies the application and hardware requirements mentioned below:

- Application is expected to have real-time processing characteristics, i.e., each video stream is processed at 25 FPS
- Support for multiple stream processing despite constant/limited GPU processing power
- Throughput and GPU utilization of the application is expected to be maximum
- Processing latency of each task is expected to be minimum
- Accuracy of the multi-camera system is expected to be maximum

Some of these requirements are conflicting. There is not a single solution that maximizes throughput, minimizes latency and which can process different streams at the required rate. This project explores to study the trade-off space between these conflicting requirements.

1.4 Research Question

In this work, we present the algorithm and system implementation for dynamic processing of multiple camera systems. The novel components include scheduling tasks for maximum GPU utilization based on resource capacity estimation and run-time adaptation of the processing schedule by classifying the business level of video streams.

Some of the research questions that are answered through the project are listed below:

1. What is an efficient processing scheme to obtain maximum throughput?
2. What is an efficient processing scheme to obtain minimum task latency?
3. How do we maximise the GPU utilization?
4. How do we classify and prioritize multiple video streams?
5. How do we achieve fairness in task execution sequence?

1.5 Outline

The report is organized as follows: Chapter 2 describes the background of the existing product and challenges in multi-camera systems. Chapter 3 draws inspiration from related works on the research topic. Chapter 4 proposes the multi-camera system with dynamic processing and design procedure. Chapter 5 focuses on the implementation of the proposed system in detail. Chapter 6 details about the the profiling tool, experimentation set up and evaluation metrics. Chapter 7 discusses the observed results of the proposed system. Chapter 8 concludes and guides the direction for future work.

Chapter 2

Application background

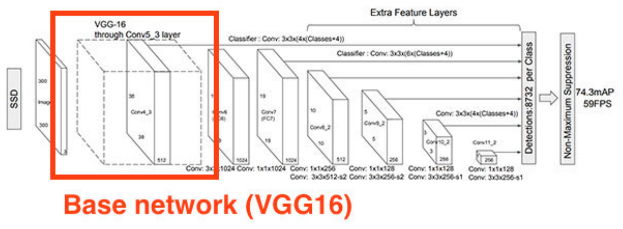
This chapter gives an insight into the base software on which the research was performed. The work was in collaboration with ViNotion B.V., a spin of the Eindhoven University of Technology. They specialize in automated video analysis using innovative techniques such as machine and deep learning [4]. One of their product ranges provides real-time traffic analysis on national and regional roads. The product is supported by an embedded hardware platform.

2.1 ViApp Software

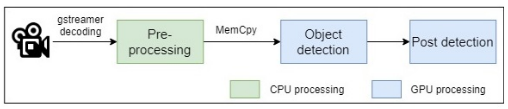
ViApp is a single-camera based software application for traffic dynamics. The software application consists of an object detection deep learning model and a tracking module to track the motion of the identified traffic participants.

2.1.1 Detection

The role of the detector is to detect and classify all traffic participants in the environment. ViNotion designed a state-of-the-art Single Shot multibox Detector (SSD) with inspiration from [5]. Their design replaced the base network of SSD by a custom ResNet-18 network for fast computation. A Non-Maximum Suppression (NMS) algorithm at the final stage removes unwanted and redundant prior boxes. The model is trained extensively on public datasets. The tasks of the detector module is described in Figure 2.1b. The pipeline consists of decoded frames pre-processed on a CPU while the inference and post-detection activities are computed on a GPU.


Extra Feature Layers
VGG-16 through Conv5_3 layer
Classifier : Conv: 3x3x(4x(Classes+4))
Classifier : Conv: 3x3x(6x(Classes+4))
SSD
Conv: 3x3x(4x(Classes+4))
Detections:8732 per Class
Non-Maximum Suppression
74.3mAP 59FPS
Base network (VGG16)

(a) SSD architecture proposed in [5]



(b) Detector pipeline

Figure 2.1: Detector

2.1.2 Tracking

The two key components of any feature tracker are accuracy and robustness. To make a balanced trade-off, a pyramidal implementation of the classical Lucas-Kanade algorithm is proposed in [6].

An iterative implementation of the Lucas-Kanade optical flow computation provides sufficient local tracking accuracy. In ViApp, the tracker obtains feature points from the detector and computes optical flow to observe the relative motion in consecutive frames. This whole process is a CPU based execution using OpenCV libraries.

2.1.3 Single Stream Pipeline

The ViApp application performs a set of tasks periodically that are captured in a pipeline in Figure 2.2. The pipeline consists of five stages. In the ingestion stage, Gstreamer, a pipeline-based multimedia framework, is used to stream video directly from a camera or pre-recorded files. At the decoder stage, the frames on the stream are decoded successively using Gstreamer plugins. During the computation phase, frames are pre-processed to make it efficient for computation. Some of the pre-processing methods include frame resizing, noise removal, colour space transformation, and Region of Interest (RoI) localization.

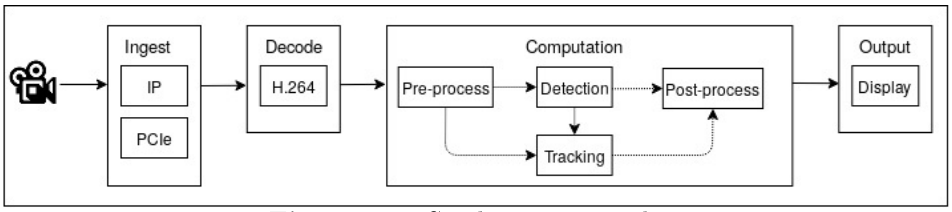


Figure 2.2: Single stream pipeline

The pre-processed frames are fed to the detector and tracker. The detector uses an object detection model to make an inference to identify traffic participants from the frames. The tracking module depends on the identified traffic participants to track in successive frames. During the post-detection task, all identified participants are enclosed in a box. To manage the amount of computation, a ViApp schedule configuration file is used to control the rate of object detection and tracking.

Computation pipeline

The computation pipeline consists of detection and tracking tasks. The frequency of execution is configurable for each frame and shall be defined as a processing schedule. The processing schedule of detector and tracker can be a fixed or dynamic scheme. For instance, a fixed processing schedule on a video stream of 25 FPS is considered and depicted in Figure 2.3. In the timeline, an upward arrow denotes an input frame arriving at every 40ms. The tasks detect and track with an execution time of T_D and T_T are scheduled in succession immediately after the frames arrive. The same sequence continues as long as the input is available. Such a fixed schedule follows a detection and tracking rate of 25 FPS since both tasks are computed on all frames.

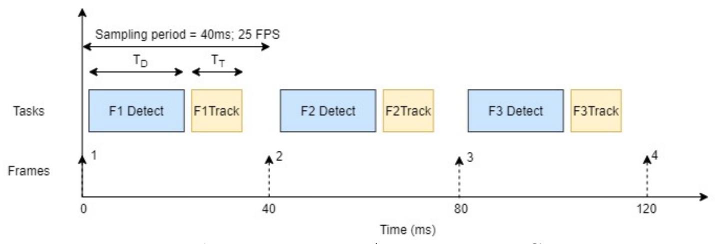
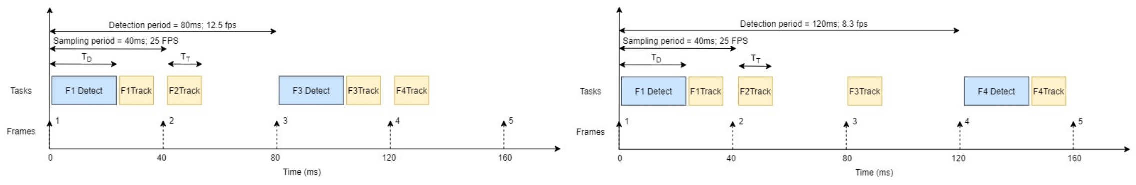


Figure 2.3: ViApp at 25 FPS

In real-time processing, tasks are expected to be computed before the next frame arrives. Mathematically, $T_D + T_T < 40$ms. However, object detection in low computing devices like Jetson executes longer than the inter-frame arrival time. In such cases, the detection is intentionally skipped from the pipeline and tracking is executed. This scheme is a popular variant of fixed scheduling where few tasks are skipped at regular intervals.

Schedules in Figure 2.4 show a strategy where frames are skipped at constant intervals in the detection network. Figure 2.4a represents a schedule with a detection rate of 12.5 FPS whose detection occurs in every two frames. The schedule in Figure 2.4b has a detection rate of 8.3 FPS and detects one in three frames. Notably, both the schedules execute the tracking task on all frames.



(a) Detection rate 12.5 FPS **(b)** Detection rate 8.3 FPS

Figure 2.4: ViApp schedule configurations

Practically, the impact of skipping a detection task on multiple frames affects the tracking quality in terms of accuracy. However, the impact is larger for busy streams and does not have a visible impact on idle streams. So, care has to be taken to schedule the detection task regularly depending upon the activity level in video streams.

2.2 Multi-stream pipeline and challenges

The video processing pipeline for multi-camera system is similar to the single camera. The difference comes from synchronization and high data rate. The most difficult problem is the high data rate. When the number of frames per second is increased many times, we can not scale CPU/GPU performance accordingly on the same hardware. Figure 2.5 illustrates this problem in the multi-stream processing on the same environment. It is observed that the computation time is greater than the inter-frame duration. As a result, the processing rate of the system decreases and concurrent execution increases the latency of individual tasks. There is a substantial challenge to complete the execution between the inter-frame duration and maintain a higher throughput.

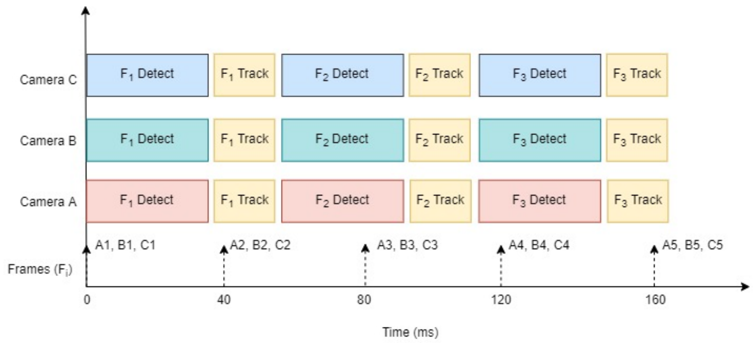


Figure 2.5: Multi-stream concurrent task scheduling

The goal of this project is to propose a scheduler for a multi-camera system that focuses on high processing rate and GPU utilization with minimum latency bounds. In the next chapter, relevant academic works are presented that address the above problems.

Chapter 3

Related Works

This chapter reviews related works on multi-camera stream pipeline, scheduling and optimization techniques. It explores various methods adopted in multi-camera systems. It identifies common scheduling approaches adopted in both multi and single stream systems. It also discusses on the frame rate processing policies used to extract a necessary frames from a stream.

3.1 Multi-camera systems

Object detection has been applied to the systems of traffic signs, face recognition, and event-based auto-recording. In most cases, these systems are set with multiple cameras for detecting the object per frame received by each camera which is at a different location. As many approaches proposed by researchers and the progress of hardware platforms, the detection accuracy has significantly increased. However, for real-time features of the mentioned systems, the main concern is the high computational cost affecting the real-time performance [1]. This can be solved by the right choice of architecture and computation pipeline.

