

ACCELERATION OF IMAGE PROCESSING MODULES IN WIDE-AREA AERIAL SURVEILLANCE

A Thesis

by

Zeynep Gülbeyaz Demirdağ

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Electrical & Electronics Engineering

Özyeğin University
December 2022

Copyright © 2022 by Zeynep Gülbeyaz Demirdağ

ACCELERATION OF IMAGE PROCESSING MODULES IN WIDE-AREA AERIAL SURVEILLANCE

Approved by:

Prof. Hasan F. Ateş, Advisor
Dept. of Electrical & Electronics
Eng.
Özyeğin University

Prof. Bahadır K. Güntürk
Dept. of Electrical & Electronics
Eng.
Istanbul Medipol University

Asst. Prof. İsmail Aktürk, Coadvisor
Dept. of Computer Science
Özyeğin University

Assoc. Prof. Dionysis Goularas
Dept. of Computer Eng.
Yeditepe University

Date Approved: December 28, 2022

Prof. H. Fatih Uğurdağ
Dept. of Electrical & Electronics
Eng.
Özyeğin University



To my family,

ABSTRACT

Wide Area Aerial Surveillance (WAAS) is a type of surveillance system that refers to monitoring of large geographical areas in real-time. A WAAS system is designed using multiple cameras and/or sensors mounted on an Unmanned Aerial vehicle (UAV). WAAS systems can scan an area of several square kilometers at once. However, the huge amount of data collected by WAAS systems can be challenging to process in real-time on board of the aircraft. This study proposes to use Graphics Processing Units (GPU) and multi-core programming techniques to accelerate the performance of the four key modules of a WAAS system: image matching/stitching, object detection, tracking, and super-resolution. Image matching/stitching involves registering and combining multiple images captured by cameras on UAV to create a panoramic image, in other words, the mosaic image of the area being monitored. Object detection and tracking modules involve identifying and tracking the movements of objects, such as cars, trucks, and people. Finally, the super-resolution module uses computational techniques to enhance the resolution of the images and provides more details on the images. The WAAS system's speed and efficiency are significantly improved by using the GPU and multi-core programming techniques to accelerate these modules. GPUs are well-suited to this task because they are designed for parallel processing, allowing them to process large amounts of data in real-time. The experiments show that using GPUs and multi-core programming techniques can significantly improve the performance of image stitching, object detection, tracking, and super-resolution, making it possible to execute these modules in parallel and process the large amount of data collected from WAAS in real-time. The accelerated modules are tested on NVIDIA Jetson AGX Xavier embedded GPU card for challenging test scenarios, demonstrating their potential for real-time surveillance on edge devices.

ÖZETÇE

Geniş Alan Havadan Gözetleme, GAHG, geniş coğrafi alanların gerçek zamanlı olarak izlenmesini ifade eden bir tür gözetim sistemidir. GAHG sistemi, insansız hava aracı (İHA) üzerine monte edilmiş birden fazla kamera veya sensör kullanılarak tasarlanmaktadır. GAHG sistemleri tek seferde birkaç kilometrekarelik bir alanı tarayabilir. Ancak, GAHG sistemleri tarafından toplanan büyük miktardaki verilerin uçakta gerçek zamanlı olarak işlenmesi zor olabilir. Bu çalışma, bir GAHG sisteminin dört temel modülünün performansını hızlandırmak için grafik işleme birimleri (GPU) ve çok çekirdekli programlama tekniklerinin kullanılmasını önermektedir: görüntü eşleştirme/birleştirme, nesne algılama, izleme ve süper çözünürlük. Görüntü eşleştirme/birleştirme, panoramik bir görüntü, başka bir deyişle izlenen alanın mozaik görüntüsü oluşturmak için İHA'daki kameralar tarafından çekilen görüntülerin kaydedilip birleştirilmesini içerir. Nesne tespit ve takip modülleri, arabalar, kamyonlar ve insanlar gibi nesnelerin hareketlerini tespitini ve takibini içerir. Son olarak, süper çözünürlük modülü, görüntülerin çözünürlüğünü artırır ve görüntüler hakkında daha fazla ayrıntı sağlar. Bu modülleri hızlandırmak için GPU ve çok çekirdekli programlama teknikleri kullanılarak WAAS sisteminin hızı ve verimliliği önemli ölçüde iyileştirilmiştir. GPU'lar, paralel işleme için tasarlandıkları ve büyük miktarda veriyi hızlı bir şekilde işlemelerine izin verdiği için bu görev için çok uygundur. Deneyler, GPU'ları ve çok çekirdekli programlama tekniklerini kullanmanın, GAHG sistemi modüllerinin performansını önemli ölçüde iyileştirebileceğini ve bu modüllerin paralel olarak yürütülmesini ve GAHG'tan toplanan büyük miktarda verinin gerçek zamanda işlenmesini mümkün kıldığını gösteriyor. Hızlandırılmış modüller, zorlu test senaryoları için NVIDIA Jetson AGX Xavier gömülü GPU kartı üzerinde test edilerek uç cihazlarda gerçek zamanlı gözetim potansiyelleri ortaya konmuştur.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisors Prof. Dr. Hasan Fehmi Ateş and Asst. Prof. İsmail Aktürk for their guidance and support. Their expertise and knowledge were invaluable in helping me to complete this thesis. I would also like to thank my mentor Prof. Hasan Fatih Uğurdağ, for providing me with the opportunity to conduct this research and for the support he provided throughout the process. Prof. Uğurdağ always encouraged me whenever I felt overwhelmed and tired; I could not imagine having a better mentor. I would like to thank the rest of my thesis committee: Prof. Bahadır K. Güntürk and Assoc. Prof. Dionysis Goularas for their encouragement and insightful comments. I would like to express my deepest gratitude to my two precious professors who have been like a milestone in my life: Asst. Prof. Hüseyin Oktay Altun and Assoc. Prof. Hulusi Açıkgöz. Their guidance, support, and mentorship have completely changed my life and opened new doors of opportunities for me. I will be forever grateful for their contributions to my personal and professional growth. Besides, I would love to thank my labmates in nEMESysLab at Özyeğin University for their support during these hard times. I am grateful for the several cups of coffee, coffee pickers, and long nights that contributed to the completion of this thesis.

I would like to extend a special thanks to the Founder of the Republic of Turkey, The Great Commander Mustafa Kemal Atatürk, for his emphasis on women's education in Turkey. His leadership and vision played a crucial role in shaping the modern Turkish state. His legacy continues to inspire future generations.

Finally, I would like to thank my beloved mother, dearest father, precious brother, and cherished sister for their love and endless support throughout my studies. Without them, this day would not have been possible. This thesis is funded by TÜBİTAK-1003 project. Project no: 118E891.

TABLE OF CONTENTS

DEDICATION	iii
ABSTRACT	iv
ÖZETÇE	v
ACKNOWLEDGEMENTS	vi
LIST OF TABLES	x
LIST OF FIGURES	xi
I INTRODUCTION	1
1.1 Problem Statement	1
1.2 What is Wide Area Aerial Surveillance?	2
1.3 Image Processing Modules	4
1.4 Why On-Device?	6
1.5 GPU-Based Acceleration	6
1.6 Contributions of the Thesis	7
1.7 Overview of the Thesis	8
II RELATED WORK	9
2.1 History of Wide Area Motion Imagery	9
2.1.1 Real Time Image Stitching	10
2.1.2 Deep Neural Network Architectures	11
2.2 Acceleration Methods	11
2.2.1 GPU Acceleration	11
2.2.2 Multi-core Programming	12
2.2.3 Deep Neural Network Optimizations	12
III REAL-TIME IMAGE STITCHING	14
3.1 Implementation of the Image Stitching Algorithm	14
3.1.1 Image Pre-Processor	15
3.1.2 Feature Matcher	15
3.1.3 Mosaic Image Stitcher	18

3.1.4	Profiling the Single-Threaded Algorithm	19
3.2	Acceleration Methods	20
3.2.1	GPU Acceleration	20
3.2.1.1	OpenCV with GPU Support	20
3.2.1.2	Offloading the Bottleneck Functions to the GPU	21
3.2.2	Maximizing CPU Utilization using Multi-core Programming	24
3.2.2.1	Python Implementation	25
3.2.2.2	C++ Implementation	31
3.3	Results and Comparison	34
IV	DEEP LEARNING MODEL OPTIMIZATIONS	39
4.1	Object Detection with YOLO Model	40
4.1.1	Multi-core Programming on YOLO	40
4.1.2	Model Compression and TensorRT Optimizations	45
4.2	Object Tracking with HM-Net Model	47
4.3	Super-Resolution	49
4.3.1	Optical Flow Module (IFNet)	50
4.3.2	Warping	51
4.3.3	ContextNet	51
4.3.4	FusionNet	51
4.4	Results and Comparison	51
V	EVALUATION OF OVERALL SYSTEM ARCHITECTURE	59
5.1	Resources	59
5.1.1	Devices and Environments	60
5.1.2	Data Set	62
5.2	How the Timing Measured?	63
5.3	Overall System Architecture	64
5.3.1	Multithreaded Overall System	66
5.3.2	Graphical User Interface for WAAS System	67
5.4	Experimental Results	67
5.5	Comparing with Other Works	74

VI CONCLUSION	76
REFERENCES	77
VITA	82



LIST OF TABLES

1	Profiling results of base code	19
2	Comparison of bottleneck functions for 100 images	22
3	Comparison of CPU vs GPU of single thread ISA	24
4	Event objects and their corresponding operations	26
5	Threads and frames to read	33
6	Comparison of single threaded ISA running on CPU and GPU . . .	35
7	Speedup comparison of ISA implementations [1]	35
8	Elapsed time of ISA implementations on Xavier	35
9	Comparison of implementations of object detection	44
10	Comparison of ISA implementations with object detection	45
11	The result of different pruning type on YoloV3	52
12	The performance of pruned Yolo	52
13	The performance of Yolo pruned with FP32	52
14	The performance of Yolo pruned with FP16	52
15	The object tracking result of original and pruned HM-Net Model . .	53
16	The optimization results of speed and model size of HM-Net model	54
17	The object tracking result of original and pruned HM-Net model . .	56
18	The optimization results of HM-Net model	57
19	The optimization results of super-resolution model	58
20	Technical specifications of NVIDIA Jetson AGX Xavier	61
21	Technical specifications of workstation	61
22	The comparisons of serial and multithreaded overall system	69
23	The comparisons of serial and multithreaded overall system	70
24	The comparisons of GoPro Dataset and Small Dataset	72

LIST OF FIGURES

1	Video surveillance modules in the system	3
2	Single threaded image stitching algorithm	15
3	The flow of upload-download operation	22
4	Three function offloaded to GPU	23
5	Comparison of single thread, multithreading and multiprocessing	25
6	The image buffers	26
7	The schematic of image pre-processor module	27
8	The schematic of feature matcher module	28
9	The schematic of mosaic image stitcher module	30
10	The schematic of multi-core implementation of ISA	31
11	The schematic of C++ implementation of ISA	32
12	Samples of the MAMI data set [2]	36
13	Intermediate mosaic images at 1st and 5th iteration	37
14	Intermediate mosaic images at 10th and 20th iterations	37
15	Intermediate mosaic images at 50th and 70th iterations	38
16	Final mosaic image	38
17	The sequential flow of object detection	40
18	Outputs obtained from the object detection	41
19	The schematic of multithreaded object detection module	42
20	The schematic of sequential object detection module with ISA	44
21	The schematic of multithreaded object detection module with ISA	45
22	The flow of the model compression process	46
23	The input-output diagram of HM-Net	48
24	The architecture of super-resolution module	50
25	Samples of the VisDrone dataset [3]	57
26	NVIDIA Jetson AGX Xavier	60
27	Samples of the WPAFB 2009 Data set [4]	62
28	The most active functions in the C++ implementation	64
29	The overall system on UAV	65

30	The overall system processing flow	65
31	The multithreaded overall system architecture	67
32	The graphical user interface	68
33	Sample of the WPAFB 2009 Data set [4]	69
34	Registered WPAFB 2009 images	70
35	Cropped area of interest	71
36	The super-resolution output for frame number 85	71
37	Tracking results of cropped area	72
38	Tracking results of cropped area of GoPro Dataset	73
39	Tracking results of cropped area of Small Dataset	73
40	The mosaic image produced using small dataset	74

CHAPTER I

INTRODUCTION

Wide area aerial surveillance is a term that describes an automated monitoring procedure that includes data collection, analysis, and interpretation to analyze object behaviors. *WAAS (wide area aerial surveillance)* system is primarily operated for the military, law enforcement, transportation, illegal fishing, monitoring in narrow waterways, etc. The term “wide area” may refer to an area of several square kilometers to several hundred nautical miles. The wide area aerial surveillance systems contain several cameras and processing units embedded into an airborne vehicle. This thesis introduces a GPU-based embedded system designed for wide-area motion imagery (WAMI) that processes and analyzes enormous amounts of aerial image data in real-time. The system contains several modules, including image matching/stitching, object detection/tracking, and super-resolution modules.

1.1 Problem Statement

Wide-area surveillance systems have grown in popularity in recent years for critical applications such as military, terrorism attacks, border protection, traffic control, and municipal security. Stitching the frames from multiple cameras gives a panoramic image of the area scanned by the Unmanned Aerial Vehicle (UAV). Thanks to the analysis of panoramic image sequences, events such as terrorist attacks and traffic collisions can be detected in real-time and possibly prevented. In order to design such a system that works in real-time, the modules used in the system must be able to process high amounts of high-resolution video frames at a high frame rate. The main motivation of this thesis is to design a compact, inexpensive and smart embedded system that processes a large amount of data in real-time for aerial imagery with advanced image and video analytical capabilities.

The proposed system tracks and monitors a wide area on ground using a Pan-Tilt-Zoom (PTZ) camera for scanning the area and a wide Field-of-View (FOV) camera. Since the focal length of the PTZ camera is long, the Ground Sampling Distance (GSD) has a lower value. GSD is defined as the geographical distance on the ground between two adjacent pixels on the image. [5] The lower value of GSD means more details are visible in the image. The GSD of the FOV camera has a higher value. Throughout this thesis, the wide-angle camera (FOV camera), is set to have a low-resolution, which means it is a low ground resolution, and the PTZ camera is set to have a high-resolution. The PTZ camera has a lower refresh rate because it scans the Area of Interest (AOI) during the surveillance. NVIDIA Jetson AGX Xavier is used as a GPU-based embedded system on unmanned aerial vehicles. The data collected with the cameras are processed on Xavier. Image stitching, object detection and tracking, and super-resolution modules are designed to run in real-time on Xavier. Since Xavier has a powerful GPU card, the modules are accelerated using GPU. In addition, multi-core programming techniques have been applied so that the modules can both accelerate and work as a single system. The deep learning architectures used in the system have also been accelerated by various optimization methods. Thus a real-time working system has been obtained.

1.2 What is Wide Area Aerial Surveillance?

Wide Area Aerial Surveillance, also known as *WAAS*, is a surveillance technology that allows for the tracking and monitoring of wide areas from an aerial platform. WAAS systems are often used for military and law enforcement applications such as tracking vehicles and people on the ground. Wide area aerial surveillance is accomplished by using multiple cameras or sensors typically mounted on an airborne vehicle. WAAS system can scan an area of several square kilometers at once. The biggest advantage of WAAS is its ability to monitor large areas in real-time.