3.1.1 System architecture

The authors of [7] address the challenges of system architecture for multi-camera surveillance systems. They propose a distributed processing architecture whose limitations include communication bandwidth and latency. Therefore, a centralized architecture is preferred in this project. Rafael Martín and José María present a multi-camera system for analytics in a tennis game. The system detects and tracks each player on the court or field in single-player sports [8]. An extension of the previous work with a scalable architecture is accomplished in [9]. The authors propose a processing module framework that manages the data acquisition and processing phases independently. This solution is modular and flexible, so the project inspires a similar framework.

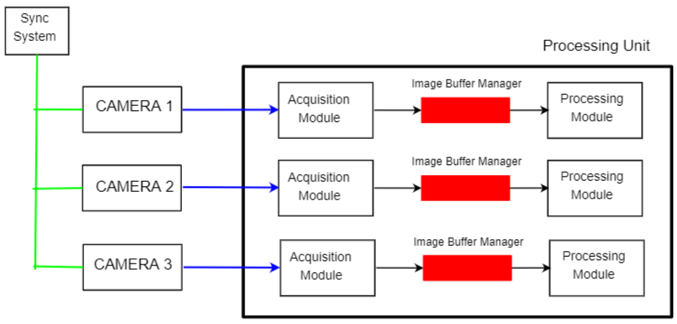


Figure 3.1: Processing unit proposed in [9]

3.1.2 Pipeline design

Multi-core CPUS and Graphics Processing Units (GPU) have been recently used to achieve the real-time performance of computer vision. However, as the consideration of performance, they are designed for different types of tasks. For applications of high parallelism and large input data size such as image processing, GPU can achieve superior performance. On the other hand, the multi-core CPU can achieve a better performance for those applications of light parallelism or heavily control-oriented [10].

A video processing pipeline on a heterogeneous system consists of data transfer and computation tasks spread over CPU and GPU. Several factors aid in boosting the computation time. Authors of [11] implement a pipeline that offloads partial computations to the CPU. However, the speaker in [12] addresses that CPU+GPU based video processing is largely a data movement problem. The data transmission results in increased traffic on shared buses used in parallel by other tasks. Hence, it is recommended to avoid computation offloading approach. He also demonstrates the increase in throughput by a concurrent copy and execution approach. A complete implementation of a video processing pipeline on a GPU is discussed in [13].

The authors of [9] device a parallelization strategy for efficient implementation of their processing module. The strategy is based on a pipeline scheme, that is one of the most suitable approaches for image processing task parallelization. Furthermore, this approach allows to parallelize the applications at a high level, thus shielding the users from thread-level programming details. The author identifies sub-blocks into which the application can be divided. Sub-blocks consist of atomic tasks such as image scaling, colour plane conversion for frame pre-processing. Due to the different characteristics, the efficient parallelization of the sub-blocks requires the observance of computational demands and data dependencies at several levels: i) between consecutive frames, ii) within a single video frame, and iii) between the processing tasks [14]. Once the sub-blocks of the application have been identified, it is possible to design the pipeline stages as a single sub-block or a set of them. Then, the pipeline is built as a cascade of stages.

3.1.3 Scheduling

Scheduling a video pipeline is the application-level mapping of computation and data flow to the hardware. The problem of task mapping in heterogeneous systems is finding the proper assignment of tasks to processors to optimize some performance metrics such as system utilization, load balancing, and minimum execution time [15]. The next generation of surveillance systems focuses on the design of scheduling control with a more robust and adaptive algorithm. A lack of resources should be taken into account when designing a scheduling strategy that aims at maximising the resource utilization and system accuracy.

To tackle the problem, researchers such as Hampapur et al. [16] proposed several camera scheduling algorithms designed for different application goals. They include, for example, a round-robin method that assigns computation sequentially and periodically to the hardware to achieve uniform coverage. Rather than assigning equal importance to each camera, Qureshi et al. [17] proposed ordering frames in a priority queue based on their arrival time and the frequency with which they are captured. Both these methods do not always guarantee the optimal use of resources.

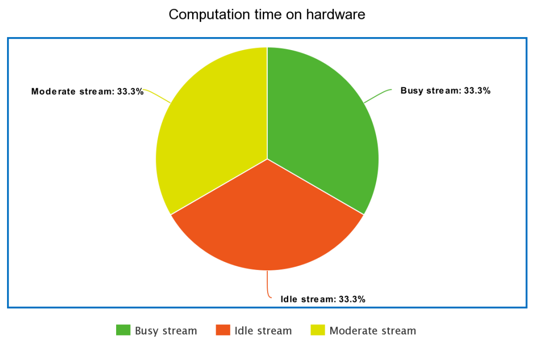
Therefore, the authors of [18] came up with scheduling strategies based on performance models. The main idea was to estimate the cost of scheduling a given task on each worker (CPU core or GPU) to help the scheduler minimize the application's execution cost. This cost can include the execution time, memory consumption, and/or power and energy consumption. To this end, one may exploit theoretical or empirical methods in this strategy. Theoretical approaches involve considering the peak processing time of each worker, the bandwidth and latency of the available memory, and the time to solution and memory complexity of each task. Based on this information, the scheduler can approximate the completion time of each task for each worker and make the appropriate scheduling decisions to reduce the overall execution time. Unfortunately, on many systems, peak performance estimates can differ significantly from the actual performance seen during execution. This discrepancy or gap is often closed by improving the performance model itself by measuring the real performance parameters of the system and the execution time of each task [18].

3.2 Dynamic processing

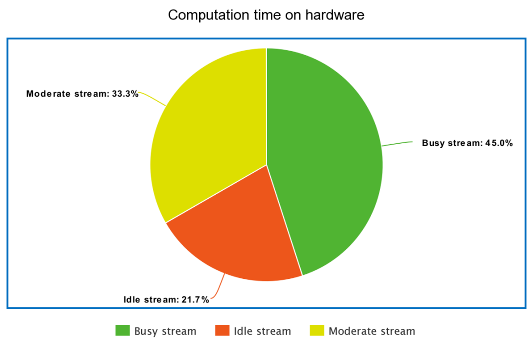
An excellent scheduling strategy facilitates in growing the efficiency of a system performance and in the utilization of available resources in the best possible way. The problem of limited resource capacity and high data rate can be analogous to the universal supply-demand problem. Due to the high data rate, the multi-camera system demands a higher computation capacity. With the computation capacity and task execution time being constant, the challenge is to balance the data inflow rate.

Load balancing is the act of distributing work loads to group of processors. Many studies have been carried out on various static and dynamic load balancing strategies [19]. In static scheduling, the assignment of loads to processors is done before program execution begins. Information regarding task execution times and processing resources is assumed to be known at compile time. The major advantage of static scheduling methods is that all the overhead of the scheduling process is incurred at compile time, resulting in a more efficient execution time environment compared to dynamic scheduling methods. However, the resulting schedule is constant and generates optimal solutions in restricted cases [20].

Dynamic scheduling is based on the redistribution of loads (frames) during the execution time. This redistribution is performed by transferring loads from the heavily loaded processors (busy stream) to the lightly loaded processors (idle streams) with the aim of improving the performance of the application. An inspiration is made from the above concept to dynamically adapt the computation time spent on busy and idle streams. The computation time is shared proportionately based on the content of video streams. This is depicted in Figure 3.2 as the computation time for the busy stream is increased by reducing from the idle stream. The advantage of dynamic load balancing over static scheduling is that the system need not be aware of the run-time behavior of the applications before execution. The flexibility inherent in dynamic load balancing allows for adaptation to the unforeseen application requirements at run-time [20].



(a) Stream wise computation time scenario



(b) Dynamic balancing

Figure 3.2: Dynamic re-distribution of computation time

3.2.1 Stream classification

In a dynamic scheduling mechanism, the scheduling is done based on any significant event. The event can be the arrival of a frame, completion of task execution, content of the video, etc. Bodor et al. [21] propose a dual-camera system with one wide-angle static camera and a PTZ camera for pedestrian surveillance. Human activities (walking, running, etc.) are prioritized based on the preliminary recognition by the wide-angle camera. The PTZ camera focuses on the activity with the highest priority for further analysis. Similarly, the number of detected objects can be used as a criteria to determine the relative stream priority.

3.2.2 Frame selection

Analyzing high frame rate video streams imposes heavy computing needs and significant loads on the network. High frame rates may not be essential for meeting the accuracy requirements of the analyses. For example, high frame rates may not be required to track cars inside a garage compared to cars on a highway. The main battle of studying how to select informative frames of videos is in the video summarization field. There are different characteristics of the video content that determine the necessary frame rate, such as the speed of motion of the object, direction of motion, and occlusion by other objects.

Yeung et al. designed a policy network to directly predict the temporal bounds of actions in action detection, which decreased the need for processing the whole video and improved the detection performance [22]. The work in [23] propose a light-weight neural network to evaluate the frame differences and schedule them for detection or tracking. The above works require a share of computation to identify key-frames.

A simple and efficient approach is implemented in [3]. The author time sample frames at fixed intervals. Unlike other methods, time sampling does not require any computation. Object detection based on sparse and temporally-adaptive key frame scheduling is discussed in [24]. The work in [25] proposes a drop-out method to determine the necessary frame rates for object detection. The frame rate is controlled by dropping some frames from the detection task at the processing end. For example, a frame drop rate of 0% is equivalent to the original frame rate (e.g., 25 FPS). A frame drop rate of 50% is achieved by dropping alternate frames and reducing the frame rate by half (e.g., 12.5 FPS). They demonstrate that the frame rate can be reduced up to 80% based on accuracy constraints.

For a jogging sequence, the frame rate can be reduced by 80% without decreasing the precision rate. Since the camera follows the person being tracked in the jogging sequence, the displacement does not increase with reducing frame rate and the frame rate can be dropped aggressively. For a basketball sequence, the frame rate can be reduced up to 50% without decreasing the precision rate. In the basketball sequence, the player moves at high speed with sudden changes in direction and is often occluded by other players. Therefore, the tracking error increases as the frame rate reduces. Similarly, for a cyclist sequence the frame rate can be reduced by 33% [25].

Conclusion

There are several major insights drawn from the above works. A centralized architecture ensures a predictable and bounded latency. The pipeline design for a multi-stream computation needs to be efficiently designed and parallelised for better performance. A dynamic scheduling with a carefully designed estimation model performs better than a static scheduler. The resources are more efficiently utilized in such a solution. The drop-out frame selection approach saves redundant computation time with a minor trade-off on accuracy. The contribution of this work is to introduce a dynamic drop-out rate based on the content of the video streams.

Chapter 4

Methodology

This chapter formulates the concept and design of a multi-stream video processing system. A new approach is adopted towards building a scheduler. The architecture and sub-modules of the system are defined and design steps are identified to create the proposed scheduler. The software architecture is completely developed in C++ and the management of image data and processing tasks is based on OpenCV libraries.

4.1 Proposed system

The project aims to build a scheduler to manage the high data rate and system performance for a multi-camera video processing system. The objectives of the proposed scheduler include resource capacity estimation and fair allocation of tasks based on the video content. The engagement level of each video stream is prioritized by a stream priority. A high engagement is defined when more objects are present in a video stream. Therefore, the results from the detection network are used to determine the engagement level of a video content. A relative prioritization on all streams deduces the stream priority.

The computation time of the system is predominately shared by detection and tracking tasks. The drop out method introduced in [25] omits detection on certain frames to reduce computations without a compromise on accuracy. The contribution of this work is to adopt a dynamic drop-out method to share the resources among video streams. The proposal is to build a dynamic scheduler that maintains a discrete detection rate on the video streams. The rate of detection is determined together with stream priority and the total available computation time.

The dynamic scheduler is designed in two levels: Global and Local level. The global scheduler determines the maximum number of detection and tracking executions per camera stream. The local scheduler selects frames by a fair selection algorithm and allocates the detection and tracking request to the hardware. The local scheduler targets a frame level scheduling at a frequency equal to the video streaming rate, i.e., 25 FPS. Whereas, the estimation by the global scheduler is expected to occur each second. Hence, the global scheduler has the capability to adopt new stream priority values every 1 second or per 25 frames.

The system is designed to operate in two modes to meet the different requirements of applications. For instance, in a self-driving application, a large enough latency could significantly increase the risk of accidents. Unexpected events such as animal crossing or jay walking can happen over just a few frames. In these cases, the response time of each task is extremely critical and must be a bounded value with a small error margin. In other non-real-time applications such as surveillance, a higher throughput can be traded with the response time for atomic tasks. Therefore, the system is proposed with two choices of operation: i) Throughput mode and ii) Latency mode. Both modes use a timing model to estimate the available computation time and the scheduler allocates tasks proportionately.

4.2 Architecture of proposed system

The focus and scope of the system demands a centralized processing approach. The architecture consists of four modules and an input console: Frame grabber, Scheduler, Processor, Stream priority module and Mode selection console. A schematic representation is presented in Figure 4.1. The hardware platform consists of Nvidia GeForce GTX 1050 Ti GPU and Intel(R) Core(TM) i7-2600K quad-core CPU platform

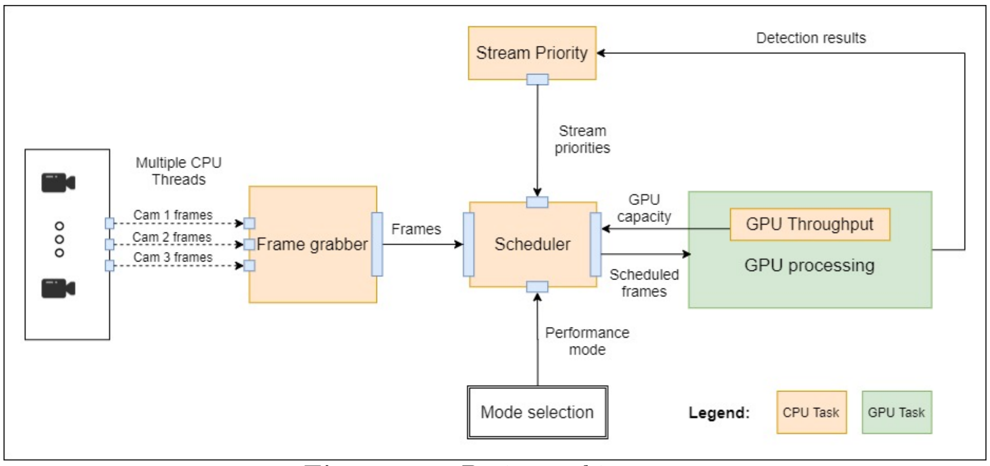


Figure 4.1: Design architecture

Frame Grabber

The cameras are interfaced to the scheduler through the grabber module. The function of a grabber module is to continuously read frames from camera streams and deliver them as a list to the scheduler. This process is implemented on the CPU and a parallelism is achieved at the camera level by multi-threading. The process contain multiple threads. A unique thread is assigned to each camera and each thread reads video frames.

GPU Throughput

A timing model estimates the total number of detection and tracking tasks. The estimation is performed each second (every 25 frames) and sent to the scheduler. In the GPU, frames are processed in the order of their arrival, regardless of which camera thread they belong to.

Stream Priority

The detection results from each camera stream are captured in this module. A relative priority algorithm is used to determine individual stream priorities. These results are passed on as a list to the scheduler.

Scheduler

The scheduler forms the heart of the system. It receives a list of frames and stream priorities as input. The estimated resource capacity and desired mode of scheduling are obtained as well. The scheduler uses a fair algorithm to select frames and schedule detection or/and tracking tasks on the hardware. A multilevel scheduling strategy is used to enable different levels of task granularity. At the coarse-grained level, the resource estimation happens every 25 FPS. Whereas, at the fine-grained level, the task allocation occurs at every frame and is scheduled on the GPU.

Mode Selection

The choice of mode is selected via the input console. The system can operate in a throughput mode or latency mode specified as commands before run-time.

4.3 Design steps

The design phase is essentially broken into 4 phases and the tasks involved are listed below. The process figures out an efficient design, estimates a theoretical system throughput and distributes to streams according to their priorities.

1. Identification of processing strategy

   The three main tasks of the application are Frame Copy, Detection and Tracking. The characteristics of each task is to be studied to map them on the CPU/GPU hardware. It is important to find an efficient computation pipeline of the tasks on the hardware to avoid stalling. An experiment is performed to study the serial and parallel execution of tasks on the system. The design choice of the computation pipeline is decided based on the outcome of the experiment. Different strategies are required to satisfy the individual throughput and latency requirements. Therefore, the first step is to discover the efficient processing strategy for each operating modes.

2. Estimation of GPU capacity

   To build the scheduler it is important to know the processing capacity i.e., maximum number of detection and tracking that can be computed per second while maintaining a fixed input sampling rate of video streams. To determine the GPU capacity, a timing model is built and the theoretical detection and tracking tasks per second are estimated. The model computes the time required for one execution cycle of the computational pipeline and formulates the possible activities in one second.

3. Dynamic priority computation

   The priority of a stream is decided by the number of objects in the scene relative to other streams. The detection results are to be monitored and categorized using a relative prioritization method. This monitoring process continues for every frame. However, the priorities are updated after every second (25 frames). These priorities together with the available computation time decides the detection and tracking requests per stream. A relative prioritization method is opted as it can dynamically adjust to the variations in the course of time and compute the priority accordingly.

4. Scheduler algorithm for fair selection

   After the allocation of tasks per stream, it is important to focus on the frame selection algorithm. All frames execute tracking task while only the selected frames execute the detection task. The combination of selected frames affect the tracking accuracy as the detection results have an impact on the tracking performance. Therefore, a selection algorithm needs to be devised to maintain a fair selection of frames between the priority updating interval (i.e., 25 frames).

   A regular drop-out method discussed in literature ensures fair selection of frames only below a factor of 2. For instance, to select 12 frames from an interval of 25 frames, we can select a frame and omit the succeeding frame for a fair selection. However, when we need to select 17 frames from an interval of 25 frames, we can not maintain a periodic selection or omission. A fractional knapsack problem is inspired to identify these 17 frames among the interval. The solution is discussed and modified to fit our requirements in the next chapter.

Chapter 5

Implementation

This chapter explores each design steps in detail. The design choices are supported by performing experiments. The formulation of the timing model and the scheduler algorithm are demonstrated.

5.1 Processing strategy

The motive this section is to build a multi-stream computation pipeline for the two modes of operation. The section begins with a task analysis of the system. Two experiments are followed to make a concrete design choice. In the end, two pipelines are designed to fit the requirements of each mode.

Single stream system characteristics

A single stream computation pipeline consists of three major tasks: Frame copy, Detection and Tracking. The characteristics of each task is reported in Table 5.1. An average measurement after multiple executions is reported as the GPU platform exhibit a non-deterministic run-time behaviour. The frame copy task takes about 0.7 ms to execute and moves frames from each camera to the GPU. The detection is the most computation-intensive task of the pipeline consisting of frame pre-processing and inference using the SSD model. It is completely a GPU task and takes about 14 ms. Tracking task requires a heterogeneous processing on CPU and GPU. The execution time depends on the number of objects tracked. The average execution time in all cases is found to be a total of 4 ms.

Table 5.1: Task characteristics

Tasks	**Resource**	**Avg. execution time (ms)**
Frame copy	CPU	0.7
Detection	GPU	14
Tracking	CPU+GPU	4

Multi stream computation experiment

A multi-camera processing consists of multiple task computations on the heterogeneous platform. Task parallelism is achieved by multi-threading on the CPU. Similarly, parallelism on GPU can be realised by CUDA/GPU streams. A CUDA/GPU stream is a sequence of operations (commands) that are executed in order. Multiple GPU streams can be created and executed together and interleaved. Operations within the same stream are ordered (FIFO) and cannot overlap. However, operations in different streams are un-ordered and can overlap.

An experiment to study the impact on the end to end latency by overlapping GPU streams w.r.t the number of video channels is performed. Figure 5.1 illustrates the observed behaviour. It can be inferred that the end to end latency decreases as the number of GPU streams increase for a particular configuration of the video channel. For instance, the end to end latency is at the minimum for a six GPU stream configuration on a six-channel multi-camera system. Hence, it can be concluded that multiple GPU streams can be overlapped for a high system throughput. Therefore, the first design choice is to maintain individual GPU streams for each video channel on the GPU. However, concurrent execution of tasks increases the individual task latency.

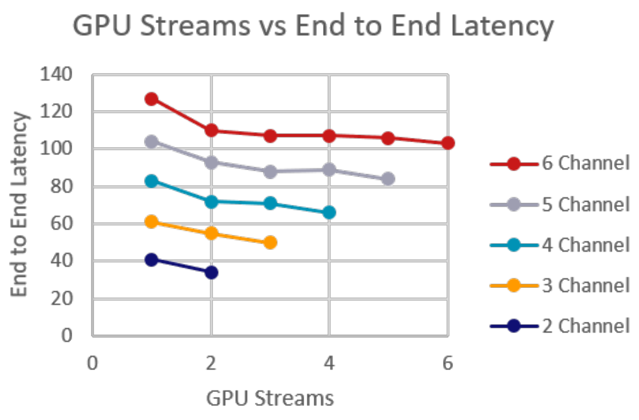
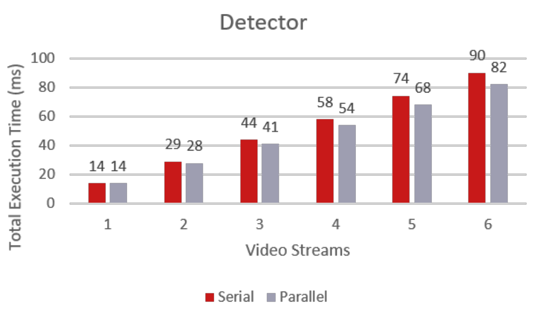


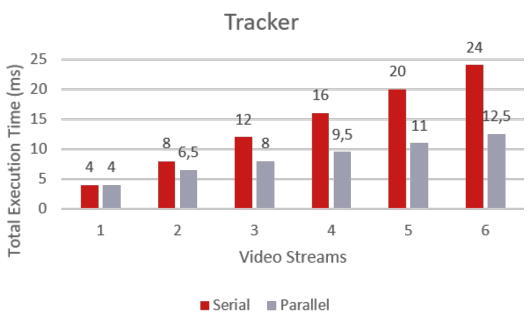
Figure 5.1: Impact of GPU streams on execution time

The second step is to study the effect of serial and task parallelism on the heterogeneous system. An experiment is performed to observe the throughput and latency metrics using the serial and parallel processing strategy. The impact of the number of input video streams on the performance metrics is also studied for up to six camera streams.

Figure 5.2 compares the results of the above experiment. It is observed that the serial and parallel execution of the detection task does not drastically affect the performance compared to the tracking task. However, the difference in execution time can be accounted for the reuse of memory during parallel inference. Due to the multi-threaded execution, the parallel tracking task witnesses a half-fold reduction on the execution time. Hence, the conclusion is to execute the trackers in parallel for a high throughput.