The WAAS system basically involves several key steps to identify helpful information from the collected data. These processes may be image registration and stitching, object detection, and tracking, and super-resolution. Also, the WAAS system may include tools to visualize the information for the user. The video surveillance modules implemented in the WAAS system studied in this thesis can be seen in Figure 1.

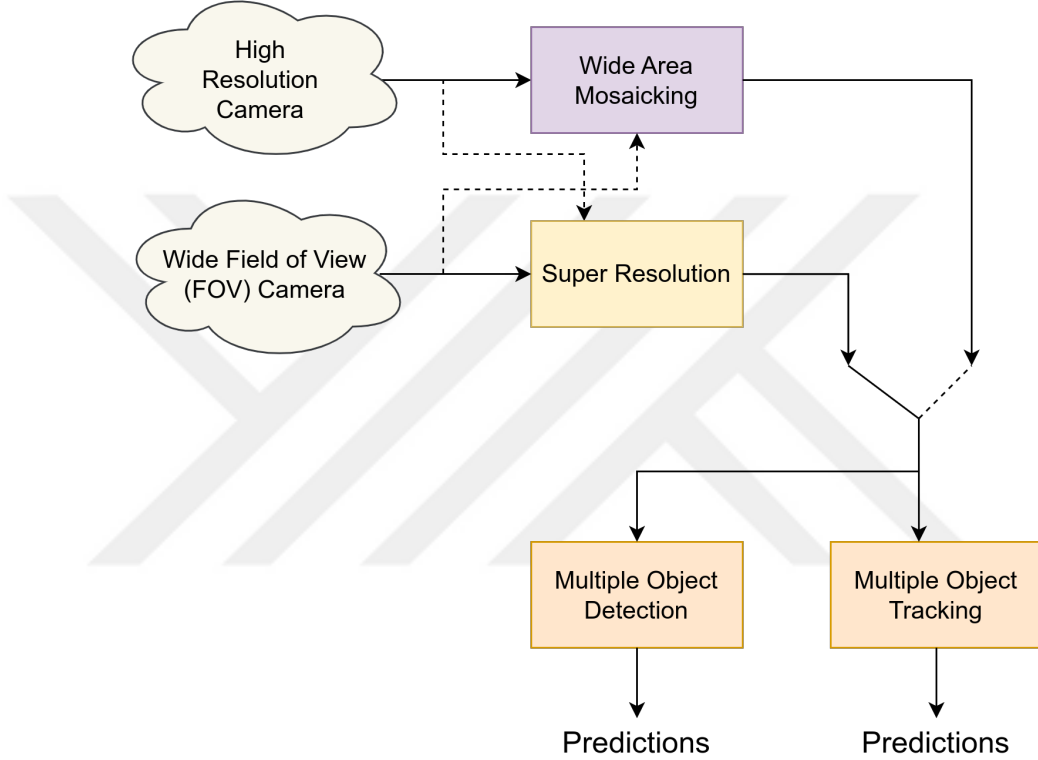


Figure 1: Video surveillance modules in the system

The proposed system shown in Figure 1 shows that two cameras are used to capture the frames. The *Wide Area Mosaicking* blocks, matches, and stitches the high-resolution frames and registers both high and low-resolution images together. The control center sends the coordinates to the UAV platform. Based on these coordinates and image registration results, the Area of Interest (AOI) is determined and cropped from both cameras. The super-resolution block uses low-resolution frames and high-resolution reference frame to produce a high-resolution image for AOI. Multiple Object Tracking and Multiple Object detection modules take

high-resolution frames.

1.3 Image Processing Modules

The GPU-Based Embedded System is an embedded system design developed for the video surveillance modules in WAAS. The system includes several modules. The modules implemented for the system can be listed in four main headings:

- Image Matching/Stitching Module
- Object Detection Module
- Object Tracking Module
- Super-resolution Module

The image stitching module is responsible for producing a single, seamless panoramic image, also known as a mosaic image, combining multiple images. This process can be done by registering the images and blending them together to create a panoramic image that shows the larger area. In computer vision applications, image stitching is frequently used to produce panoramic photos or high-resolution images from several sources. There are several steps involved in the image stitching process. First, images need to be aligned to be stitched together. This alignment can be done by identifying and extracting the corresponding points in each image by the feature-extracting algorithms. Then, the images must be stitched together to create the mosaic image. This can be done by stitching the images from overlapping regions of the images. Overall, image stitching is a powerful tool that can be used to create an image for wide area surveillance.

The object detection module is responsible for running the neural network to detect the object in the images. Object detection is basically a computer vision technique for detecting objects in images or videos. It involves a machine learning algorithm to analyze the frames. It detects the instance of objects from predefined classes, such as cars, trucks, trees, or pedestrians. Object detection is used in many

fields, including security, monitoring, and surveillance. It can be used to identify the objects in the video frames in real-time.

The object tracking module is a key part of this thesis. Object tracking and monitoring are used in many applications of WAAS systems. By using the object tracking module in the WAAS, the system can automatically detect and track the movements of objects on the ground. Detecting and tracking objects and providing information about their movements in real-time is important because it allows us to take quick and efficient precautions in vital events. Overall, the ability of WAAS systems to give real-time information about the movements of objects on the ground and to support various kinds of analysis and applications makes object tracking and monitoring a critical part of many WAAS technology applications.

Super-resolution module is a method used to increase the resolution of an image or video. [6] The super-resolution module is responsible for producing images that have higher resolution. The super-resolution algorithm combines multiple low-resolution and reference high-resolution frames to produce a high-resolution frame. Our module, super-resolution, takes two frames as input: one low-resolution image at a given t time and previous high-resolution image as a reference. The frame rate of the low-resolution camera is assumed to be two times faster than the high-resolution camera. The super-resolution module produces a high-resolution image for at a given t time. The super-resolution is important in the WAAS system for several reasons. First, in the proposed WAAS system, two different cameras capture images over a wide area. The super-resolution module provides a more detailed and accurate view of the scanned area with a higher frame rate. Also, super-resolution can improve the performance of other modules, such as object detection and tracking used in the system. Using high-resolution images in the object detection and tracking modules provides more accurate information about detected objects. Overall, in the WAAS system designed in this thesis,

super-resolution is an important module because it produces high-resolution images and enhances the functionality of other modules.

1.4 *Why On-Device?*

WAAS system needs to be on-device and working in real-time. The first reason is that WAAS systems are mainly used for surveillance and tracking. Thus, real-time information about what is happening on the ground is essential. The ground control center can react quickly to any threats or incidents if the system works in real-time.

Another reason for the requirement for an on-device and real-time system is that WAMI captures and processes enormous amounts of data from cameras or sensors. It is crucial to have significant computational power, which may only be available if the data is sent to a central server for processing. By having the necessarily embedded system on-device, such as Xavier, the system can analyze and process the data efficiently and provide real-time information to the ground control center.

1.5 *GPU-Based Acceleration*

In the *Wide Area Aerial Surveillance (WAAS)* applications, it is essential to achieve acceleration in terms of processing time to run the applications in real-time. This section answers why GPU-Based Acceleration techniques were used for the video surveillance modules in WAAS and why we need them.

GPU-based acceleration refers to the use of Graphics Processing Units (GPU) to accelerate the processing of data in the WAAS system. A GPU is a specialized type of processor designed to perform the applications that need high computing power, such as rendering graphics, video, and other types of visual data. [7] The WAAS system designed in this thesis can offload some of the computational tasks involved in processing large amounts of data; using GPU allows it to operate more efficiently and in real-time.

Moreover, in the WAAS system, various modules such as image and video

processing, object detection and tracking, super-resolution, and other types of analysis can be accelerated using the GPU. These modules include detecting and identifying objects in the UAV's images, tracking the motion of the objects during scanning time, and producing information about the objects and their movements. The system can process the data more quickly and accurately using GPU on these tasks, allowing the central control center to respond in real-time. In conclusion, applying GPU-based acceleration techniques to the WAAS system can improve the system's performance, reliability, and accuracy.

1.6 Contributions of the Thesis

In this thesis, for video surveillance in the WAAS system, different modules are implemented. The image processing modules are accelerated in order to create an overall system that runs in real-time. The wide area aerial surveillance system is developed for NVIDIA Jetson AGX Xavier, using Python programming language. The modules in the system are accelerated using two different approaches: GPU acceleration and multi-core programming. The design includes image stitching modules to register frames captured by scanning camera and wide field of view camera. The other module implemented for the system is super-resolution module to produce high-resolution frames with a high frame rate. Finally, to detect and track moving objects over the area being monitored, the object detection and object tracking modules are implemented and accelerated.

The image matching/stitching module computes the keypoints and the features of the frames and registers the frames into a wide field of view frame. The coordinates are going to use cropping the coverage area in the frames is calculated using an image stitching module.

The super-resolution module produces a high-resolution frame at a given t time. As a result of this module, the system can process the frames with high-resolution and a high frame rate. Then, using deep neural network algorithms, the moving objects in the area being monitored are detected and tracked.

The main contributions to the literature are as follows:

- Tracking moving objects on high-resolution video frames in real-time
- Designing a compact and inexpensive a wide area surveillance system
- Accelerating the modules taking advantage of the GPU on the Xavier

1.7 Overview of the Thesis

The rest of the thesis is organized as follows. The literature survey and the history of the wide area aerial surveillance are presented in Chapter 2. The modules designed in the system and acceleration methods employed in the design modules will be examined in detail.

Chapter 3 will provide an explanation of the real-time image stitching methodology and the acceleration techniques suggested in this study. These techniques are GPU based acceleration and multi-core programming. The acceleration results that were achieved in this study will be discussed.

Deep learning models used in the WAS system will be examined in Chapter 4. The optimization techniques implemented in those modules and their results will be examined in detail. The modules in the system are as follows: object detection and tracking and super-resolution.

Chapter 5 evaluates the overall system architecture and the modules running in real-time. The resources such as devices, programming environments, and data set used in this study, time-measuring methodology, graphical user interface, overall system architecture, and its multithreaded version will be discussed. Also, Chapter 5 includes the experimental results and comparison with other works.

Finally, in Chapter 6, this thesis will be concluded, and the contribution of this study to the literature will be presented.

CHAPTER II

RELATED WORK

Wide Area Aerial Surveillance (WAAS) is an enormous topic with various application areas, especially in the military. For this topic, there are many previous works, and academic research has been done. In addition, Wide Area Motion imagery (WAMI) includes diverse computer vision and deep learning implementations. The first WAMI implementation [8] came on the stage in the early 2000s and has drawn the attention of many researchers in recent years. This chapter will discuss the history of Wide Area Motion Imagery and its applications in the literature.

2.1 History of Wide Area Motion Imagery

The notional sensor architecture and the methods utilized to use the gathered imagery are the first known applications of wide area motion imaging. The first WAMI System is developed in the beginning of 2000s as part of the Sanoma Persistent Surveillance Program by a team with leadership of John Marion at Lawrence Livermore National Laboratory (LLNL). [8]. They took and analyzed a data set using two 11 Megapixel cameras and a high-quality Inertial Navigation System (INS), demonstrating WAMI's performance. For many researchers, this work represented the birth of WAMI. While WAMI systems was mostly developed for military use, it has also often found its way into civilian applications such as traffic analysis. The images gathering using helicopters are used to survey city traffic at the heaviest hours of the day.

Wide area aerial surveillance systems are known for processing and extracting information from a large amounts of data compared to other surveillance tasks. Wei et al. proposed a method that stitches images in order to create a panoramic view and moving object tracking on small UAV platform in 2017. [9] In 2019,

Zhou et al. proposed a patrol system based on panoramic stitching and object detection unmanned aerial vehicles. [10] In 2005, Heikkilä et al. published a fully automatic image mosaicking method for wide area surveillance systems. [11]

2.1.1 Real Time Image Stitching

Image stitching combines multiple images to produce a panoramic image, also known as mosaic image, to capture wide area with high-resolution. The image obtained from the algorithm is a wide angle mosaic image. It is constructed using a set of images captured by a drone.

In order to register the frames to each other the features and keypoints need to be extracted and matched. There are two main approaches for image stitching techniques: direct and feature based techniques. Direct approach is based on comparing all the pixels and their intensities of the frames with each other while feature based approach is identify the similar points on frames and matches them. There are many feature detector algorithms, such as Harris [12], SIFT [13], SURF [14], FAST [15], and ORB [16] algorithms.

Because of the large amount of Unmanned Aerial Vehicle (UAV) data, it has become essential to speed up the algorithm so it can work in real-time. Researchers have been focused on new techniques to solve this problem. Most researchers have used GPU to speed up the image stitching algorithm. Lin et al. (2018) proposed a method [17] that uses the CUDA toolkit to accelerate the high-resolution image stitching algorithm to acquire a better initial guess. The article published by Nakov et al. [18] discussed parallel GPU implementation of image stitching algorithm based on NVIDIA CUDA. In contrast, Xu et al. [19] discussed a hybrid approach that uses both CPU and GPU to speed up the matching operation. Lai and Huang (2016) proposed an acceleration obtained by a hybrid implementation on computers with GPU. [20]

2.1.2 Deep Neural Network Architectures

In the last decade, deep neural networks have become common in various computer vision [21] applications such as image classification [22], object detection [23] and tracking [24], and natural language processing [25]. As a result of these developments, the data used as an input and the neural network architectures are extensive. Due to this large-scale data set and new architectures, acceleration and compression have become necessary. Different approaches are developed to satisfy this requirement, including pruning and model compression.

2.2 Acceleration Methods

Since in WAAS systems monitoring geographical areas in real-time is essential, there are numerous research about acceleration methods has been done. These methods mainly GPU-based acceleration [26] and multi-core programming [27] techniques.

2.2.1 GPU Acceleration

Due to the requirement of processing a large amount of data in Wide Area Motion Imagery (WAMI), minimal execution time in the WAAS applications has become essential. From the beginning of the GPUs produced, GPUs have been used particularly in the computer vision domain. [28] Currently, CPUs are referred to as *multi-core* processors while GPUs are *many-core* processors. GPUs are often used in applications such as computer graphics, computational physics, computational chemistry, signal processing, etc.

With the development of GPUs, their use in academic research and the civilian field has become quite common. It has become much preferred, mainly in computer vision, where processing time is essential. The first work in history, called image processing with GPU, was published in 2000 by M. Hopf and T. Ertl. [29] The open-source Digital Image Processing libraries became available in 2003. [30] Colantoni et al. in color image processing used GPU programming. The library

called *OpenVidia* [31] was proposed by Fung et al. to use single or multiple graphics cards to accelerate image processing and computer vision implementation in 2005. Strzodka and Telea (2004) proposed a bi-dimensional distance transform with GPU acceleration in [32].

2.2.2 Multi-core Programming

Multi-core programming decreases the computation time in systems by executing instructions concurrently. It is deployed on multiprocessor platforms. In image processing and computer vision applications, since the execution time is essential, new techniques to satisfy these requirements have been developed by researchers in recent years. Saxena et al. (2013) published a study [33] that shows image processing tasks in medical imaging and its acceleration using parallel computing in multi-core architecture. In a study by Kim et al. (2011) [34] uses multi-core CPUs with Intel parallel programming techniques to implement on 2D convolution algorithm that needs high-performance computing power. A study published in 2019 by Georgis et al. [35] discussed acceleration techniques on multi-core CPU, GPU, and FPGA for super-resolution implementation.