(a) Detector



(b) Tracker

Figure 5.2: Timing performance

Pipeline design

It is important to identify an efficient computation pipeline for the two performance modes based on the experiments above. In a throughput mode, the goal is to process as many frames as possible irrespective of the latency of individual tasks. On the contrary, a latency mode emphasises on obtaining the results within a stipulated time. Therefore, task parallelism is adopted in throughput mode and a serial execution is preferable in latency mode. The design of the computational pipeline is depicted in Figure 5.3. This sequence will be adopted by the scheduler during processing.

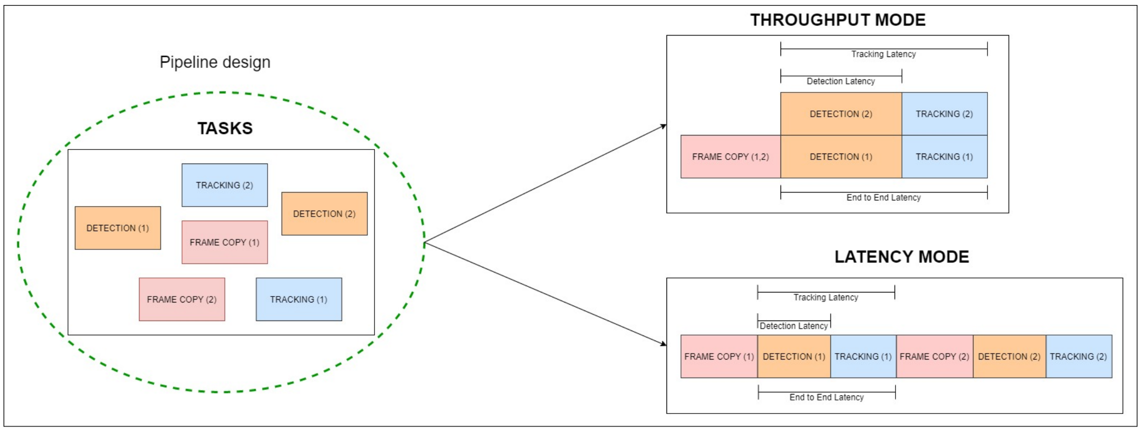


Figure 5.3: Pipeline design

5.2 Throughput estimation on GPU

The GPU throughput module of the system estimates the application throughput on the hardware resources. A timing model is created based on the task characteristics for the estimation of computation time. The throughput is estimated for every 1 second (25 frames) using the created model. One execution cycle consist of frame copy, detection and tracking tasks. The cumulative sum for N execution cycles is formalised below:

$$TotalFrames * CopyTime + TotalDetection * DetectionTime \\ + TotalTracking * TrackingTime = Nexecutioncycles$$

Where N is the number of frames per second and

$$TotalFrames = No.ofVideoStreams * Nframes,$$
$$TotalTracking = No.ofVideoStreams * Nframes,$$
$$TotalDetection = \sum DetectionInEachVideoStream$$

Considering a 3 camera system with an input rate of 25 FPS and the tracker is required to be executed for all frames,

$$TotalFrames = 3Streams * 25frames = 75,$$
$$TotalTracking = 3Streams * 25frames = 75,$$
$$TotalDetection = Detectionin3VideoStreams = D$$

The objective is to find the value of D for a three-camera system by using the corresponding execution time obtained from Figure 5.2. The per-stream computation time for a three-camera system operating in a parallel/throughput mode is found to be:

$$CopyTime = 0.7ms, DetectionTime = 41ms \div 3 = 14ms, TrackingTime = 8ms \div 3 = 2.67ms$$

The estimation is performed for every one second after obtaining the execution timing information as below:

$$\begin{aligned} 75 * CopyTime + D * DetectionTime + 75 * TrackingTime &= 1000ms \\ 75 * 0.7ms + D * 14ms + 75 * 2.67ms &= 1000ms \\ 52.5ms + 15ms * D + 200ms &= 1000ms \\ Therefore, D &= 53 \end{aligned}$$

A similar estimation is performed with different camera streams in both modes of operation. The theoretical estimation plot of the expected number of detection frames w.r.t video streams is plotted in 5.4. The timing model estimates a total of 53 possible detection executions in throughput mode and 43 in the case of latency mode for a three-camera system.

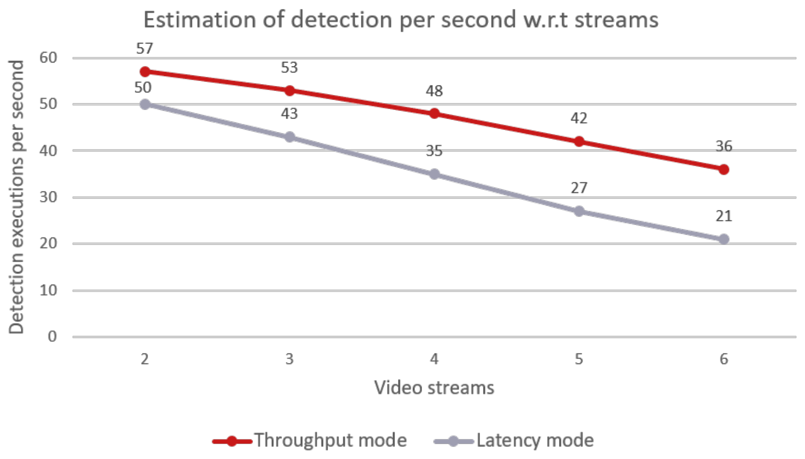


Figure 5.4: Theoretical estimation of detection per second

5.3 Dynamic priority computation

This module identifies the stream activity by collecting the detection network results of each stream. The average of the detected objects is found every one second (25 frames). A relative share determines the proportion of the average objects in each stream. Finally, the number of detection frames are calculated by multiplying the relative share with the estimation from the previous section.

$$RelativeShare_i = \frac{X_i}{\sum_{i=1}^{N} X_i}$$

Two examples of priority calculations in a three-camera system are shown below. Table 5.2 shows the calculation for the throughput mode and Table 5.3 for the latency mode.

Example i): Mode = Throughput, Max. Detection execution estimated = 53

Table 5.2: Priority calculation in throughput mode

Channel	Avg. object detected	Relative share	Detection frames
Stream 1	18	$18 \div 45 = 0.40$	$0.40 * 53 = 21$
Stream 2	15	$15 \div 45 = 0.33$	$0.33 * 53 = 17$
Stream 3	12	$12 \div 45 = 0.27$	$0.27 * 53 = 14$
Total	**45**		**52**

Example ii): Mode = Latency, Max. Detection execution estimated = 43

Table 5.3: Priority calculation in latency mode

Channel	**Avg. object detected**	**Relative share**	**Detection frames**
Stream 1	18	18 ÷ 45 = 0.40	0.40 * 43 = 17
Stream 2	15	15 ÷ 45 = 0.33	0.33 * 43 = 14
Stream 3	12	12 ÷ 45 = 0.27	0.27 * 43 = 12
Total	**45**		**43**

In this way, the number of detection frames per camera stream is identified. This information is passed onto the scheduler to select and schedule the respective detection frames from each stream.

5.4 Scheduler algorithm for fair selection

The responsibilities of the scheduler can be divided into two. First, calculation of the estimated GPU capacity. Second, frame selection and detection or/and tracking task allocation. A multilevel scheduling strategy is used for this purpose. A global scheduler performs the resource estimation every 25 FPS. Whereas, a local scheduler does the task allocation every frame.

The local scheduler receives a list of frames and stream priorities as input. Together they determine the number of detection frames for each stream. Now, the challenge is to select the frames for detection from the interval of 25 frames spread equally. A fair selection algorithm is to be devised.

The inspiration is taken from the "Fractional Knapsack Problem". It is an algorithmic problem in combinatorial optimization in which the goal is to fill a container (the "knapsack") with fractional amounts of different materials chosen to maximize the value of the selected materials. Given the weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack. An example is visualised in Figure 5.5.

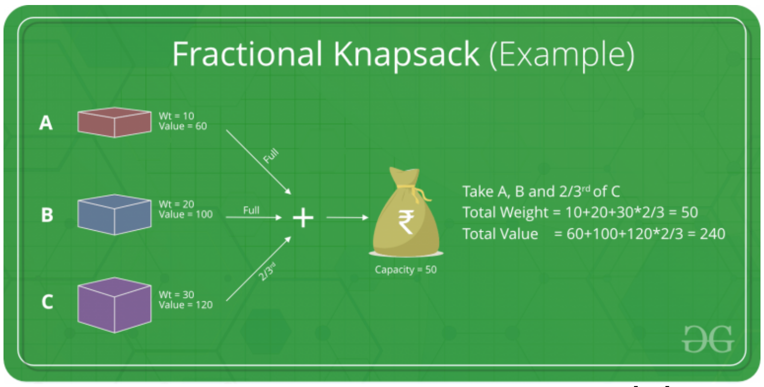
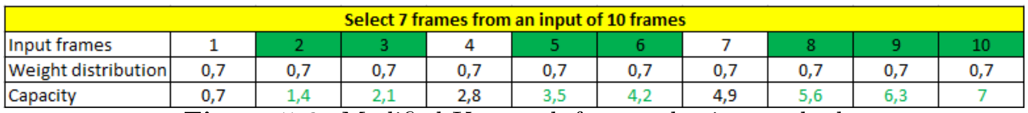


Figure 5.5: Fractional-Knapsack example in [26]

There are three items A, B, and C than can fill a knapsack of capacity W. The problem is to identify a knapsack combination whose total value is the largest. To figure out the combination, all items are sorted according to its value per weight and the fractional share is identified.

A multiple knapsack problem is an extension of the current problem where more containers are to be filled. A similar approach is adapted with equally distributed weights and values to pick the frames that make a container full. An illustration of this concept is shown in Figure 5.6. The task is to select seven frames from an input of 10 frames. Therefore, as a first step, the weights are distributed equally (7÷10 = 0.7) among all 10 frames as shown in row 'Weight Distribution'. Each frame is stacked upon another and the frame that fills every unit container is selected.



Select 7 frames from an input of 10 frames										
Input frames	1	2	3	4	5	6	7	8	9	10
Weight distribution	0,7	0,7	0,7	0,7	0,7	0,7	0,7	0,7	0,7	0,7
Capacity	0,7	1,4	2,1	2,8	3,5	4,2	4,9	5,6	6,3	7

Figure 5.6: Modified Knapsack frame selection method

A pseudo-code of the above algorithm is made in Algorithm 1. The goal is to identify frames that will perform the detection task in an interval of N frames. The execution begins with weight per frame calculation. The weight is stacked upon to check if the unit capacity has overflown for each frame. When the full capacity is achieved, the corresponding frame is scheduled to do a detection task. The excess capacity is carried forward to the next container. In this manner, a fair selection is achieved and frame numbers 2, 3, 5, 6, 8, 9, 10 will be selected for the task in Figure 5.6.

Algorithm 1 Frame selection

```
1: Weight = Number of frames to be selected ÷ Total frame interval
2: for $frames = 1, 2, \ldots, N$ do
3:     Capacity += Weight
4:     if $Capacity >= 1$ then
5:         Schedule frame for Detection task
6:         Capacity -=1
7:     end if
8: end for
```

The above algorithm is extended to multiple streams and the detection tasks are allocated in a fair manner. Figure 5.7 demonstrates a fair frame selection in a three-camera stream on a small-scale GPU with pre-determined priority values.

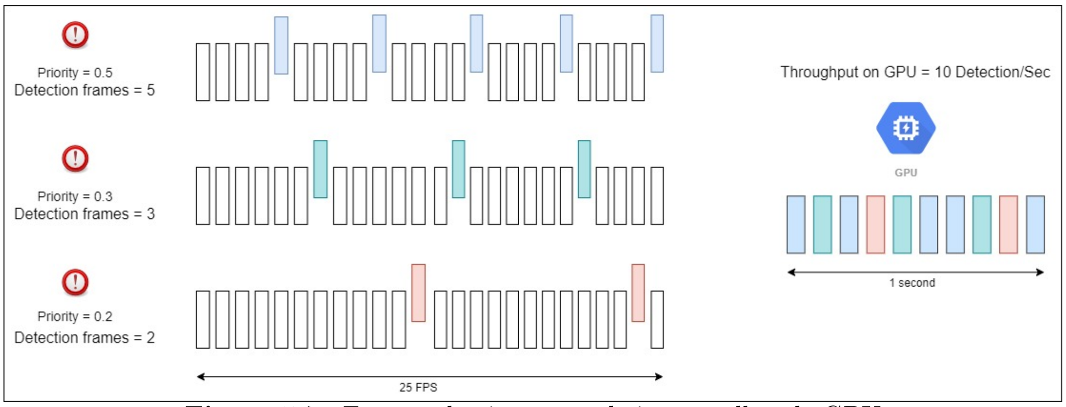


Figure 5.7: Frame selection example in a small-scale GPU

Chapter 6

Validation framework

The chapter aims to discuss the performance metrics and evaluation of the proposed multi-camera system. As mentioned, the system will be evaluated based on the metrics such as throughput, latency, GPU resource utilization, average precision, and average recall for object detection. The throughput, latency and GPU utilization are computed using software timers and NVIDIA's Nsight System tool. There are several tools to provide a timeline view of application behaviour in run time. NVIDIA's Nsight System enables a holistic view of the entire system, CPU, GPU, OS, and the workload itself with low overhead profile collection and minimal perturbation [27].

6.1 Experiment set-up

The general set-up for the experiment is detailed below:

- OS: Ubuntu 18.04
- CPU: Intel 4-Core i7-2600K CPU @ 3.40GHz
- GPU: GeForce GTX 1050 Ti (768 core Pascal)
- CUDA version: 10.0.326
- cuDNN version: 7.6.3
- TensorRT version: 6.0.1
- OpenCV version: 4.1.1
- NVIDIA Nisght System version: 2020.1.1
- Profiling method: Host to Host

6.2 Throughput, Latency and GPU utilization

In this section, the focus is on measuring the throughput, task latency, and GPU utilization in both operating modes. Throughput is measured as the number of detection executions per second. A task latency is measured as the amount of time to complete that particular task. The GPU utilization is computed as the percentage of time in a unit second when the GPU is busy. The measurement is performed via two interfaces. Internally, software timers are triggered as required. Externally, Nsight Systems provides a timeline view of the process executed in the GPU and helps to identify idle periods. Therefore, the throughput, task latency, and GPU utilization is affirmed by both methods.

6.2.1 Sample benchmarking

The multi-camera system on the GPU as GTX 1050 Ti is profiled on Nsight Systems. Figure 6.1 is a graphical representation of the timeline. The following information is collected and presented on Nsight Systems from the timeline view: Threadwise API traces of CUDA, TensorRT, Profiler overhead and GPU utilization. On top of the timeline, all threads running in the application appears. For a two-camera system, there are three major threads, namely: Camera 1 (31528), Camera 2 (31529), and Frame copy (31482). Each thread contains information on the schedule of various APIs along with profiler overhead if any.

In the bottom of the timeline, the GPU node appear and contains CUDA streams. The streams contain memory operations and kernel launches on the GPU. Kernel launches are represented by blue, while memory transfers are displayed in red and CUDA synchronizations are represented by green blocks. The light blue dense regions above the GPU node indicate the occupancy of GPU. Blank spaces between the dense regions show that the GPU is idle.

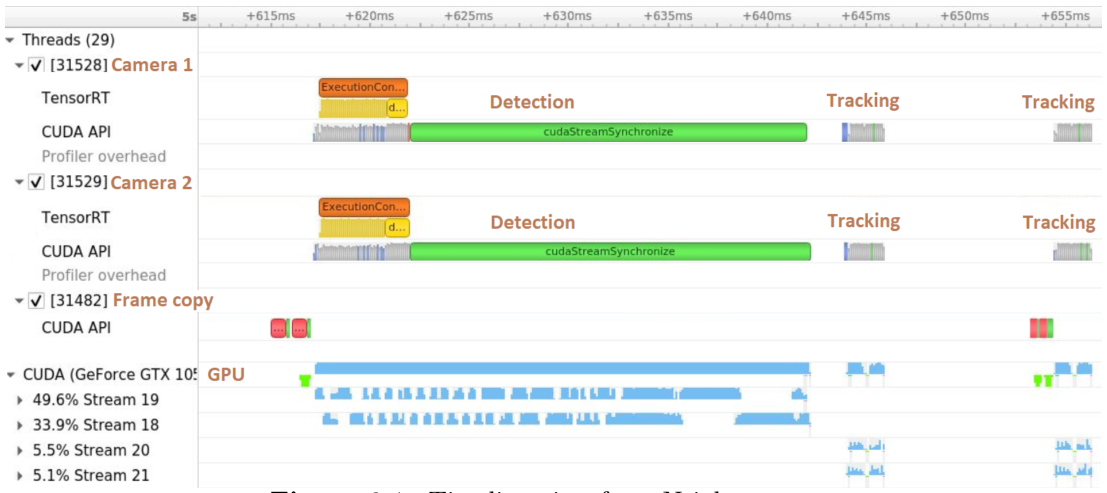


Figure 6.1: Timeline view from Nsight systems

The sequential activities in Figure 6.1 begin with the transfer of frames from the two cameras to the GPU. The threads Camera 1 and Camera 2 run concurrent detection and tracking tasks. The cycle continues as per the tasks allocated by the scheduler. The throughput and GPU utilization are identified for every interval of one second. Latency is preferred to be calculated by software timers as it is tough to measure manually for each task.

6.3 Accuracy evaluation

Object detection models generate a set of detections where each detection consists of coordinates for a bounding box and a confidence score. These results are evaluated over the annotations to determine the performance of an object detector. The performance is measured in average precision, average recall, and F1-score. Precision indicates how many of the bounding boxes detected are actual objects, where recall shows how many of the objects are found. The F1-score is the harmonic mean of the precision and recall.

6.3.1 Evaluation dataset

UA-DETRAC is a challenging real-world multi-object detection and multi-object tracking benchmark. The dataset consists of videos of vehicles in four categories, i.e., car, bus, van, and others. It also contains four categories of weather conditions, i.e., cloudy, night, sunny, and rainy with different occlusion and truncation ratios. The videos are recorded at 25 frames per second (fps), with a resolution of 960×540 pixels. There are more than 140 thousand frames in the UA-DETRAC dataset and 8250 vehicles that are manually annotated [28][29][30]. Five videos of a highway intersection are selected with different video characteristics. The object detection performance is analysed for a static (Round-Robin) and the proposed dynamic scheduling algorithms.

Figure 6.2: DETRAC dataset

6.3.2 Performance metrics

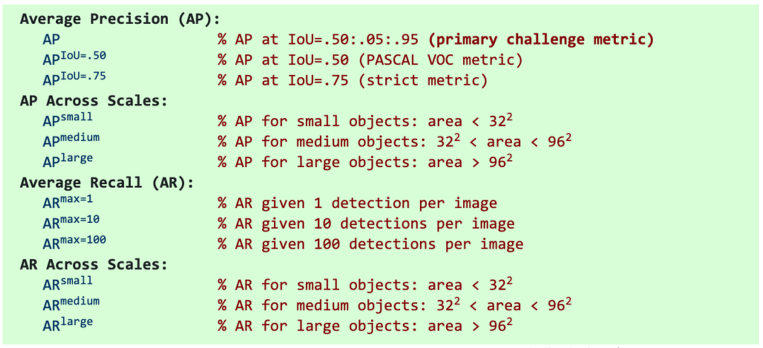
The concept of Intersection over Union (IoU) is used for object detection. IoU computes intersection over the union of the two bounding boxes; the bounding box for the ground truth and the predicted bounding box. An IoU of 1 implies that the predicted and the ground-truth bounding boxes perfectly overlap. It is possible to set a threshold value for the IoU to determine if the object detection is valid or not. For instance, IoU $\geq$ 0.5, classify the object detection as True Positive (TP). IoU $<$ 0.5, then it is a wrong detection and classify it as False Positive (FP). When a ground truth is present in the image and model fails to detect the object, it is classified as False Negative (FN). Precision and Recall are calculated using TP, FP, and FN.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

By setting the threshold for confidence score at different levels, we get different pairs of precision and recall. With recall on the x-axis and precision on the y-axis, we can draw a precision-recall (PR) curve, which indicates the association between the two metrics. The area under the curve is used to obtain Average Precision. These are automatically extracted using the COCO APIs: cocoEval.evaluate(), cocoEval.accumulate(), and cocoEval.summarize(). The summarized metrics of the evaluation are listed in Figure 6.3. Among the 12 metrics, $\text{AP}^{\text{IoU=0.50}}$, $\text{AP}^{\text{IoU=0.75}}$, $\text{AR}^{\text{Medium}}$, and AR^{Large} are reported for comparison.



Average Precision (AP):

AP	% AP at IoU=.50:.05:.95 **(primary challenge metric)**
$AP^{IoU=.50}$	% AP at IoU=.50 (PASCAL VOC metric)
$AP^{IoU=.75}$	% AP at IoU=.75 (strict metric)

AP Across Scales:

AP^{small}	% AP for small objects: area < 32^2
AP^{medium}	% AP for medium objects: 32^2 < area < 96^2
AP^{large}	% AP for large objects: area > 96^2

Average Recall (AR):

$AR^{max=1}$	% AR given 1 detection per image
$AR^{max=10}$	% AR given 10 detections per image
$AR^{max=100}$	% AR given 100 detections per image

AR Across Scales:

AR^{small}	% AR for small objects: area < 32^2
AR^{medium}	% AR for medium objects: 32^2 < area < 96^2
AR^{large}	% AR for large objects: area > 96^2

Figure 6.3: Metrics supported by COCO API

Chapter 7

Results

This chapter observes the performance of the proposed system in different metrics. It discusses the key differences of each mode. A scatter plot of variations in the task latency is plotted. Finally, the system is evaluated against the DETRAC dataset and the accuracy metrics are compared to a static and dynamic scheduler.

7.1 Processing mode validation

The design choice of the computation pipeline for each mode is verified using Nsight Systems. The timeline view illustrates the flow of computation along with the GPU utilization.

Throughput mode

A three-camera system is configured and profiled in throughput mode. Figure 7.1 represents the timeline where multiple detection and tracking tasks are run in parallel. As a result, we observe increased task latency of each parallel task. This is visible on the first tracking task of camera 3. The tracking task is not able to complete due to resource contention by the detection tasks of camera 1 and 2. The end-to-end latency of the task is measured to be 35ms. The GPU is mostly occupied between two execution cycles.