Zhao et al. (2009) [36] proposed an image processing system that runs in real-time based on a multi-core processor. Wang et al. (2017) [37] presented a similar method using multi-core programming to extract features for aerial image stitching uses the ORB algorithm. Blattner et al. developed a hybrid CPU-GPU system of image stitching implementation for microscopic images. [38] Xu et al. [19] presents stereo-matching acceleration with two different techniques using two architecture parallelism enabled, such as multi-core programming and GPU-based acceleration.

2.2.3 Deep Neural Network Optimizations

Yu et al. (2021) [39] proposed a Hessian-Aware pruning method to reduce the memory footprint of the network and reduce the FLOPs (Floating Point Operations) related to the network. Liu et al. in their study [40], developed a novel

system to reduce the model’s size, decrease the memory footprint and lower the number of computing operations. In the study by Lym et al. [41], a novel pruning method called PruneTrain is developed by dynamic sparse model reconfiguration, and altogether they achieved a reduction of 39% in terms of training time.



CHAPTER III

REAL-TIME IMAGE STITCHING

This chapter will discuss the implementation of the real-time image stitching algorithm (*ISA*) and its acceleration methods. The design includes three acceleration methods: GPU offloads of the bottleneck functions to maximize CPU utilization, multithreading, and multiprocessing parallelization techniques. The GPU offloads of the bottleneck functions are a way to run bottleneck functions of the stitching algorithm on a GPU. Thus with the high-speed computational power of the GPU, we gain significant acceleration. In addition, to achieve even more acceleration, CPU utilization has been further improved. In order to do this, multithreading and multiprocessing techniques have been developed and implemented. The algorithm achieved real-time speed with the combination of the GPU offload and CPU maximization.

The design overview of the image stitching algorithm in this thesis will be explained in Section 3.1. Then, in Section 3.2, acceleration methods used in the implementation will be explained. Their respective subsection will explain the GPU acceleration and maximizing CPU utilization using multi-core programming.

3.1 Implementation of the Image Stitching Algorithm

The image stitching algorithm is an iterative algorithm to produce mosaic images. This section explains the image stitching algorithm implementation and clearly defines one iteration that runs several times on the system in detail. The stitching algorithm continuously takes a new image as input, adjusts the image and sends it to the feature matcher module, and produces an intermediate mosaic image which will be used in the next iteration. In Figure 2, the overall single-threaded stitching algorithm can be seen. This process continues until all the images are processed and the final mosaic image is produced. The *Mosaic Image* grows cumulatively

through all loops. Each module is connected to the next, and their output data is taken as input to other modules.

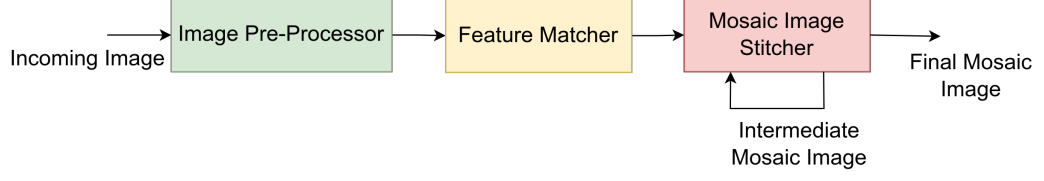


Figure 2: Single threaded image stitching algorithm

The image stitching algorithm consists of three main modules: *Image Pre-Processor*, *Feature Matcher*, and *Mosaic Image Stitcher* modules. Each of the modules performs its operations at every iteration during the process. In this section, these three main modules will be explained in detail.

3.1.1 Image Pre-Processor

The purpose of the *Image Pre-Processor* module is to prepare the image for operations in the *Feature Matcher* module. The *Image Pre-Processor* module for a single frame can be seen in Algorithm 1. The *Image Pre-Processor* module reads one image at a time and prepares it for further processing. Inside the module, there is the *imread* function of the *OpenCV* library is used to reading frame, *resize* function that scales images by half size, and the *cvtColor* function that converts colors from RGB to gray scale. The original frames are resized down by a scaled factor S . Since processing an image with a smaller size and gray scale color is faster, these operations are done in this module.

Algorithm 1 Image Pre-Processor Module

- | | |
|---|---|
| 1: ii
2: $img \leftarrow \text{imread}(imagePaths[ii])$
3: $img \leftarrow \text{resize}(img, S)$
4: $imgGray \leftarrow \text{cvtColor}(img, RGB2GRAY)$ | $\triangleright$ Frame number

$\triangleright$ Scaled down by S
$\triangleright$ Color convert, RGB to gray |
|---|---|
-

3.1.2 Feature Matcher

In image processing algorithms, feature matching is used for identifying similar or corresponding features in two or more images. The goal of feature matching

is to find corresponding regions between images. This can be used for implementations such as image stitching, registering, and aligning. There are several standard algorithms to be used for extractor and detector such as *Corner Detection*, *Scale Invariant Feature Transform (SIFT)*, *Speed-Up Robust Features and Oriented FAST and Rotated BRIEF (ORB)*. Since reducing the elapsed time for processing one frame is the primary aim in this part of the thesis, the *ORB* algorithm is used as a keypoint detector and descriptor extractor. ORB is a binary descriptor that is based on the FAST corner detector and the BRIEF descriptor. It is designed to be efficient implementation.

Feature matching is an important task in image stitching and registering. To stitch two images together, keypoints and features for both images need to be identified. Once these keypoints and features have been identified, they can be used to stitch and blend images together. The purpose of *Feature Matcher* module is to detect keypoints between the current image, which is *Intermediate Mosaic Image*, and the incoming input image. After that, using these key points, the features for both images are extracted by the feature extractor function. Once the module retrieves images, *detectAndDescribe* method is called. This method takes an image as input, detects keypoints, and extracts features from the input image.

The proposed WAAS system uses ORB (Oriented FAST and Rotated BRIEF) as a feature extractor and descriptor method. The ORB algorithm is one of the fastest feature detection algorithms in computer vision tasks. It extracts keypoints from an image and describes them using binary strings, making it a fast and efficient feature extractor for real-time applications. ORB also utilizes a multi-scale and rotation-invariant approach to improve the efficiency and accuracy of feature detection. ORB was proposed in 2011 by Rublee et al. [42].

To match the keypoints and features found by ORB, KNN (K-Nearest Neighbour) algorithm is used. The KNN algorithm is a widely used algorithm in image processing and computer vision applications. The KNN algorithm is based on the

idea of finding the k nearest neighbor of a feature in one image and then determining if the same feature exists in the consecutive image by finding the closest match. The features extracted using ORB from both images are compared and using the Euclidean distance, the distance between the two features is calculated. Afterward, for each feature in one image, the KNN algorithm finds the k nearest neighbors in the other image and calculates the distance between them. In order to filter raw matches, a threshold is applied, and it filters out matches that are too far. Thus, the closest match is chosen as the corresponding feature in the other image. In addition, RANSAC (Random Sample Consensus) parameter is used to eliminate false matches. This process is repeated for all features in the first image to find matches between two consecutive images. [43] The simplest version of the *Feature Matcher* module designed in the system can be seen in Algorithm 2.

Algorithm 2 Feature Matcher Module

```

1:  $ii$  ▷ Frame Number
2: if  $ii == 0$  then
3:    $keypointsB, featuresB \leftarrow \text{detectAndDescribe}(prevImage)$ 
4:   continue;
5: else
6:    $keypointsA, featuresA \leftarrow \text{detectAndDescribe}(queryImage)$ 
7: end if
8:  $rawMatches \leftarrow \text{matchKeypoints}(featuresA, featuresB)$ 
9:  $goodMatches \leftarrow \text{findGoodMatches}(rawMatches)$ 
10:  $pointsA, pointsB \leftarrow \text{findPoints}(keypoints, goodMatches)$ 
11:  $homography \leftarrow \text{findHomography}(pointsA, pointsB, RANSAC)$ 

```

At the first iteration, the *Feature Matcher* module computes the keypoints and features of the first frame, which is called $keypointsB$ and $featuresB$. After the first iteration, it computes the new incoming frame's keypoints and features and stores them to $keypointsA$ and $featuresA$ until the last frame. B and A representing these data belong to the first and second images, respectively. At the end of every iteration, $featuresA$ and $keypointsA$ values are copied to the $keypointsB$ and $featuresB$. Afterward, $knnMatch$ function matches the features between two frames and finds how many features from both images match each other. Lastly, $findHomography$ function takes keypoints and matches them as input arguments

and computes a homography matrix. This calculated homography matrix is sent to the *Mosaic Image Stitcher* module along with the two images.

3.1.3 Mosaic Image Stitcher

The purpose of *Mosaic Image Stitcher* module is to stitch intermediate mosaic image and incoming image together. In this module, mosaic image grows cumulatively. The simplest version of the module can be seen in Algorithm 3. In the first iteration, *mosaicImage* stores the first input image, which is used as a starting frame for the stitching process. The *queryImage* variable represents the current image. There is no stitching process at the first iteration. Nevertheless, after the first iteration, the module performs operations at every iteration until the last frame.

Algorithm 3 Mosaic Image Stitcher Module

```

1: ii ▷ Frame number
2: if ii == 1 then
3:   mosaicImg  $\leftarrow$  queryImage
4: else
5:   w, h  $\leftarrow$  calculateSize(mosaicImage, queryImage)
6:   H  $\leftarrow$  dotProduct(H_prev, H)
7:   result  $\leftarrow$  warpPerspective(H, queryImage, w, h) ▷ Current warped img
8:   mosaicImage  $\leftarrow$  padding(mosaicImage)
9:   mosaicImage  $\leftarrow$  copy(result)
10:  H_prev  $\leftarrow$  H
11:  keypointsB  $\leftarrow$  keypointsA
12:  featuresB  $\leftarrow$  featuresA
13: end if

```

As seen in Algorithm 3, the module calculates the new width and height for the mosaic image. Then, it calculates the dot product of the previous homography matrix (*H_prev*) and the current homography matrix *H*. For the first frame, the homography matrix is used as an identity matrix with a size of 3x3. After that, it computes the perspective transform *result* using the *warpPerspective* function, which takes *queryImage*, *homography*, *width* and *height* as input arguments. Then, the intermediate *mosaicImage* is padded with zeros and so creates enough space to be able to fit the warped image (*result*) on it. After copying the *result* to

mosaicImage, the stitching process for one iteration is completed. Finally, the *Homography* matrix, *featuresA*, and *keypointsA* computed at the current iteration are saved and transferred to the next iteration.

3.1.4 Profiling the Single-Threaded Algorithm

Since the goal of this part of the study is to make the stitching algorithm run in real-time, the most time consuming functions need to be identified. Even though the algorithm has various functions, only the most time-consuming functions are focused on being accelerated. To identify and analyze the current performance of the algorithm the code profiling method is applied to the base code. Code profiling is a method used for analyzing the performance of the functions in a code. Thus, the profiler produces a report showing how many times each method in the code is called and how long each of those methods takes. The code profiling results can be seen in Table 1. Only the first eight functions that take the most time are shown in Table 1. The time results are in seconds.

Table 1: Profiling results of base code

Function Name	Number of Calls	Time per Call	Total Time
imread	102	0.174	17.707
knnMatch	100	0.175	17.489
detectAndCompute	101	0.089	8.960
warpPerspective	97	0.066	6.350
findContours	195	0.01	1.858
Padding	97	0.016	1.156
Copy	292	0.004	1.116
cvtColor	296	0.003	0.835

Based on the results shown in Table 1, if the acceleration is achieved in these eight functions, a significant speed-up will be achieved in total. For this purpose, some acceleration methods are developed and implemented on the base code.

3.2 Acceleration Methods

After identifying the bottleneck functions in the ISA, it was aimed to accelerate these functions. In order to do this, two different approaches are proposed: offloading bottleneck functions to the GPU and applying multi-core programming techniques to the ISA. Using a GPU for image processing is essential because GPUs are specifically designed for processing the large amounts of data and parallel computations required for applications such as. GPU has a large number of cores that are optimized for parallel operations, whereas CPU has limited cores. Additionally, multi-core programming is important because applications such as image stitching require high-speed data processing. Multi-core programming allows for better utilization of the available computing resources. In Section 3.2, these acceleration methods will be discussed and analyzed.

3.2.1 GPU Acceleration

Since the *GPU*'s has thousands of cores, image processing applications running on GPUs are faster than the ones running on CPUs. The proposed method in this thesis is a method that accelerated the image stitching algorithm by using the *CUDA* toolkit. *CUDA* [44] is a programming model and software environment that allows programmers to write parallel programs using a direct extension of the C language. *CUDA* only works with NVIDIA GPUs. OpenCV and *CUDA* have been used in image processing applications for more than ten years. However, the combined implementation of both is not so general [45]. There is a *GPU/CUDA* module for *OpenCV* library that has some of the built-in functions has the *GPU/CUDA* counterparts. These bottleneck functions are offloaded to the GPU using *CUDA* counterparts. To enable this module, the *OpenCV-contrib* library is built from the source, and the *CUDA* functions are activated.

3.2.1.1 OpenCV with GPU Support

OpenCV is an open-source library for computer vision applications. [46] It is written in C and C++ languages and it runs under various operating systems such as

Linux, Windows, and Mac OS X. OpenCV is developed to improve computational efficiency and is specially focused on real-time implementations [46]. Since in most scenarios *OpenCV* utilizes CPU, it does not always give desired performance. In order to solve this problem, a new module that supports GPU acceleration using CUDA is distributed. OpenCV GPU module is written in CUDA. Thus it takes advantage of CUDA’s computational power.

Although the library’s functionality is not covered, the module constantly expands and is being modified for new technologies and GPU architectures. The GPU module of OpenCV is not enabled by default; it has to be built from source using *OpenCV-contrib* repository. The main pre-requisite is to install the *NVIDIA CUDA toolkit* and the *NVIDIA CUDA Deep Neural Network library (CuDNN)*. To install the GPU module, the NVIDIA device architecture needs to be identified. In this thesis, *NVIDIA Jetson AGX Xavier* is used, whose compute capability is 7.2. If GPU architecture is not set correctly, the installation will fail. After preparing the environment and setting the flags for installation, the OpenCV library with GPU support is installed.

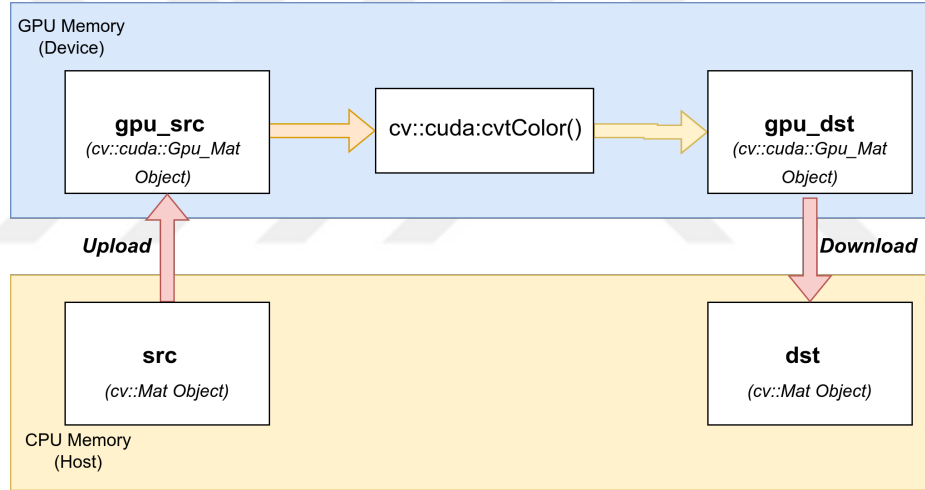
3.2.1.2 Offloading the Bottleneck Functions to the GPU

According to the profiling results, the four most time-consuming functions are *imread*, *knnMatch*, *detectAndCompute*, and *warpPerspective*. These are *OpenCV*’s built-in functions. Since these functions are the most time-consuming ones, accelerating them would increase the overall algorithm’s speed. Before offloading them to ISA, the functions are executed for a total of 100 images with the size of 1024x1024 to measure each elapsed time. The results for comparing bottleneck functions for 100 images can be seen in Table 2. The results in Table 2 refer to the time between the start and completion of each function. The time consumed when data is uploading/downloading from CPU to GPU memory is considered. As seen in the results, even though there is overhead from the upload/download operations, the GPU versions have significant speed up compared to their CPU versions.

Table 2: Comparison of bottleneck functions for 100 images

	detectAndCompute	knnMatch	warpPerspective
CPU	7.306s	6.351s	8.526
GPU	2.218s	2.935s	0.709
Speedup	3.29x	2.16x	12.05x

After the comparison, all bottleneck functions are replaced with their counterparts in the ISA. However, since the data must be moved between the CPU and GPU, there is an overhead because GPU cores work in a dedicated memory from the system memory. Therefore, images stitched together must be moved back and forth between the host (CPU) and device (GPU). The simplest version of processing an image using GPU is shown in the Figure 3.

**Figure 3:** The flow of upload-download operation

Firstly, only three functions were replaced with their GPU versions: *detectAndCompute*, *knnMatch*, and *warpPerspective*. The algorithm runs as a single thread, and it runs the *Image Pre-Processor*, *Feature Matcher*, and *Mosaic Image Stitcher* modules serially. However, those three functions are run on GPU; the remaining ones are run on CPU. The Figure 4 shows an implementation of single-thread ISA with three functions offloaded to GPU.

When using GPU functions, since the images processing on GPU cores, they

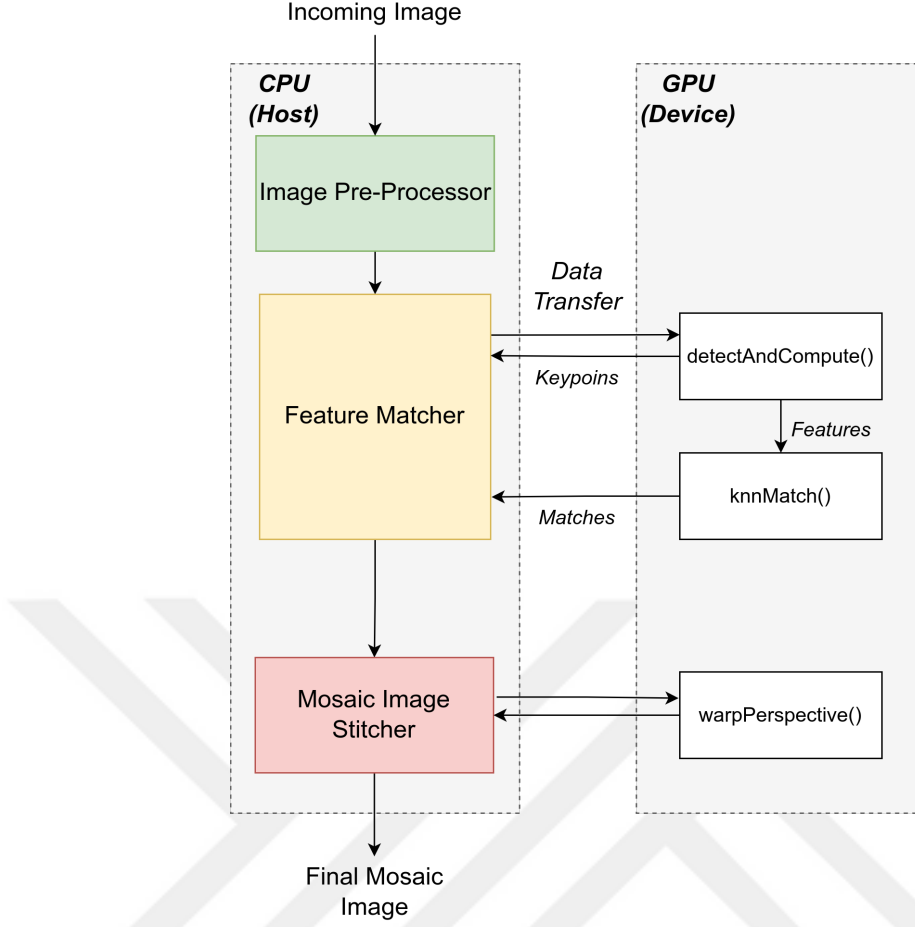


Figure 4: Three function offloaded to GPU

need to upload to the GPU’s memory. Although the GPU version of the single-threaded ISA implementation is significantly accelerated despite this overhead, upload/download time should be minimized. This problem is accomplished by replacing nearly all of the built-in *OpenCV* functions used in the method with their GPU equivalents. The image reading is the top of bottleneck functions. Since image reading starts on the CPU, it does not need to be replaced. Also, there is no CUDA version of *cv::imread* function.

After reading images from the camera, the frames are immediately uploaded to the GPU, and if there are no functions that run on GPU, they are not downloaded back. The comparison results of the single-threaded image stitching algorithm with functions offloaded to GPU and the single-thread image stitching algorithm running on the entire CPU can be seen in the Table 3. The results are measured for processing 100 images.

Table 3: Comparison of CPU vs GPU of single thread ISA

	Single Thread CPU	Single Thread GPU
Elapsed Time	31.517 seconds	23.717 seconds

The GPU acceleration on a single-threaded image stitching algorithm is a practical approach. However, the elapsed time during processing can be improved significantly by using multi-core programming techniques. The multi-core programming techniques give the benefits of utilizing the CPU, which will be discussed in the Section 3.2.2.

3.2.2 Maximazing CPU Utilization using Multi-core Programming

The ISA achieved significant speed up using the GPU acceleration technique described in Section 3.2.1. After that, the processing speed improved using multi-core programming techniques such as *multithreading* and *multiprocessing*. Multi-core programming techniques are developed to run tasks concurrently and achieve acceleration on an algorithm. The use of many CPU cores to carry out concurrent computations in a computer program is known as multi-core programming. A large computational task can be divided into smaller, independent tasks that can be done concurrently on several CPU cores utilizing multi-core programming. As a result, total performance of the system, may be enhanced by its ability to handle more data in the same amount of time. This approach aims to multitasking. Multitasking can be achieved in two ways: *Multithreading* and *Multiprocessing*. Multitasking is essential in image processing applications, especially in real-time applications. *Multithreading* is the ability of a processor to run numerous threads simultaneously, each of which is a separate process. In contrast, multiprocessing defines a system's capability to use many processors simultaneously, with each CPU able to manage one or more threads. The diagram in the Figure 5 shows the comparison of running applications on single processor single thread, single processor multithread, and multiprocessing. In this study, the image stitching algorithm

is implemented in two languages such as *Python* and *C++*, to compare the performance and choose the best implementation. The *Python* implementation will be discussed in Section 3.2.2.1, and the *C++* implementation will be discussed in Section 3.2.2.2.

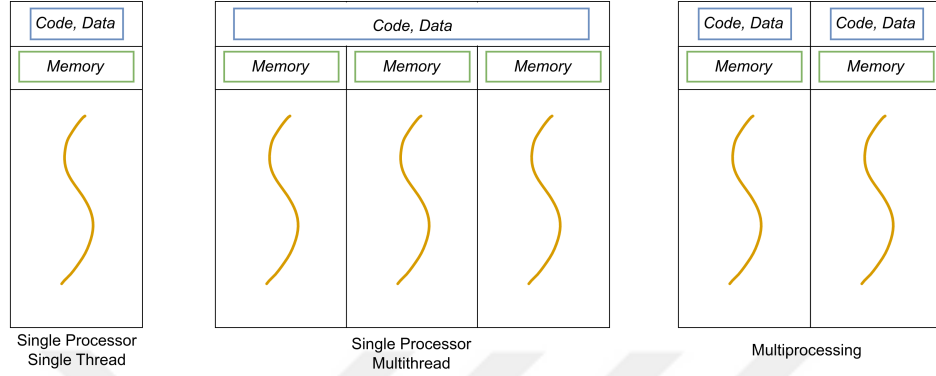


Figure 5: Comparison of single thread, multithreading and multiprocessing

Overall, the use of multi-core programming can be important for improving the performance of image processing and other applications that require high-speed data processing. By allowing for better utilization of the available computing resources, multi-core programming can enable systems to process data more quickly and efficiently.

3.2.2.1 *Python Implementation*

The CPU is more utilized using multi-core programming techniques but using multiple threads/processes comes with an issue related to the data transfer and synchronization of threads/processes. In the *Python* implementation of ISA, a 3-buffer ping-pong mechanism is proposed to solve the synchronization and data transfer problem. The buffer that stores three images can be seen in Figure 6. The image buffer seen in Figure 6 has three indexes with a fixed size of $W \times H$ (W : Weight of input image, H : Height of input image) pixels and is used to store three images. The *idx* shown in the diagram refers to the index of buffer, as in first in first out (FIFO). The image stitching starts the process of images in the buffer according to their indexes. Due to the quantity of started threads and processes, a total number of buffers are formed.

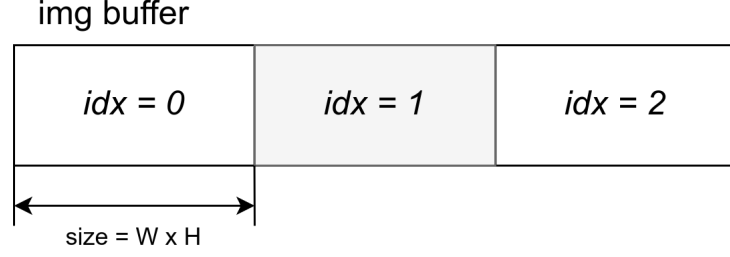


Figure 6: The image buffers

The images are stored until the ISA completes the stitching process, and buffers are ready to store new incoming frames when processing the previous frame is completed. Since the image stitching algorithm is designed sequential fashion, the buffers store the frames in order, $img[idx]$ where idx is 0, 1, and 2.

An event-based synchronization mechanism is created to keep the threads and processes in sync. Events are phenomena in an algorithm that allows threads and processes to communicate actions made by sending signals to one another. The events are created from the *threading* library for multithreading and *multiprocessing* library for multiprocessing in Python. An event has various calls, such as *set*, *wait*, and *clear*, which can be called from any thread at any time in the program. The event objects created in the Python implementations and their corresponding operations are shown in the Table 4.

Table 4: Event objects and their corresponding operations

Event Name	Description
r	Read operation completed
rN	Read operation continue
m	Match operation completed
mN	Match operation continue

The Python implementation of the image stitching algorithm is separated into three different threads/processes: *Image Pre-Processor Module*, *Feature Matching Module*, and *Mosaic Image Stitcher Module*. In order to accelerate the algorithm,

these three modules are assigned to different threads/processed for the multi-threading and multiprocessing approach. Due to the nature of the image stitching algorithm, these threads/processes should communicate with each other and share data. In order to solve this, an event mechanism is developed. The schematic of the image pre-processor module can be seen in Figure 7. It takes the *imagePath* as an input, reads the frames using the *imread* method of OpenCV, resizes the frame, and then changes the color channel to the gray scale. Adjusted frames are sent to the *Feature Matcher Module*.

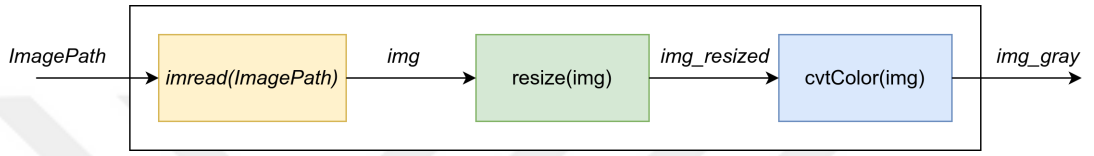


Figure 7: The schematic of image pre-processor module

The Algorithm 4 shows the synchronization mechanism proposed in the image pre-processor module.

Algorithm 4 Synchronization mechanism in the image pre-processor

- 1: $idx_r \leftarrow 0$
 - 2: $rN[idx_r].wait()$
 - 3: $image \leftarrow imread()$
 - 4: $img_gray \leftarrow.cvtColor(image)$
 - 5: $img[idx_r] \leftarrow resize(img_gray, S)$ ▷ Scaled down by S
 - 6: $r[idx_r].set()$
 - 7: $rN[idx_r].clear()$
 - 8: $idx_r \leftarrow (idx_r + 1) \% 3$
-

In the Algorithm 4, the index that chooses one buffer out of three buffers is referred to as idx_r . The N indicates the total number of images read by the module, which is also equivalent to the number of iterations that will run. The *Image Pre-Processor Module* waits for rN event objects to be *set* at every iteration. Later, a new frame is stored in the image buffer at the index idx_r . The img refers to an array of three buffers to store frames. After the frame is read, r event object calls *set* and rN calls *clear* methods, and idx_r is incremented by 1. The variable idx_r needs to be between 0 and 2 due to the requirement of

only three buffers and three events of each type. This procedure continues until all frames are received and the other threads/processes have finished their operations. The frames read and adjusted by the *Image Pre-Processor Module* are sent to the *Feature Matcher Module*. The schematic of the *Feature Matcher Module* is shown in the Figure 8.

As can be seen in the diagram, the gray scale frame sent by *Image Pre-Processor Module* is input to *Feature Matcher Module*. Initially, iteration is checked to see if it is the first one. If the incoming frame is the first frame, then the module only finds the frame's keypoints and features and stores them to the variables called *keyPointsB* and *featureB*. After the first iteration, the module computes keypoints and features of new frames. Then, using those outputs, it is computed the matches between frames. After computing *goodMatches* using *rawMatches*, it finds the coordinates of matches on frames. Then, it finds the *Homograph Matrix* with *findHomography* method using keypoints and *goodMatches* as input arguments. The *Homography Matrix* is sent to the *Mosaic Image Stitcher* thread in order to perform the stitching operation using homography.