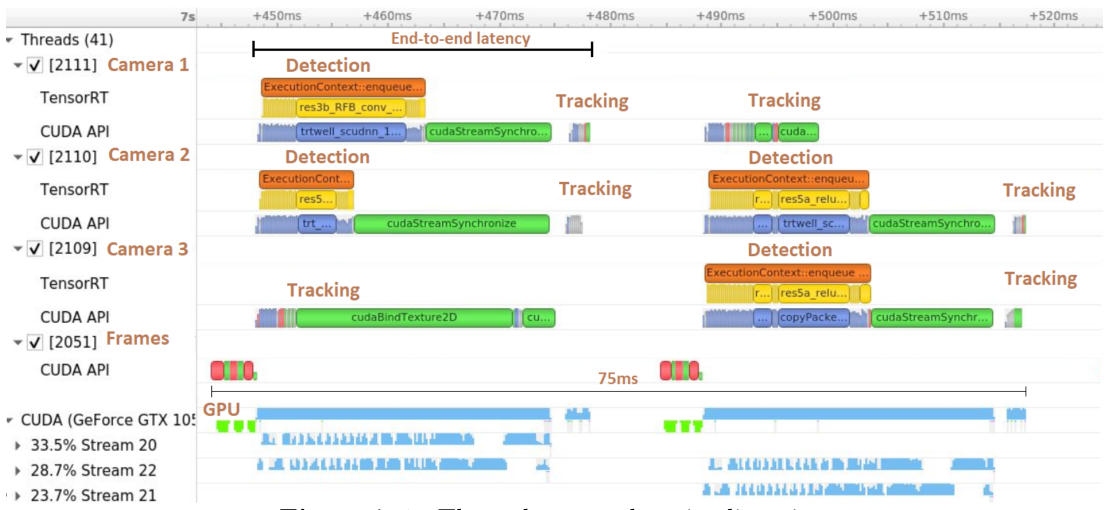


Figure 7.1: Throughput mode - timeline view

Latency mode

The same three-camera system is now configured in latency mode. Figure 7.2 represents the timeline where multiple detection and tracking tasks are executed one after another. A difference in processing strategy is observed in the frame reading and execution. Evidently, the problem of longer task latency is resolved in latency mode. The end-to-end latency is also lower compared to the other mode. However, a new problem was introduced. The execution time was less than the inter-frame arrival duration. Hence, the GPU is idle and waiting for the next frame to arrive. Due to this phenomenon, the throughput of the system is less than the other mode. It is observed from both timelines that two execution cycles (3 frames per cycle) in latency mode takes almost 90ms whereas only 75ms in the throughput mode.

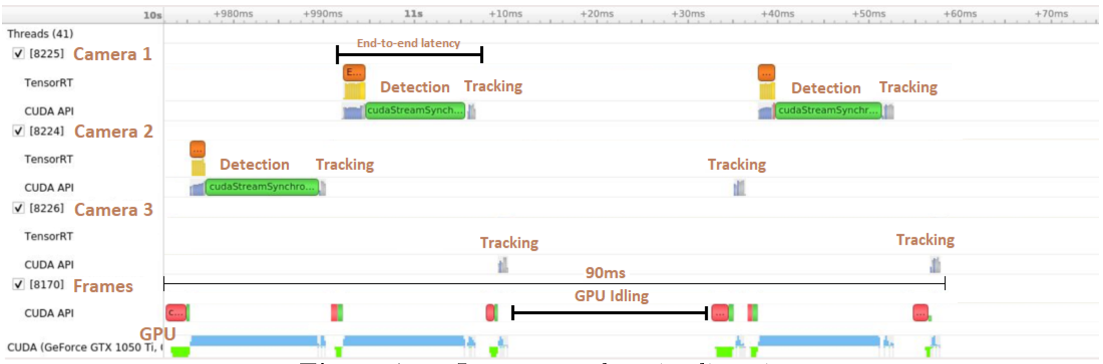
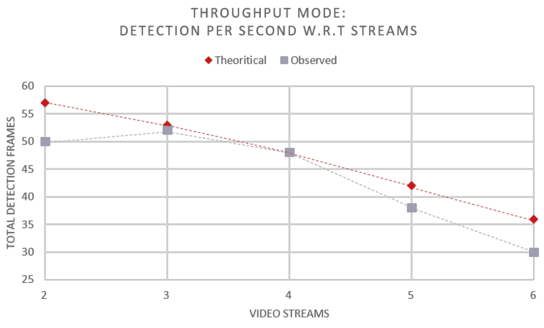


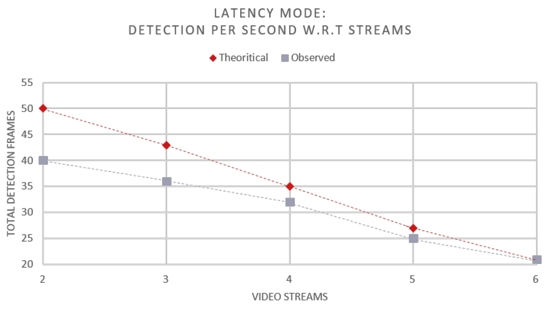
Figure 7.2: Latency mode - timeline view

7.2 Timing model verification

The timing model built-in section 5.2 estimates the number of detection and tracking executions possible in a unit time interval. The multi-camera system experiments for both modes with increasing input streams and total detection per second is plotted in Figure 7.3. The plot in both the sub-figures are almost in line with the theoretical expectations. Some deviations are observed because the model does not take in to account the idling of GPU when the next frame has not arrived yet.



(a) Throughput mode



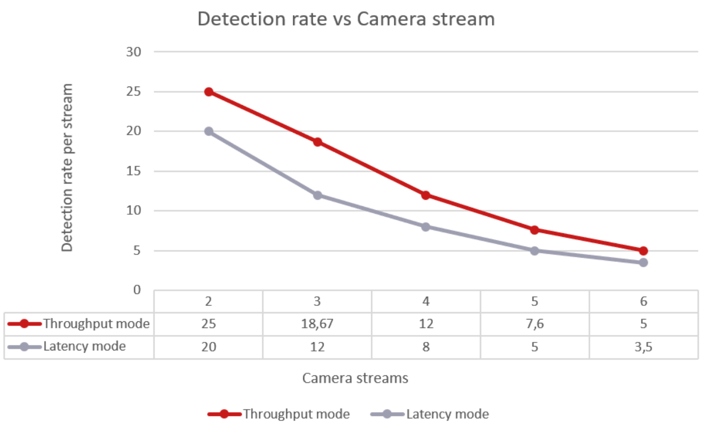
(b) Latency mode

Figure 7.3: Results of GPU capacity estimation

Therefore, the current model always provides an upper bound estimation. The estimation is proved to be realistic when there are minimal idling periods in the computation pipeline. An interesting observation as a part of this experiment was noted in CPU load. An increase in the CPU load was observed as the no. of streams increased. So, there is a certain limit to the no. of input streams the system can handle with the same hardware.

7.3 Throughput observation

The throughput of the system is predominantly decided by the detection rate as the tracking task is executed for all the frames. The detection rate of a multi-stream computation pipeline in the throughput mode is studied against that of latency mode. Figure 7.4 shows the impact on processing rate on the number of camera streams. The detection rate of individual stream goes down by adding further input streams on the same hardware. The throughput modes deliver a better detection rate for individual streams than latency mode.



	2	3	4	5	6
Throughput mode	25	18,67	12	7,6	5
Latency mode	20	12	8	5	3,5

Figure 7.4: Throughput per camera stream

7.4 Latency observation

The latency of detection and tracking tasks are studied and scatter plots are made for each stream. The experiment was performed on a six-camera input system on both the operating modes.

7.4.1 Detection task

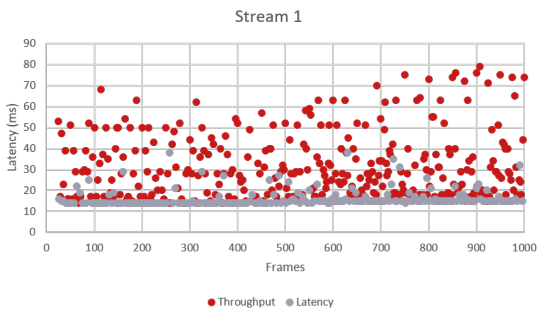
The detection task is always scheduled after the frame copy. The latency computation is tabulated in Table 7.1.

Table 7.1: Detection task latency

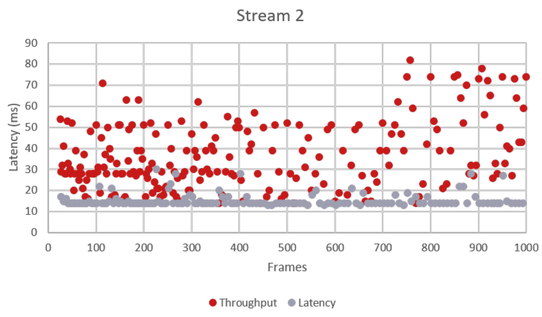
Stream	Throughput mode (36 det/sec)				Latency mode (21 det/sec)			
	Best Case (ms)	Worst Case (ms)	Avg. (ms)	Std. Dev	Best Case (ms)	Worst Case (ms)	Avg. (ms)	Std. Dev
Camera 1	14	79	31.85	15.56	14	38	16.21	4.13
Camera 2	14	82	36.40	15.88	13	30	15.30	3.15
Camera 3	13	78	44.98	17.16	13	33	15.03	3.01
Camera 4	13	78	32.15	15.12	14	36	16.39	4.35
Camera 5	15	78	39.99	16.18	13	37	15.66	4.05
Camera 6	14	78	45.36	16.31	13	33	15.31	3.71

The observations from the table are listed below:

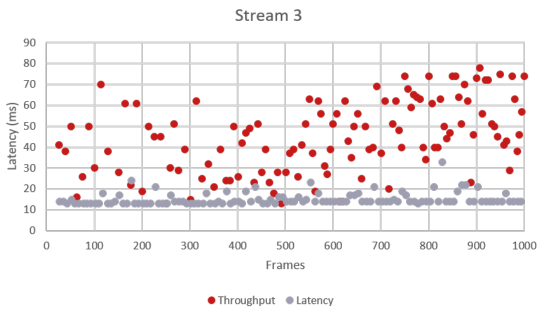
- The best case latency is similar in both modes.
- The worst-case latency is half-folded in latency mode when compared to throughput mode. This is due to the difference in the design choice of the pipeline.
- The average task execution time is observed to be high in throughput mode.
- Throughput mode also has a very high latency deviation due to concurrent execution.
- With smaller deviations, execution time is closely bounded in latency mode.



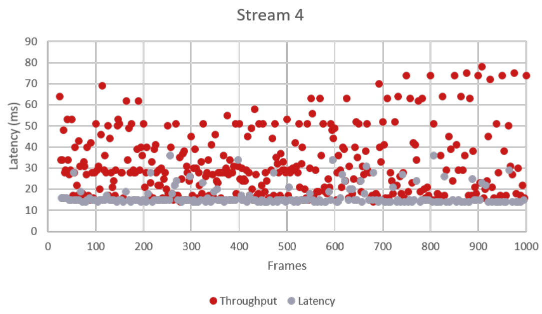
(a) Video stream no: 1



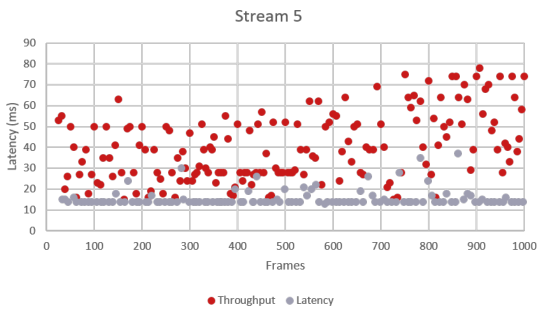
(b) Video stream no: 2



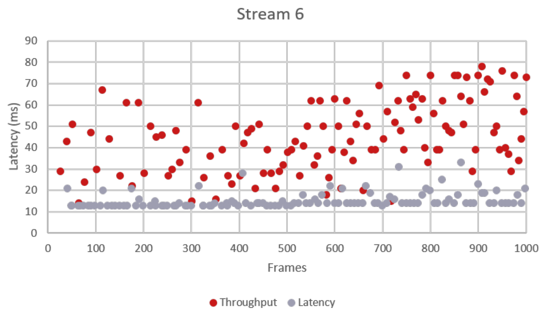
(c) Video stream no: 3



(d) Video stream no: 4



(e) Video stream no: 5



(f) Video stream no: 6

Figure 7.5: Comparison of detection latency on two modes

The scatter plot of task latency for each of the six streams is shown in Figure 7.5 and the insights are listed below:

- The noisy execution time (red dots) in throughput mode is bounded in latency mode. This is because two or more parallel executions in throughput mode prolongs the execution time of other concurrent tasks.
- A straight line of similar values is visible in the latency mode. There are few deviations observed that are probably noises due to the non-deterministic nature of GPU. These contribute to a slightly higher error margins in the latency mode.