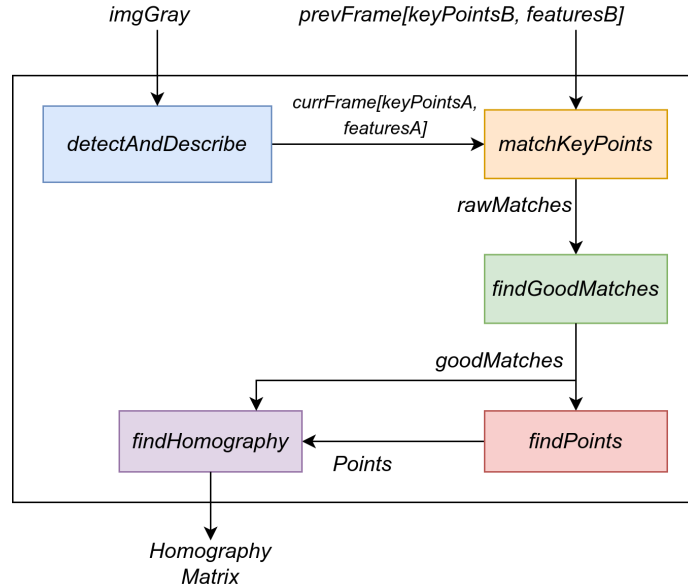


Figure 8: The schematic of feature matcher module

The synchronization mechanism created in the *Feature Matcher Module* can be seen in the Algorithm 5. A similar approach to the *Image Pre-Processor Module*

is developed for the synchronization mechanism. The second thread/process that contains *Feature Matcher Module* runs for N iterations and uses idx_m for the buffers and event object index. The variable idx_m is a local index that selects one buffer at a time between 0 and 2. At every iteration, the thread/process waits for mN and the corresponding index of image buffer r to be set. Then, it computes the match operations between two current frames and an intermediate mosaic frame. It uses the image at index idx_m as the current image. In the end, the event object called m is set, and the index idx_m is incremented by 1.

Algorithm 5 Synchronization mechanism in the feature matcher

```

1:  $idx\_m \leftarrow 0$ 
2:  $mN[idx\_m].wait()$ 
3:  $r[idx\_m].wait()$ 
4:  $homography \leftarrow match(img[idx\_r])$ 
5:  $Pipe \leftarrow homography$ 
6:  $m[idx\_m].set()$ 
7:  $idx\_m \leftarrow (idx\_m + 1) \% 3$ 

```

In the third thread/process the *Mosaic Image Stitcher Module* is executed. The schematic of the module is shown in the Figure 9. The thread/process is takes a set of data to use in the module. These data are as follows: *prevImg*, which is the intermediate mosaic image, *currentImg*, which is the new incoming frame to be stitched to the mosaic image, and the *homography matrix* to warp current frame properly. In the module, using homography matrix new incoming frame is warped and copied corresponding coordinates in the intermediate mosaic image. This process is running until all the frame are stitched and produces a whole panoramic image at the end. The number of iterations is equal to the number of the frame captured from the drone camera.

The synchronization mechanism used in the *Mosaic Image Stitcher* thread is shown in the Algorithm 6. In the last thread/process, as a variable to select buffers, idx_s is used. At every iteration, thread/process waits for m and r event objects related indexes to be set. When the m and r events are, *set* means the *Feature Matcher Module* is done its operations, and the frames are ready to

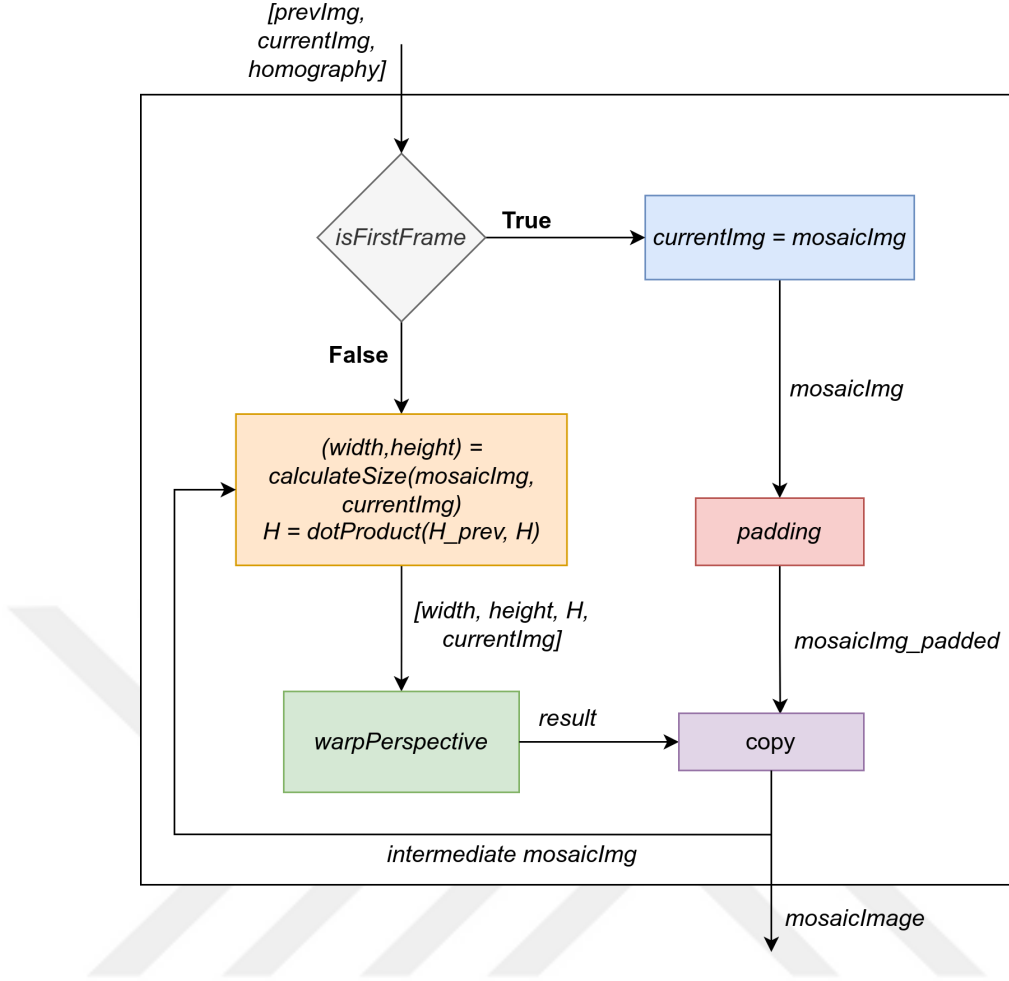


Figure 9: The schematic of mosaic image stitcher module

stitch. After the ready signals come from both *Image Pre-Processor Module* and *Feature Matcher Module*, the stitching operations are executed. Only images are manipulated in the *Mosaic Image Stitcher Module* throughout the entire stitching algorithm. The other modules only used the images to extract features and compute the necessary values. In other words, the image in the buffer at index idx_s is free, and a new frame can be stored at that index.

In this study, three threads/process is implemented for multithreading and multiprocessing. Overall implementation is shown in Figure 10. The modules are split into threads *Thread1/Process1*, *Thread2/Process2* and *Thread3/Process3* respectively. All threads/processes need to coordinate their tasks and communicate with each other. Otherwise, the threads/processes have become deadlocked, and the stitching algorithm could not be completed correctly. The synchronization

Algorithm 6 Synchronization mechanism in the image stitcher

```
1:  $idx\_s \leftarrow 0$ 
2:  $m[idx\_s].wait()$ 
3:  $r[idx\_s].wait()$ 
4:  $homography \leftarrow Pipe$ 
5:  $stitch(img[idx\_s])$ 
6:  $r[idx\_s].set()$ 
7:  $mN[idx\_s].set()$ 
8:  $m[idx\_s].clear()$ 
9:  $idx\_s \leftarrow (idx\_s + 1) \% 3$ 
```

events must be created carefully to prevent race conditions between threads/processes because each core computes its frequency. The *Image Pre-Processor Module* is enabled to write images to image buffers, whereas the *Feature Matcher* and *Image Stitcher* modules only read images from the image buffers.

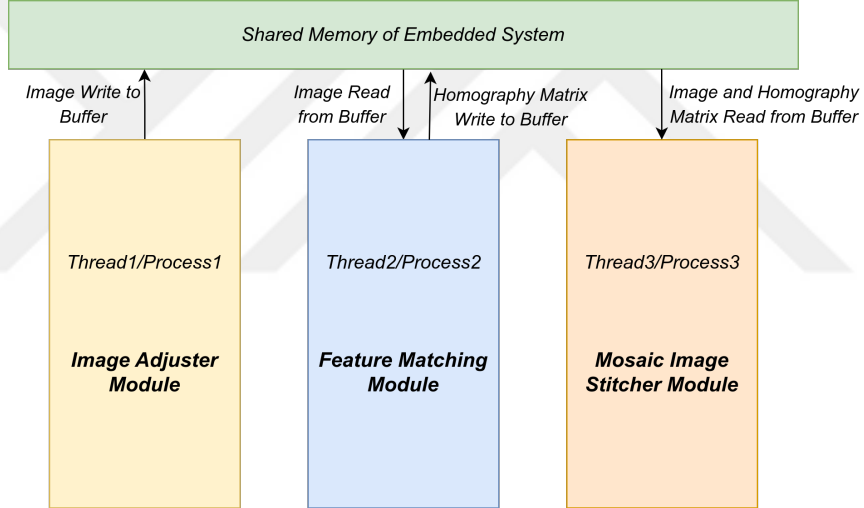


Figure 10: The schematic of multi-core implementation of ISA

3.2.2.2 C++ Implementation

The ISA written in *Python* language, is converted to the *C++* language. This version of the ISA uses the *C++* version *OpenCV*. It is compiled from the source to enable GPU support with *CUDA* functions. In *C++* implementation of the image stitching algorithm, a kind of producer-consumer problem has been adapted to solve the synchronization problem. This approach creates a queue for the images, and the queue starts to fill when the frames are started to read. While the queue is not empty, the process starts to read frames from the queue and continues to

process. If the queue is empty, there is no incoming frame, and the match and stitch threads wait. The *FrameQueue* is used in the *Image Read* thread, and *Image Read* threads fill the *Frame Queue*. The schematic of the C++ implementation of ISA using multithreading is shown in the Figure 11.

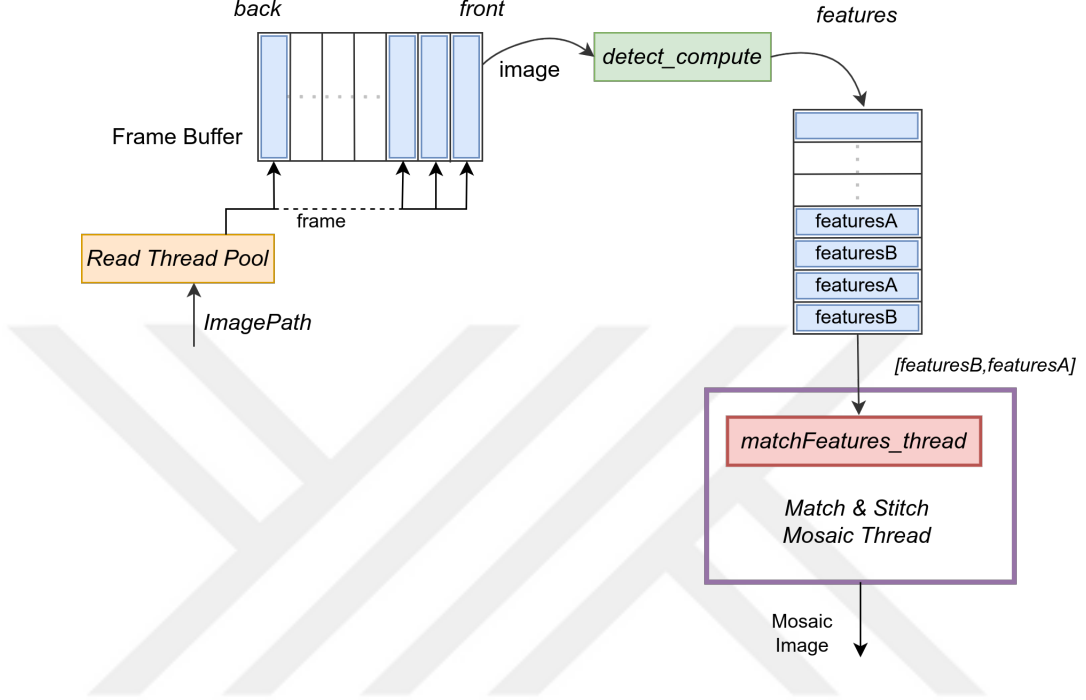


Figure 11: The schematic of C++ implementation of ISA

In the C++ version of the ISA, a queue data structure is used to store data. Three global queues are created to store frames, keypoints, and features. The first thread, called *read_thread*, is responsible for image reading and is pushed to the frame buffer tasks. The frame buffer is implemented in a First In First Out (FIFO) fashion, and as soon as the image is pushed to the queue, the *detect_compute* thread dequeues the frame from the buffer. Then, *detect_compute* thread computes the keypoints and features for the first frame and stores them into the queue created for them. When the features of the first two frames are computed, the *matchFeatures_thread* starts its tasks. Afterward, the points computed by *matchFeatures_thread* are sent to the *Match & Stitch Mosaic Thread*. This process continues until all the frames are stitched together. Since the *Frame*

Buffer does not have a fixed depth, it can store frames after reading without waiting for space in the queue. A similar approach applies to queues for keypoints and features.

The *matchFeatures_thread* is performs its tasks in the *Match & Stitch Mosaic Thread*. The main thread, which is *Match & Stitch Mosaic Thread*, creates and starts the threads to compute matches between frames. In the Algorithm 7 simplest version of the *read_thread* is shown.

Algorithm 7 C++ implementation of *read_thread*

```

1: thr_idx                                     ▷ Thread index
2: img  $\leftarrow$  imread(thr_idx)                 ▷ Each threads reads related images
3: img_scaled  $\leftarrow$  resize(img, S)           ▷ Scaled down by S
4: img_gray  $\leftarrow$  cvtColor(img_scaled)
5: FrameBuffer[thr_idx]  $\leftarrow$  img_gray

```

There are eight *read_thread* created in the implementation, and each of the threads is responsible for reading a specific range of images. For instance, when there are 100 frames to be stitched, the first thread will read images between 0-12. The rest of the thread and frame numbers, their responsible ones, are shown in the Table 5.

Table 5: Threads and frames to read

Thread	Frames to read
Thread #1	0 to 12
Thread #2	13 to 24
Thread #3	25 to 37
Thread #4	38 to 50
Thread #5	51 to 63
Thread #6	64 to 76
Thread #7	77 to 89
Thread #8	90 to 99

3.3 Results and Comparison

The experiments are done on the NVIDIA Jetson AGX Xavier, which has an 8-Core ARM CPU and 512-Core Volta GPU. In this thesis, a total number of 100 images with a size of 1024x1024 was used for the image stitching task. The sequential version of the ISA running on the CPU is assumed as a reference implementation. This work develops the various versions of ISA implementation: Serial-CPU, GPU-Accelerated, CPU-MT, GPU-MP, GPU-MT, and GPU-MP. The definitions of the versions are as follows:

- *Serial-CPU* : Single thread implementation of ISA. It runs in sequentially.
- *GPU-Accelerated* : Accelerated version of single thread implementation using GPU.
- *CPU-MT* : Multithreaded implementation using multi-threads. It runs on the CPU.
- *CPU-MP* : Multiprocessing implementation using multi-process. It runs on the CPU.
- *GPU-MT* : GPU-Accelerated version of the CPU-MT implementation.
- *GPU-MP* : GPU-Accelerated version of the CPU-MP implementation.

By offloading the bottleneck functions identified using profiling onto the GPU with the single thread running on the CPU, the run time was decreased significantly. The results in Table 6 show that the GPU-accelerated version is 1.32 times faster than the CPU version of the ISA. The frame rate of the GPU-accelerated implementation of ISA is 4.21 fps, whereas the frame rate of the base implementation is 3.17 fps.

In this thesis, developments are done on different environments such as a personal computer, a workstation, and Xavier. The workstation (WS) has Intel i9-9880H 2.3 GHz CPU and Quadro RTX 5000 GPU with 32GB RAM. The personal

Table 6: Comparison of single threaded ISA running on CPU and GPU

	Serial CPU	Serial GPU
Total Elapsed Time	31.517 sec	23.717 sec

computer (PC) has an Intel Core i7-10510U CPU 2.3GHz and NVIDIA GeForce MX250 GPU. The results of these environments are compared in the Table 7. The elapsed time for Xavier the image stitching implementations for 100 images is shown in Table 8.

Table 7: Speedup comparison of ISA implementations [1]

ISA Implementations	Xavier	Personal Computer	Workstation
Serial CPU	31.51 sec.	47.34 sec.	29.63 sec.
GPU Accelerated	1.3x	1.7x	2.1x
CPU-MT	1.5x	1.1x	1.1x
CPU-MP	2.1x	2.7x	3.1x
GPU-MT	2.5x	1.7x	1.3x
GPU-MP	2.6x	2.3x	2.9x

Table 8: Elapsed time of ISA implementations on Xavier

ISA Implementations	Elapsed Time	Speedup
Serial CPU	31.517 sec.	1x
GPU Accelerated	23.717 sec.	1.32x
CPU-MT	20.959 sec.	1.50x
CPU-MP	14.651 sec.	2.66x
GPU-MT	11.818 sec.	2.15x
GPU-MP	12.096 sec.	2.60x

Seen in the result in Table 7 the fastest implementation is GPU accelerated multithreaded version. It is 3.4x faster than Xavier’s serial CPU implementation, 2.7x faster for the personal computer, and 3.1x faster for the workstation. The GPU-MT version is the most efficient solution as it uses multithreading and better

suits the I/O intensive ISA. Since they do not take advantage of the GPU, CPU-MT and CPU-MP are the slowest implementations compared to other versions. The results show that the most efficient solution is the multithreaded CPU-GPU hybrid technique proposed in this thesis. The WAS systems can benefit from the CPU-GPU hybrid implementation to meet real-time requirements.

The minor area motion imagery (MAMI)[2] data set is used in the image stitching module development. In the MAMI data set, there are set of images captured by a large format electro-optical airborne platform. In the airborne vehicle, four ground-based cameras are used. The data was created in August 2013 at Wright Patterson Airforce Base (WPAFB), Ohio. The Figure 12 shows four samples of the MAMI data set.

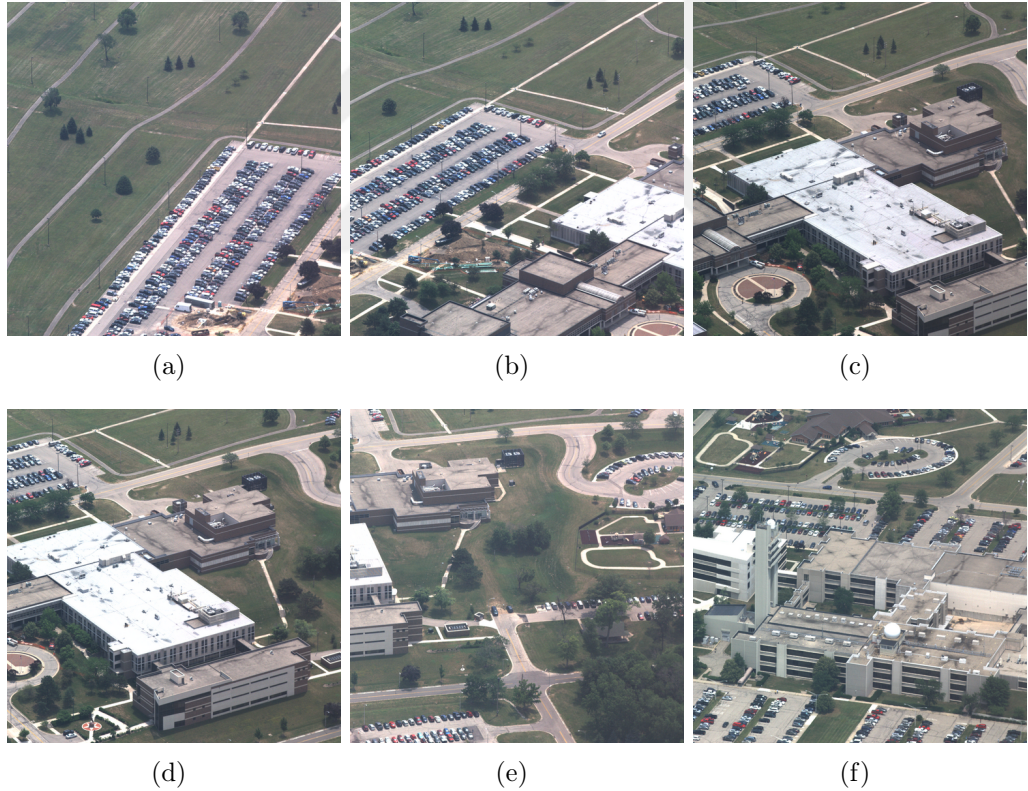


Figure 12: Samples of the MAMI data set [2]

The intermediate mosaic images at different iterations are shown in Figure 13, Figure 14 and Figure 15. The mosaic image produced using MAMI [2] data set at the end of execution is shown in Figure 16.



(a) Iteration 1

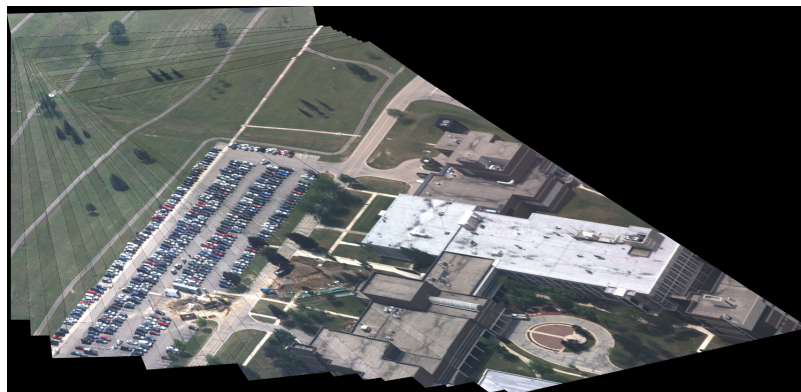


(b) Iteration 5

Figure 13: Intermediate mosaic images at 1st and 5th iteration

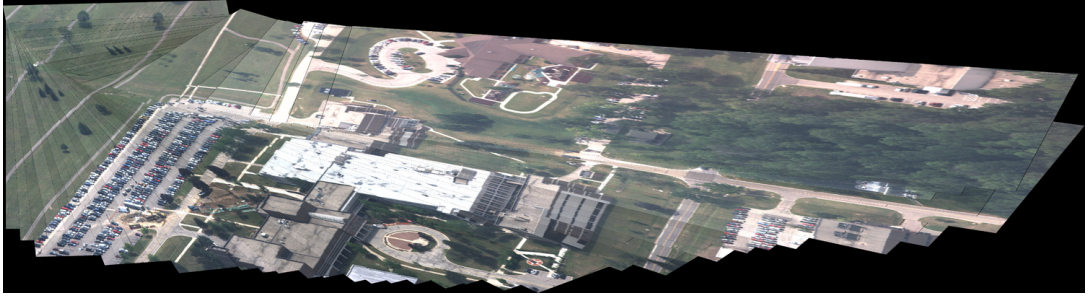


(a) Iteration 10

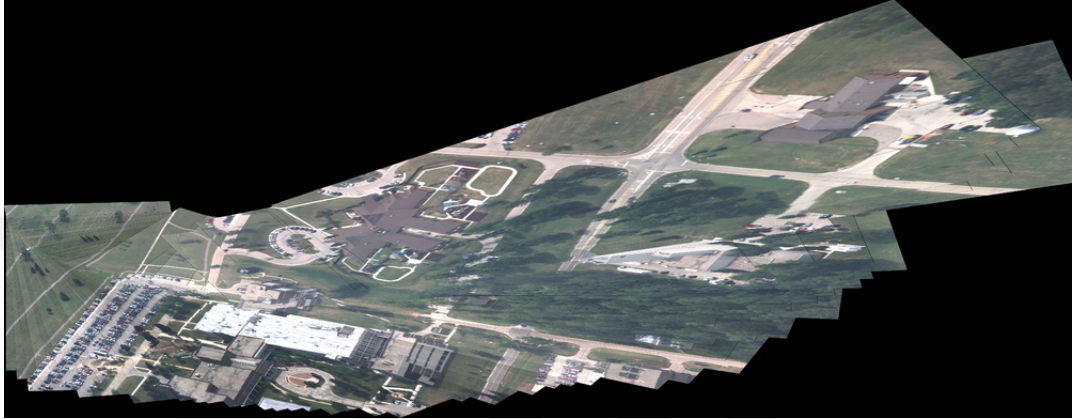


(b) Iteration 20

Figure 14: Intermediate mosaic images at 10th and 20th iterations



(a) Iteration 50



(b) Iteration 70

Figure 15: Intermediate mosaic images at 50th and 70th iterations

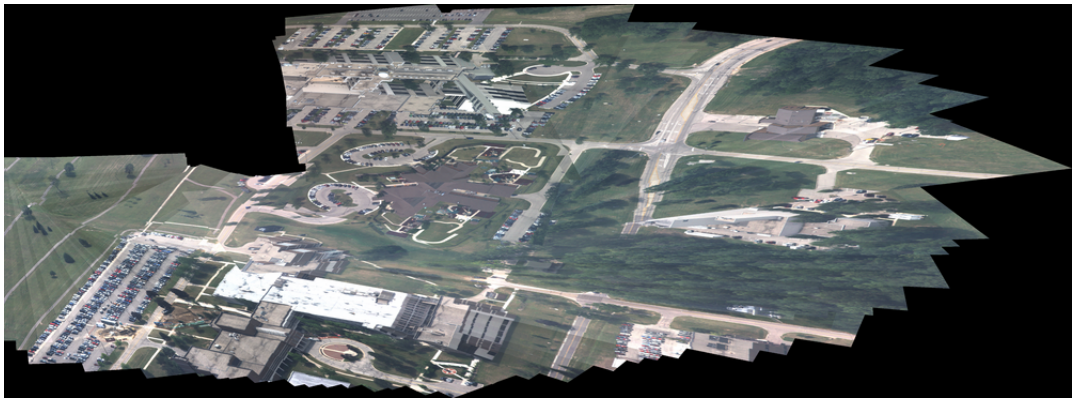


Figure 16: Final mosaic image

CHAPTER IV

DEEP LEARNING MODEL OPTIMIZATIONS

This chapter will discuss the implementation of deep neural network models and their optimizations. The overall system includes three deep neural network modules: Object detection, object tracking, and super-resolution. It is aimed to develop such a system, all system components to run in real-time on the embedded system. Different optimization methods are applied to these three modules. First, the YOLO (You Only Look Once) model is used for the object detection task. The method used for YOLO is *TensorRT* optimizations of the model. Additionally, multithreading is applied for object detection to achieve acceleration in terms of inference time. Then, model pruning and compress methods are applied for the object tracking. The NNI (Neural Network Intelligence) library developed by *Microsoft* is used for automated pruning tasks. Finally, for the super-resolution model, *TensorRT* optimizations are applied.

The main goal for the DNN optimizations is to accelerate the deep learning implementation in a system for WAAS. Since the memory size of the Jetson Xavier, it is important to be able to run all the modules concurrently and in real-time requires optimizations. For example, due to model pruning, the DNN model's size and inference time are reduced. Thus, this solves the memory consumption problem and runs the system in real-time.

The object detection with YOLO and its optimization will be explained in Section 4.1. Then, in Section 4.2 object tracking model used in the system and its optimization methods will be explained. The *TensorRT* and their results will be explained in Section 4.3.

4.1 Object Detection with YOLO Model

A widespread problem in the field of wide area aerial surveillance systems is object detection applications. [23] This study is aimed at detecting moving or fixed objects in the wide area. Since the data processed in the system has high-resolution and other modules need to run in a pipeline fashion, the object detection model needs to be optimized as much as. Three approaches are applied to object detection to solve the optimization problem: multi-core programming techniques, model compression, and TensorRT optimizations. In the Section 4.1.1 multithreading programming, and the model compression and TensorRT optimizations are disclosed in the Section 4.1.2.

4.1.1 Multi-core Programming on YOLO

The object detection model has three main tasks for the process to one frame: *pre-processing*, *inference*, and *post-processing*. The sequential flow of object detection is shown in Figure 17.

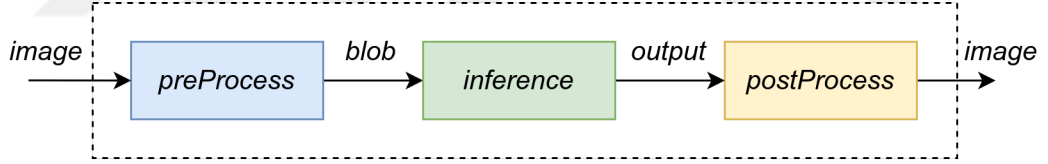


Figure 17: The sequential flow of object detection

The image captured from the cameras is taken as input to the object detection module. After *pre-processing*, the image is converted to a so-called blob object to network. The input of the network needs to be a 4D NumPy array object. The dimensions of the blob are images, channels, width, and height. Afterward, the network forwards the blob in order to produce the output. In *post-processing*, objects found on the image are extracted and marked on the image with their label. The sample output obtained from the object detection is shown in the Figure 18. The objects detected, such as car, truck, person, etc., are labeled on the frames.