7.4.2 Tracking task

The tracker is either executed after the detection task or immediately after the frame copy when detection is not scheduled. When a tracker is executed after detection, the results from detection are always copied to the tracker. The execution time depends on the number of objects being tracked. The latency computation for the tracking task is tabulated in Table 7.2.

Table 7.2: Tracking task latency

Stream	Throughput mode (36 det/sec)				Latency mode (21 det/sec)			
	Best Case (ms)	Worst Case (ms)	Avg. (ms)	Std. Dev	Best Case (ms)	Worst Case (ms)	Avg. (ms)	Std. Dev
Camera 1	1	58	13.61	9.56	1	38	3.25	3.97
Camera 2	0	49	10.73	9.32	0	22	1.50	1.84
Camera 3	0	52	11.51	9.38	0	19	1.33	1.34
Camera 4	2	56	13.79	9.31	1	40	3.25	3.42
Camera 5	1	57	12.13	9.68	0	30	1.48	1.79
Camera 6	0	49	12.27	9.19	0	24	1.52	1.93

The observations from the table are listed below:

- The best case latency is similar in both modes and a value of zero indicates no objects were present in the scene and hence the execution of the tracker was redundant.
- The worst-case latency is decreased in latency mode when compared to throughput mode. This is due to the difference in the design choice of the pipeline.
- The average task execution time is observed to be high in throughput mode.
- Throughput mode also has a very high latency deviation due to concurrent execution.
- With smaller deviations, execution time is closely bounded in latency mode.

It is observed in Figure 7.6 that a tracking task immediately after a detection executes for a longer period. The same behaviour is not observed in subsequent frames where only tracking is scheduled. This is due to the transfer of a new set of feature points from the detector to the CPU to continue further execution. Therefore, seemingly two sets of grey lines are visible on each plot in Figure 7.7.

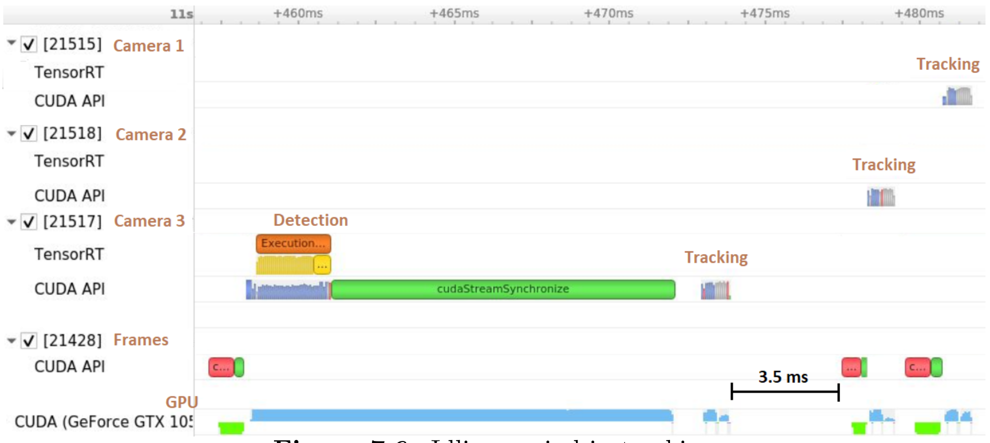
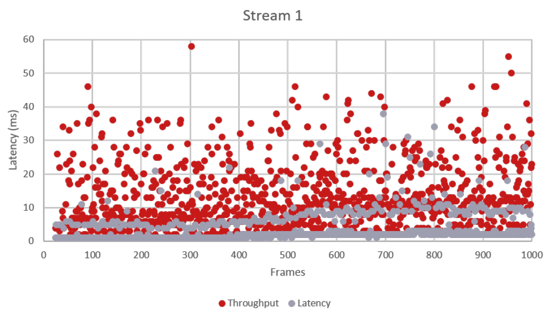
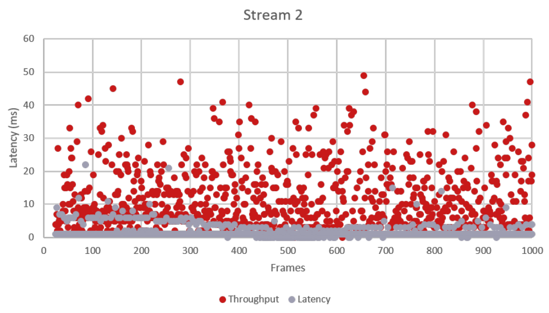


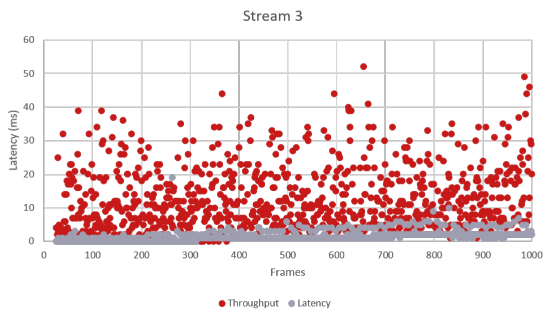
Figure 7.6: Idling period in tracking



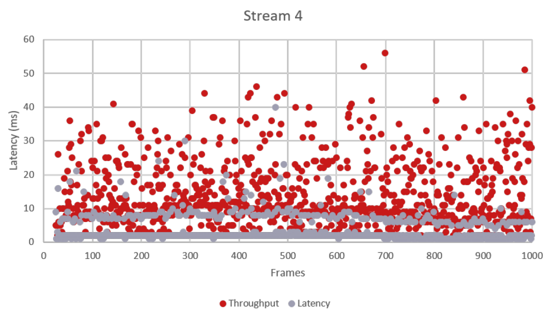
(a) Video stream no: 1



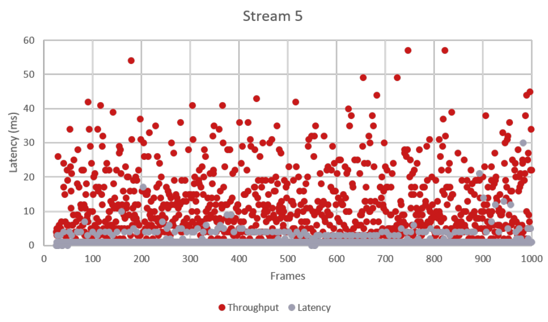
(b) Video stream no: 2



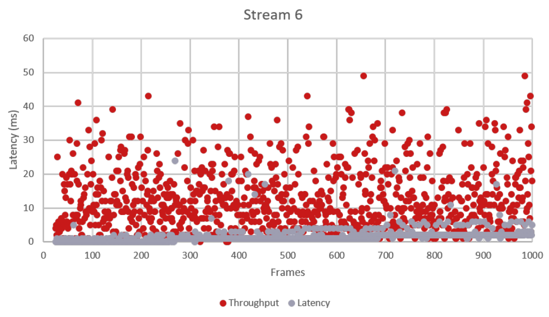
(c) Video stream no: 3



(d) Video stream no: 4



(e) Video stream no: 5



(f) Video stream no: 6

Figure 7.7: Comparison of tracker latency on two modes

The scatter plots of task latency for each of the six streams are shown in Figure 7.7 and the insights are listed below:

- The noisy execution time (red dots) in throughput mode is bounded in latency mode. This is because two or more parallel executions in throughput mode prolongs the execution time of other concurrent tasks.

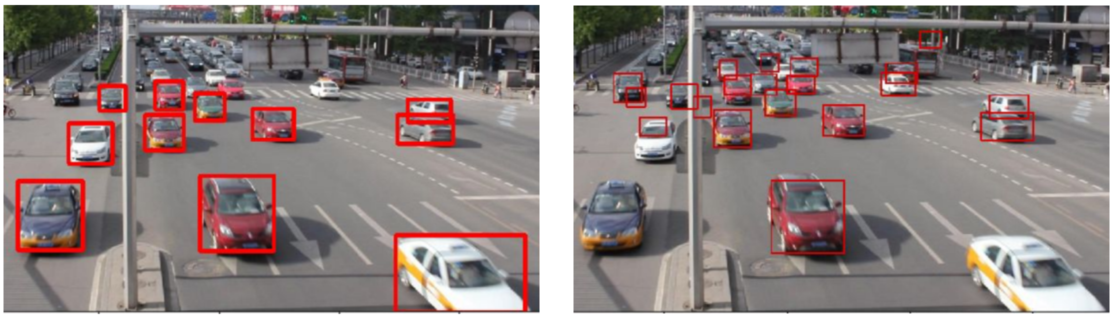
- Seemingly, two straight line of similar value are visible in the latency mode. The transfer of a new set of feature points prolongs the execution time after every detection. Hence, two

7.5 Accuracy evaluation

The need for accuracy evaluation is due to the variable detection rate of multiple streams in the proposed system. To judge the performance results, the proposed dynamic scheduler is compared with a static scheduler. The static schedule uses a round-robin method to schedule the detection task. The experiment is performed on a five-camera system with five video streams of different characteristics. The confidence score is set at 0.5, anything below this is not classified as detection.

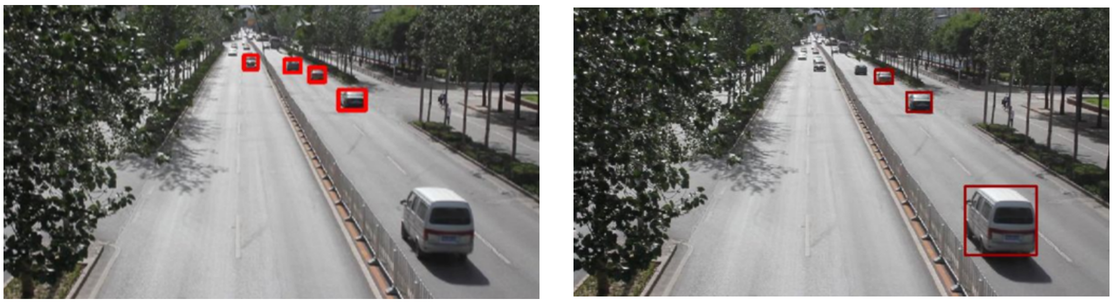
7.5.1 Baseline comparison

The work by Zwemer et. al presents a detector trained on 'train set' and evaluated over 'validation set' on the DETRAC dataset. Their Average Precision (AP) was found to be 90.6% [31]. The same detector configuration is used in the project so a similar value is expected. However, only an AP of 34% was observed. The inconsistency was investigated and found that the detector identified more than what is present in the ground truth. Therefore, it was treated as a false positive prediction during the evaluation which drops the precision. This is observed in Figure 7.8. However, in Figure 7.9 a different case impacting the overall recall is observed. Some of the very small appearing cars are failed to be identified but available in the ground truth.