The schematic of the multithreaded object detection module implemented in

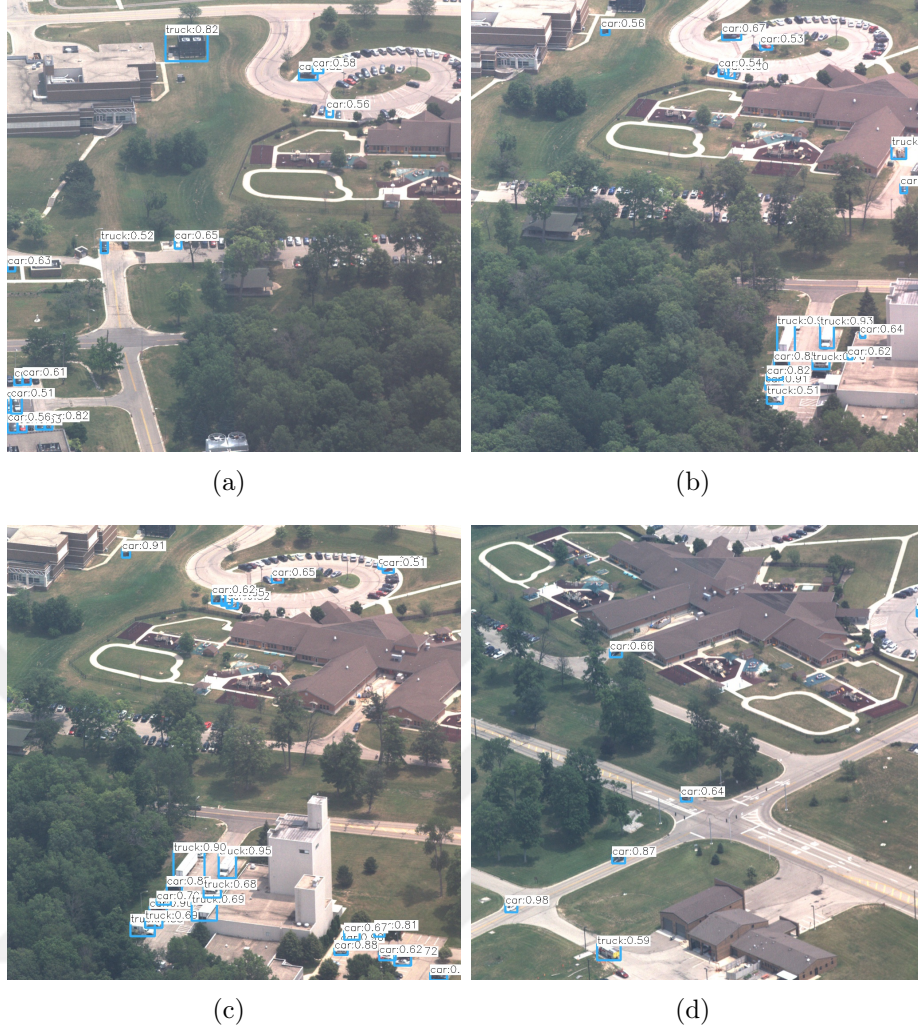


Figure 18: Outputs obtained from the object detection

the system is shown in Figure 19. The multithreaded version of the object detection module implementation includes three main threads for each task. All the tasks run concurrently as long as they can take necessary data from the related queues. The first thread is called *preProcess* and is responsible for pre-processing operations to the frames. Once the frames are done with the pre-processing operations, they are stored in the queue called *blobQueue*. Then, the second thread, called *inference*, performs the forward operation. The network used for object detection is created as a global network to be callable from inside the threads. Once there is data in the *blobQueue*, the inference thread takes it as input and starts the inference operation without waiting for the other threads. After the inference is made for a frame, the obtained output is stored to the *outQueue*. Finally, the

third thread called *postProcess* takes the output data from the *outQueue* and extracts the output to draw and label them on the frame. The *postProcess* thread is mainly responsible for the visualization of the object detected from the network. All queues used in the implementation are designed as FIFO and global variables so that threads can access them.

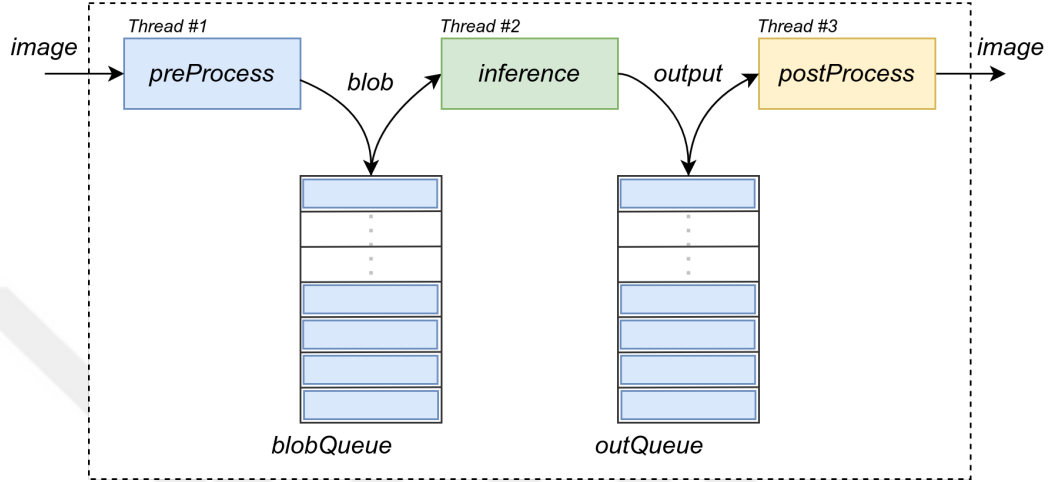


Figure 19: The schematic of multithreaded object detection module

Compared to sequential one, the advantage of this implementation is that the main tasks do not need to wait to process one frame to start incoming ones. Once there are enough data to process in the queues, threads can perform their operations and send the outputs to the queue or save the results. Also, the network is moving to the GPU to accelerate it. Thus, this implementation uses the advantages of both GPU and CPU and maximizes the utilization of both. The Algorithm 8, Algorithm 9 and Algorithm 10 show the threads called *preProcess*, *inference* and *postProcess* respectively, and their simplest version of the multithread implementation of object detection module.

The Algorithm 8 shows the simplest version of the thread called *preProcess* and its tasks. First, it reads images captured by cameras. Then, the image is converted to a blob object using *OpenCV*'s function named *blobFromImage* image. Finally, the blob object converted from the image is pushed to the *blobQueue* and sent to the second thread.

The inference process is shown in the Algorithm 9. If there is data to process,

Algorithm 8 Multithreaded Implementation of Object Detection

```
1:  $N$  ▷ Total number of images
2: function PREPROCESS(image)
3:   for  $ii == 0$  to  $N$  do
4:      $image \leftarrow \text{imread}(imagePath[ii])$ 
5:      $blob \leftarrow \text{blobFromImage}(image)$ 
6:      $blobQueue.push(blob)$ 
7:   end for
8: end function
```

Algorithm 9 Multithreaded Implementation of Object Detection

```
1:  $net$  ▷ Global variable YOLO Network
2: function INFERENCE(blob)
3:   for  $ii == 0$  to  $N$  do
4:     if  $blobQueue.isEmpty() \neq True$  then
5:        $blob \leftarrow blobQueue.pop()$ 
6:        $out \leftarrow net.forward(blob)$ 
7:        $outQueue.push(out)$ 
8:     end if
9:   end for
10: end function
```

the thread is popped the blob object from the *blobQueue*. Then, the image will be processed and sent to the network as a blob input. The output of the inference process is pushed to the *outQueue* to send the *postProcess* thread.

Algorithm 10 Multithreaded Implementation of Object Detection

```
1:  $net$  ▷ Global variable YOLO Network
2:  $image$  ▷ Processed image
3: function POSTPROCESS(out)
4:   for  $ii == 0$  to  $N$  do
5:     if  $outQueue.isEmpty() \neq True$  then
6:        $out \leftarrow outQueue.pop()$ 
7:        $image, predictions \leftarrow \text{postProcess}(out, net)$ 
8:        $cv2.imwrite(image)$  ▷ processed image saved
9:     end if
10:   end for
11: end function
```

As can be seen in the Algorithm 10, in the *postProcess* thread, the predictions are extracted and drawn to the current frame that is processing. When there is enough data to process in the queue called *outQueue*, the thread is performed its operations. The output obtained from the inference includes the predictions of

targets such as cars, trucks, people, trees, traffic lights, etc. After post-processing, the detected image is saved using the *imwrite* function of the *OpenCV* library.

The comparison of multithreaded and sequential versions of the object detection module is shown in the Table 9. The results show that the GPU-multithreaded one is 4.02 times faster than the CPU-sequential version of the module. All results are measured in seconds.

Table 9: Comparison of implementations of object detection

	Sequential	Multithreaded
CPU	58.213 second	24.441 second
GPU	31.837 second	14.458 second

Additionally, the object detection module is combined with the image stitching algorithm. Firstly, the object detection module is embedded in the image stitching algorithm serially. The block diagram of the implementation is shown in the Figure 20. However, in this implementation, the image stitching algorithm needs to wait for the object detection to be done. Thus, the goal of designing a real-time system has yet to be obtained. In order to solve this problem, different approaches are explored. Since these two modules do not need to wait for each other, a design is developed to run both simultaneously.

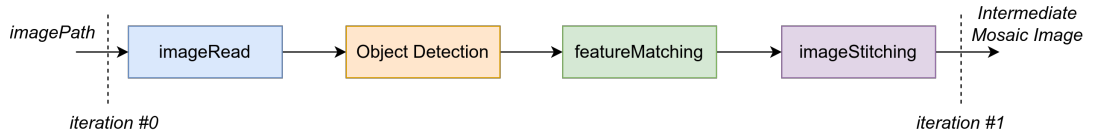


Figure 20: The schematic of sequential object detection module with ISA

The goal of this multithreaded architecture is to execute both modules concurrently. The object detection module and image stitching algorithm are assigned to various threads in order to do that. As a result, different connected threads are in charge of different algorithms. The schematic for the design is shown in Figure 21.

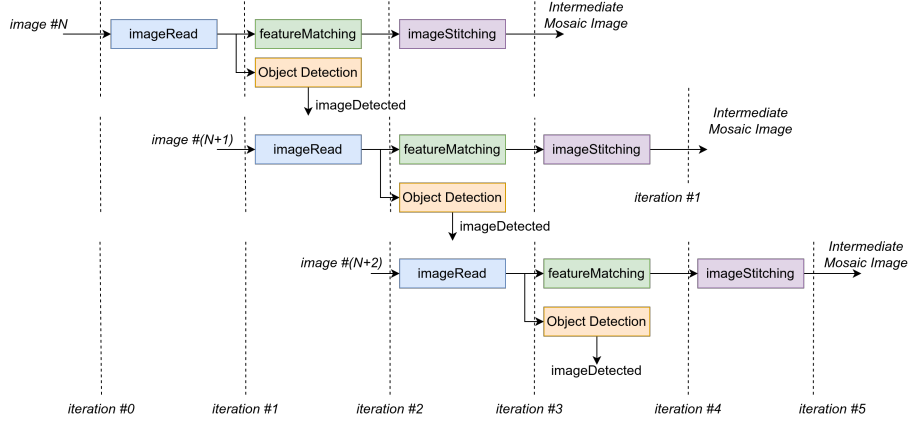


Figure 21: The schematic of multithreaded object detection module with ISA

According to the Table 10, combining object detection and image stitching and algorithm with a multithreading approach gives 1.30 time speedup for CPU, 2.74 time speedup for GPU comparison to the sequential CPU version. This implementation is done using Python language, and all results are in seconds.

Table 10: Comparison of ISA implementations with object detection

	Sequential	Multithreaded	Multiprocessing
CPU	41.171 second	31.629 second	22.944 second
GPU	-	15.011 second	17.414 second

4.1.2 Model Compression and TensorRT Optimizations

Second technique applied to the object detection module to accelerate it is optimizations using *TensorRT*. *TensorRT* is a library implemented based on C++ for high-performance computing on NVIDIA GPUs. [47] Since the system designed for the video surveillance module is GPU-based, *TensorRT* is an excellent choice to achieve acceleration. In addition to TensorRT optimizations, model compression algorithms are applied to the YoloV3 object detection model.

There are four main steps in model compression: sparse training, prune channels with small scaling factors, fine-tuning of the pruned network, and slimming. These steps are applied to the model in predefined numbers of iterations. The block diagram in the Figure 22 shows the model compression flow.

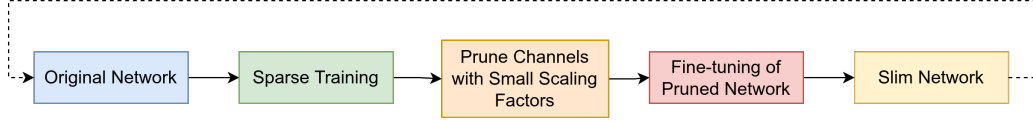


Figure 22: The flow of the model compression process

Sparse weight matrices on the deep learning layers are obtained by thresholding the model coefficients with small absolute values. Increasing model inference speed and reducing memory consumption are achieved by sparse matrix optimizations. In order to prevent losses in model performance, the sparsified model needs to train again. The model is fine-tuned in the sparse training approach, while the model is gradually sparse during every epoch. There is an inverse proportion between the sparsity ratio and the learning speed of the model. When the targeted sparse rate is increased, the learning speed is decreased, which improves the model's learning performance.

The channel pruning technique is applied to the other deep neural networks on the system. It is based on deleting the specific channels from each layer. The channel pruning applied for YoloV3, mainly applied to the shortcut connection, does not impact model performance while the acceleration is observed. In YoloV3, 23 shortcuts were used to evaluate various levels of channel pruning.

The layer pruning technique removes a deep learning layer completely. For the YoloV3 model, the pruning of the shortcut layers is focused. Two layers in front of the pruned are also pruned when a shortcut layer is pruned. (CBL, Convolution - Batch Normalization - Leaky ReLu).

The network slimming technique is based on the pruning of BN (Batch Normalization) layers and their connected layers. The convolutional layer to which the BN layer is connected has less significance in terms of model performance because the learning scale parameter in the BN layer is small. As a result, it offers a chance to prune everything at once. The loss function used during the model dilution regularization is shown in the Equation (1).

$$L = \sum_{(x,y)} l(f(x, W), y) + \lambda \sum_{\gamma \in \Gamma} g(\gamma) \quad (1)$$

In the Equation (1) (x, y) refers to the test data, W refers to the parameters of the training model, γ refers to the scaling factors on BN layers, and $g(\gamma)$ refers to the loss function to those parameters. Using $g(\gamma)$ limits the scaling factors' size and gives an automatic pruning impact during model training. At the end of the training, layers with a small γ value are removed, and the model is compressed successfully. The compression results are given in Section 4.4.

4.2 Object Tracking with HM-Net Model

In this study, the HM-Net (Heat Map Network) model is used for the object tracking operation. HM-Net is a regression network for object center detection and tracking on WAAS, namely Heat Map Network.[48] It is significantly faster than state-of-the-art tracking methods. The HM-Net algorithm is used center-based joint object detection and tracking approach. The algorithm takes as input the current frame, previous frame and the object detection heat map produced from the previous frame. The model predictions include the heat maps of object centers, motion, and sub-pixel location refinement. The information in those heat maps is object center locations, motion vectors, and precision of the objects, which is sub-pixel refinement of places. Each heat map has the same width and height (W: width of the input image, H: height of the input image) as the input image. [48] The input/output diagram of HM-Net is shown in Figure 23. Despite the HM-Net's tracking success, the complexity of the model needs to be decreased for Jetson Xavier real-time execution.

The trade-off between algorithm speed and GPU-based system requirements, like video frame rate (fps) and resolution, is examined in order to reduce complexity. In order to reduce the complexity, model pruning, quantization, and *TensorRT* optimizations have been applied. As a result of these studies, model pruning-quantization-training for fine-tuning is designed and made into re-configurable

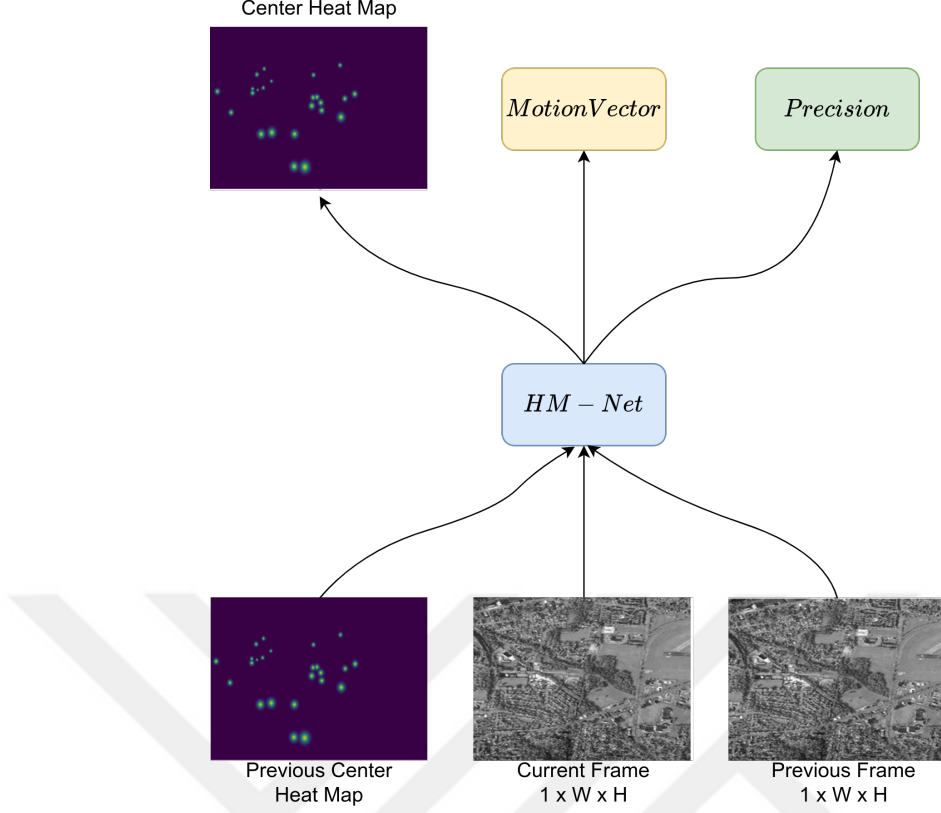


Figure 23: The input-output diagram of HM-Net

modules.