(a) Ground truth **(b)** Detection result

Figure 7.8: Object detection and tracking in a busy highway



(a) Ground truth **(b)** Detection result

Figure 7.9: Object detection and tracking in an inactive highway

Therefore, a baseline metrics was required to compare and contrast the performance of the schedulers. The baseline was created by obtaining results by executing detection on every frames over the five video streams. The same is used as a base for comparison with schedulers of different detection rate.

7.5.2 Results

The experiment was performed in throughput mode with a static and dynamic scheduler. The static round-robin scheduler schedules a detection once in every five frames periodically. Whereas, the dynamic scheduler prioritizes the video content and decides a relative detection rate. So, busy streams have a higher detection rate and idle streams are set a minimum detection rate at the worst case. It is then possible to detect objects in busy streams early and tracking is done accurately. More detection help to reduce tracking of false-positive predictions in streams. By maintaining a minimum detection rate the system does not compromise the performance of idle streams. Therefore, we expect the average precision (AP) and average recall (AR) of the dynamic scheduler to increase. The classic AP metric used in PASCAL VOC challenge (AP @ IoU =0.50), AP @ IoU = 0.75, AR of medium and large-sized objects are reported to compare the performance. The expected results are obtained in Table 7.3.

Table 7.3: Static vs Dynamic scheduler comparison

Metrics	**Baseline**	**Static**	**Dynamic**
AP @ IoU=0.50	34%	27.9 %	31.0 %
AP @ IoU=0.75	22%	15.4 %	18.5 %
AR medium	36%	29.7 %	33.0 %
AR large	27.3%	24.5 %	26.4 %

The F1-score of static and dynamic scheduler is calculated to be 0.28 and 0.32 respectively. A good F1-score indicates that the system has good precision and a good recall simultaneously. Interestingly, the dynamic scheduler in latency mode and a static scheduler is the same. They both operate on the same detection rate. Hence, the dynamic scheduler outperforms a static scheduler in all aspects.

Chapter 8

Conclusions

8.1 Summary

This work has shown the impact of different system configurations on the performance of a multi-camera system. A multi-camera system that performs dynamic scheduling based on video content is designed and implemented on the same hardware platform. The video streams are prioritized relatively based on the detection results. A timing model estimates the total computation time and proportionately allocates the detection task based on the stream priorities. A scheduler algorithm was devised for fair selection of frames from video streams.

Two operating modes are created to achieve different application requirements. The throughput mode focuses on high detection rate per stream and the latency mode achieves low task latency. The characteristics of each task were studied and an effective computation pipeline for each mode was designed to meet their needs. The system was developed in the high performing C++ language and deployed in a standard GeForce GTX 1050Ti GPU platform. The system utilized the SDK provided by NVIDIA, thus ensuring optimal performance.

The performance of the system is evaluated in metrics such as throughput, latency, GPU utilization, and accuracy. A throughput mode maintains a high detection rate per second and high GPU utilization. This is achieved by task parallelism in the pipeline. However, it suffers from high task latency due to concurrent processing. The latency mode bounds the execution time and minimizes the task latency by 50%. It is also observed to be consistent with very minimal deviations.

The potential of the dynamic and static scheduler is studied by evaluation on the DETRAC dataset. The dynamic scheduler outperforms a static scheduler in all aspects. It helps to improve the tracking accuracy on busy streams. The average precision @ IoU = 0.50 and average recall of medium objects for the object detection performance show an increase by 11%. The average precision @ IoU = 0.75 and the average recall of large objects increase by 20% and 8% respectively. The F1-score is higher by 12.5%.

The improvement of the multi-camera system in metrics such as throughput, latency, and accuracy is demonstrated, which suggests that dynamic processing is an efficient approach when there are more constraints on the hardware, such as in an embedded environment.

8.2 Future work and direction

This work has opened the doors for some future directions of research. Some of the possible future works are described below:

- Better metric to classify video streams

  The classification of a video stream is implemented based on the number of objects present in the frame. An unfair classification occurs when the objects are static. For example, cars waiting at a traffic signal should be classified as an idle stream. However, it may not be so when 100 cars are waiting.

- Mixed mode processing

  The system considers all streams operating on the same mode. An interesting challenge would be to optimize one stream for latency and another for throughput.

Bibliography

[1] C. Lin, S. Teng, Y. Chen, and P. Hsiung. Real-time object detection for multi-camera on heterogeneous parallel processing systems. In *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 446–450, 2013. 1, 9

[2] James Allen. Ai traffic video analytics platform being developed. `https://www.traffictechnologytoday.com/news/traffic-management/ai-traffic-video-analytics-platform-being-developed.html`, September 2019. 2

[3] Robin Schrijvers, Steven Puttemans, T. Callemein, and Toon Goedemé. Real-time embedded person detection and tracking for shopping behaviour analysis. In Jacques Blanc-Talon, Patrice Delmas, Wilfried Philips, Dan Popescu, and Paul Scheunders, editors, *Advanced Concepts for Intelligent Vision Systems*, pages 541–553, Cham, 2020. Springer International Publishing. 3, 12

[4] ViNotion B.V. Product overview. `https://vinotion.nl/en/products/`, February 2020. 5

[5] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016. 5

[6] J.Y. Bouguet. Pyramidal implementation of the lucas kanade feature tracker. *Tech.rep., Microprocessor Research Labs, Intel Corporation*, 2000. 5

[7] Yong Wang, Rui Zhai, and Ke Lu. Challenge of multi-camera tracking. In *Proceedings - 2014 7th International Congress on Image and Signal Processing, CISP 2014*, 10 2014. 9

[8] R. M. Nieto and J. M. M. Sánchez. An automatic system for sports analytics in multi-camera tennis videos. In *2013 10th IEEE International Conference on Advanced Video and Signal Based Surveillance*, pages 438–442, 2013. 9

[9] Massimo Camplani and Luis Salgado. Scalable software architecture for on-line multi-camera video processing. *Proceedings of SPIE - The International Society for Optical Engineering*, 7871, 02 2011. 9, 10

[10] C. Lin, S. Teng, Y. Chen, and P. Hsiung. Real-time object detection for multi-camera on heterogeneous parallel processing systems. In *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 446–450, 2013. 10

[11] H. Qiu, X. Liu, S. Rallapalli, A. J. Bency, K. Chan, R. Urgaonkar, B. S. Manjunath, and R. Govindan. Kestrel: Video analytics for augmented multi-camera vehicle tracking. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 48–59, April 2018. 10

[12] Thomas J True. Topics in gpu based video processing. In *2016 GPU Technology Conference*, July, 2016. 10

[13] Puren Guler, Deniz Emeksiz, Alptekin Temizel, Mustafa Teke, and Tugba Taskaya Temizel. Real-time multi-camera video analytics system on gpu. *Journal of Real-Time Image Processing*, 11, 03 2013. 10

[14] S. Momcilovic, A. Ilic, N. Roma, and L. Sousa. Dynamic load balancing for real-time video encoding on heterogeneous cpu+gpu systems. *IEEE Transactions on Multimedia*, 16(1):108–121, 2014. 10

[15] Doaa M. Abdelkader and Fatma Omara. Dynamic task scheduling algorithm with load balancing for heterogeneous computing system. *Egyptian Informatics Journal*, 13(2):135 – 145, 2012. 10

[16] A. Hampapur, S. Pankanti, A. Senior, Ying-Li Tian, L. Brown, and R. Bolle. Face cataloger: multi-scale imaging for relating identity to location. In *Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance, 2003.*, pages 13–20, 2003. 10

[17] Qureshi and Terzopoulos. Surveillance camera scheduling: a virtual vision approach. In *Multimedia Systems*, pages 269–283, 2006. 10

[18] Negin Bagherpour, Sven Hammarling, Jack Dongarra, and Mawussi Zounon. Novel methods for static and dynamic scheduling. *Parallel Numerical Linear Algebra for Extreme Scale Systems*, pages 1–19, 2017. 10, 11

[19] M. Hamdi and C.K. Lee. Dynamic load-balancing of image processing applications on clusters of workstations. *Parallel Computing*, 22(11):1477 – 1492, 1997. 11

[20] Behrooz Shirazi, Ali Hurson, and Krishna Kavi. Scheduling and load balancing in parallel and distributed systems. *Wiley-IEEE Computer Society Press*, pages 1–5, 1995. 11

[21] R. Bodor, R. Morlok, and N. Papanikolopoulos. Dual-camera system for multi-level activity recognition. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 1, pages 643–648 vol.1, 2004. 11

[22] Yangyu Chen, Shuhui Wang, Weigang Zhang, and Qingming Huang. Less is more: Picking informative frames for video captioning. *CoRR*, abs/1803.01457, 2018. 12

[23] Hao Luo, Wenxuan Xie, Xinggang Wang, and Wenjun Zeng. Detect or track: Towards cost-effective video object detection/tracking. *CoRR*, abs/1811.05340, 2018. 12

[24] Xizhou Zhu, Jifeng Dai, Lu Yuan, and Yichen Wei. Towards high performance video object detection. *CoRR*, abs/1711.11577, 2017. 12

[25] A. Mohan, A. S. Kaseb, K. W. Gauen, Y. Lu, A. R. Reibman, and T. J. Hacker. Determining the necessary frame rate of video data for object tracking under accuracy constraints. In *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, pages 368–371, 2018. 12, 13

[26] GeeksforGeeks. *Greedy Algorithms.* GeeksforGeeks Blog, November 2017. `https://www.geeksforgeeks.org/greedy-algorithms/`. 21

[27] Scott McMillan. *Transitioning to Nsight Systems from NVIDIA Visual Profiler / nvprof.* NVIDIA Developer Blog, August 2019. `https://devblogs.nvidia.com/transitioning-nsight-systems-nvidia-visual-profiler-nvprof/`. 23

[28] Longyin Wen, Dawei Du, Zhaowei Cai, Zhen Lei, Ming-Ching Chang, Honggang Qi, Jongwoo Lim, Ming-Hsuan Yang, and Siwei Lyu. UA-DETRAC: A new benchmark and protocol for multi-object detection and tracking. *Computer Vision and Image Understanding*, 2020. 25

[29] Siwei Lyu, Ming-Ching Chang, Dawei Du, Wenbo Li, Yi Wei, Marco Del Coco, Pierluigi Carcagnì, Arne Schumann, Bharti Munjal, Doo-Hyun Choi, et al. Ua-detrac 2018: Report of avss2018 & iwt4s challenge on advanced traffic monitoring. In *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2018. 25

[30] Siwei Lyu, Ming-Ching Chang, Dawei Du, Longyin Wen, Honggang Qi, Yuezun Li, Yi Wei, Lipeng Ke, Tao Hu, Marco Del Coco, et al. Ua-detrac 2017: Report of avss2017 & iwt4s challenge on advanced traffic monitoring. In *Advanced Video and Signal Based Surveillance (AVSS), 2017 14th IEEE International Conference on*, pages 1–7. IEEE, 2017. 25

[31] Matthijs Zwemer, R.G.J. Wijnhoven, and Peter H.N. de With. Ssd-ml: hierarchical object classification for traffic surveillance. In Giovanni Maria Farinella, Petia Radeva, and Jose Braz, editors, *15th International Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISAPP2020)*, pages 250–259. SCITEPRESS-Science and Technology Publications, Lda., February 2020. 33