Following model pruning, both inference time and the model's memory footprint are decreased. The steps in the model pruning procedure are as follows:

- Pruning the weights of the model,
- Quantization the weights of the model,
- Train the model for fine-tuning purposes.

The L1Filter Pruning algorithm is used to prune the feature channels on the HNet convolutional layers. The L1Filter Pruning algorithm is used in the Neural Network Inference (NNI) library. After pruning, the model is re-trained using the VisDrone data set for fine-tuning. A straightforward pruning algorithm known as L1Filter removes filters in convolutional layers. [49] In this algorithm, the related channel in the layer is eradicated. Since the layer dimension is changed, the front

and back layers connected with the pruned one need to change their dimensions to prevent a mismatch. The steps to identify the layers that are going to be pruned:

1. Find the summation of weights of the kernels on each filter (F_i) on a specific layer

$$s_j = \sum_{i=1}^n \sum |K_i^j| \quad (2)$$

The n in the Equation (2) refers to the total number of input channels, and K_i^j refers to the kernel weights of each input channel.

2. Sort the filters according to their S_j values.
3. Prune m filter with lowest S_j value. (m is a pruning ratio, and the user defines it.)
4. Remove the channel of pruned filter and prune the kernel weights that filter on this channel in the next layer.
5. After pruning, create a new weight matrix with new dimensions for the next layers and copy the values from the original model to that matrix.

The original and pruned model's training and test results and their comparison are presented in the Section 4.4.

4.3 *Super-Resolution*

This section explains the super-resolution module used in the proposed WAAS system. It is important to have a high spatio temporal resolution for better precision in WAAS systems. The super-resolution module aims to produce a frame sequence with a high frame rate and high-resolution. As the primary goal was to develop a high-performance model, the RIFE architecture was first retrained to use low-resolution frame information to achieve a frame rate increase. [50] The architecture of the super-resolution model is shown in the Figure 24 [51].

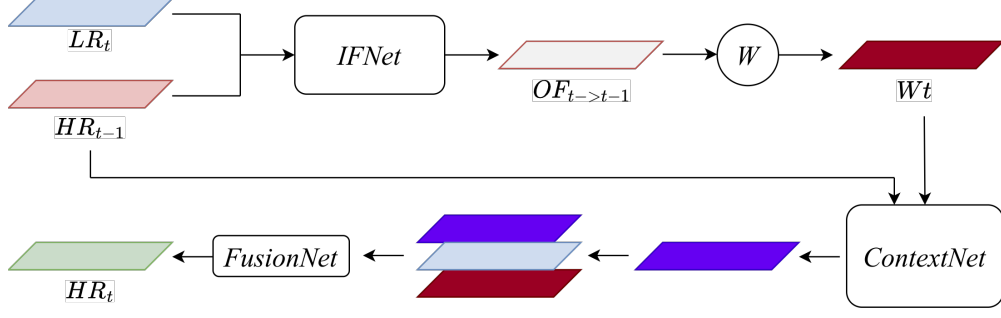


Figure 24: The architecture of super-resolution module

While the original RIFE architecture uses only high-resolution frames, model used in the proposed system uses low-resolution frames as well. The super-resolution model takes a low-resolution frame for a given t time and a high-resolution frame for the previous $t - 1$ time. As a result, a high-resolution frame for a given t time is produced. The model has four main modules: Optical Flow (IFNet), warping, ContextNet, and FusionNet. The IFNet is used for producing warping versions of the high-resolution frames and optical flow between low-resolution frames. Then, after obtaining the optical flow between the low-resolution frames, the warped forms of low-resolution frames are obtained by applying the back warping process. Afterward, LR and HR frames and optical flows between them are given to the ContextNet model and obtained warped feature channels on different resolutions. Finally, warped high-resolution frame, warped low-resolution frame, low-resolution frame at t time and output of ContextNet module is sent to the FusionNet module as inputs, and high-resolution frame at t time is obtained.

4.3.1 Optical Flow Module (IFNet)

The IFNet model is used to produce optical flows in the super-resolution module. IFNet is used to produce optical flow for both high-resolution and low-resolution frames. The IFNet module uses high-resolution frame at $t - 1$ time and low-resolution frame at t time.

4.3.2 Warping

Warping is the image registration module used to register one frame onto the other with the help of optical flow. Using the existing back warping module in the IFNet model, both high-resolution and low-resolution frames are warped and aligned with the frame at t time. This module is a derivable function and therefore allows the backward gradient convolution within the model.

4.3.3 ContextNet

ContextNet is a feature extraction module used from the RIFE architecture. Frames and optical flows are taken as input, and features for different resolutions are produced. Therefore, detailed features of the frames can be obtained.

4.3.4 FusionNet

The FusionNet used in the super-resolution module is inspired by the skip connection UNet model. In the super-resolution module, FusionNet takes warped frames, low-resolution, and high-resolution frames as a reference, and feature maps coming from the ContextNet as inputs.

4.4 *Results and Comparison*

The Table 11 shows the result of different pruning techniques and their comparisons. The model inference time (FPS) is measured on AGX Xavier. Regular pruning done without sparse training is caused significant performance loss and does not provide an acceleration. Network slimming, layer pruning, and layer-channel pruning techniques, thanks to fine-tuning operations, achieve mAP and fps values more than the original model. Since the network slimming pruned only the channels, acceleration is low. By layer pruning the shortcut connection of YoloV3, significant acceleration and better mAP results were obtained.

The Table 12, Table 13, and Table 14 present the performance comparison of YoloV3, YoloV4, and optimized YoloV4 model with original, optimized with TensorRT-FP32 and optimized with TensorRT-FP16 models on Xavier. While the

Table 11: The result of different pruning type on YoloV3

	Original Model	Normal Pruning	Layer Pruning	Network Slimming	Layer-Channel Pruning
mAP	23.5	9.0	31.5	30.7	30.8
Layers	106	106	94.0	104	88.0
Params (10^6)	62	42	52	48	48
FPS	16.0	14.0	22.0	17.0	22.0

models optimized with TensorRT using FP32 precision do not provide a significant acceleration, the result of training with FP-16 precision is that both mAP values are preserved, and acceleration obtained.

Table 12: The performance of pruned Yolo

Model: AGX Xavier	Yolov4	Optimized Yolov4	Yolov3
mAP @ 0.5	28.0	33.0	23.5
FPS	12.0	12.0	16.0

Table 13: The performance of Yolo pruned with FP32

Model: AGX Xavier FP32	Yolov4	Optimized Yolov4	Yolov3
mAP @ 0.5	30.0	33.0	23.3
FPS	13.0	13.0	16.0

Table 14: The performance of Yolo pruned with FP16

Model: AGX Xavier FP16	Yolov4	Optimized Yolov4	Yolov3
mAP @ 0.5	29.0	33.0	23.2
FPS	30.0	30.0	37.0

The Table 15 shows the result for object tracking performance of the original and pruned HNet model on the VisDrone test dataset. The models compared are as follows:

- Original HM-Net
- Model 1: Compressed by %30
- Model 2: Compressed by %30, 32-bit FP (floating point) precision, optimized by TensorRT
- Model 3: Compressed by %30, 16-bit FP (floating point) precision, optimized by TensorRT (with FP-16 option)

The results in the Table 15 and Table 16 are obtained from L1Filter pruning algorithm. Clearly seen in the Table 15, there is no accuracy lost in terms of object tracking performance in the model with %30 compression, in fact, the accuracy is slightly increased. While TensorRT optimizations have no impact on object tracking performance in 32-bit floating points, it is clearly seen that performance is increased in 16-bit floating points. As a result, firstly, the model needs to fine-tune with the 16-bit FP format, then the TensorRT optimizations must apply.

Table 15: The object tracking result of original and pruned HM-Net Model

Model	mAP				Classes				
	AP	AP25	AP50	AP75	Car	Bus	Truck	Peds	Van
Orig.	22.79	34.45	20.06	13.86	47.79	27.45	9.570	9.03	20.10
Model-1	23.20	34.36	21.95	13.29	47.58	22.05	11.90	11.33	23.16
Model-2	23.18	34.30	21.90	13.34	47.88	22.04	11.74	11.17	23.09
Model-3	17.57	27.88	16.24	8.580	34.56	18.61	7.810	10.08	16.79

In the Table 16, the model’s speed on Jetson AGX Xavier and memory consumption are listed. As seen in the results in the Table 16, although the size of the compressed model named Model-1 is %30 smaller than the original model, the speed-up ratio is only %17. Since the pruning process reduces the number of channels and does not change the depth of the model, the speed-up ratio remains less. Also, reducing the model depth causes a significant loss in object tracking performance.

Table 16: The optimization results of speed and model size of HM-Net model

Model	Speed (fps)	Model Size (MB)
Original	1.45	71.7
Model - 1	1.70	48.3
Model - 2	3.76	37.2
Model - 3	8.40	52.6

The Table 16 shows that using TensorRT optimizations on the model gives significant acceleration. The TensorRT library of the NVIDIA, with specific optimizations for Xavier, gives 260 acceleration ratio even though 32-bit FP format is used. When 16-bit FP optimization is used, the speed-up ratio increases to 580. The size of the Model-2, which has a 32-bit FP format, and TensorRT optimization, is 137.2MB, while the size of the Model-3, which has a 16-bit FP format, is decreased to 51.6 MB.

The results in the Table 16 are only shows the inference time. The other metrics, such as pre-processing (image read, producing heat map), identifying the object centers from the heat map, and creating object navigation from registering objects in consequent frames, are not included in those results. While the time-consuming post-processing tasks for the original model is approximately 10% of the inference time, for the 16-bit FP model with TensorRT optimizations, this consumed time is increased to 50% of the inference time. Thus, in addition to model optimization, it is important to optimize pre-processing and post-processing tasks as well.

After pruning the HNet model with the L1Filter pruning algorithm, a different algorithm is used for pruning the model: FPGM (Filter Pruning via Geometric Median). [52] FPGM pruning algorithm is based on identifying and removing the filters containing residual information on the pre-trained CNN model. To identify the residual filters, geometric means for all filters are calculated. Then, the filters close to that mean are assumed residual and prunable.

First of all, the configuration list for the pruning process is described. The configuration list contains the pruning type, sparsity ratio, and operation type. The operation type feature defines which operations are going to compress and pruned. This study uses three sparsity ratios: %30, %40, and %50. Also, quantization is applied to the model to reduce the training time and memory consumption. Pytorch library and its built-in functions are used for quantization. In the Pytorch library, Pytorch-AMP (Automatic Mixed Precision) is used to perform some operations using float32 datatype and others using float16 data type. This library supports quantization-aware model training. This library can do this operation automatically and mix. With the AMP, each operation is matched with a suitable datatype. Thus, training time and memory consumption can be reduced.

The simulation results are shown in the Table 17 and in the Table 18. The models compared are as follows:

- Original HM-Net
- Model-1: Compressed with L1Filter by %30
- Model-2: Compressed with L1Filter by %40
- Model-3: Compressed with L1Filter by %50
- Model-4: Compressed with FPGM by %40
- Model-5: Compressed by %30, 16-bit FP precision, optimized with TensorRT (with FP-16 option)
- Model-6: Compressed by %30, trained with AMP (mixed with 32-bit/16-bit), optimized with TensorRT (with FP-16 option)

The results in the Table 17 show that there is no loss in the accuracy in terms of object tracking performance with compression by %30; on the contrary, there is a slight increase in the accuracy. The results with %30 and %40 compression ratio shows that accuracy slightly decreased. There is no significant difference between

Table 17: The object tracking result of original and pruned HM-Net model

Model	mAP			
	AP	AP@0.25	AP@0.50	AP@0.75
Original	22.79	34.45	20.06	13.86
Model-1	23.20	34.36	21.95	13.29
Model-2	20.89	29.55	20.32	12.81
Model-3	19.30	29.13	17.41	11.36
Model-4	20.28	31.62	18.06	11.15
Model-5	17.57	27.88	16.24	8.58
Model-6	16.38	28.71	13.08	7.37

the results of L1Filter Pruning and FPGM Pruning; FPGM pruning achieves relative acceleration.

In addition, using the PyTorch AMP library, the HM-Net model is trained with mixed precision quantization. The purpose of this, with quantization-aware training, is to prevent a loss in accuracy that comes from FP 16-bit quantization. The training was completed successfully with lower training and validation losses. However, when the model is tested using the test dataset, the result shows a decrease in mAP (mean Average Precision) values. The mAP is a metric commonly used to measure object detectors and trackers. In the Table 17, Model-6 refers to a model trained with AMP and optimized with TensorRT using the FP-16 option. While the object tracking performance was expected to increase, on the contrary, a decrease in mAP values was observed compared to Model-5, which is not trained with a quantization-aware training technique. The model’s architecture is likely the cause of this issue. Since the results with FP 16-bit is not as good as expected, INT-8 quantization needs to be studied.

The results in the Table 18 only show the inference time of the model. Again, the other metrics, such as pre-processing, identifying the object centers from the heat map, and creating object navigation from registering objects in consequent

Table 18: The optimization results of HM-Net model

Model	Speed (fps)	Model Size (MB)
Original	1.45	71.7
Model - 1	1.70	48.3
Model - 2	1.80	37.9
Model - 3	1.91	30.9
Model - 4	1.85	37.6
Model - 5	8.40	52.6
Model - 6	8.40	52.6

frames, are not included in those results. While the time consuming for pre-processing tasks on the original model is %10 of the model inference time, pre-processing time on the optimized model with TensorRT using 16-bit FP is %50 of the inference time.



Figure 25: Samples of the VisDrone dataset [3]

For the object tracking, object detection and super-resolution evaluations, the VisDrone [53] data set is used. The AISKYEYE team collects the VisDrone data set at the Lab of Machine Learning and Data Mining, Tianjin University, China.

The data set is collected for various computer vision tasks. The images are captured using various drone platforms in different scenarios, and it is taken from 14 different cities in China. The data set mainly focuses on four major tasks: object detection in images challenge, object detection in videos challenge, single-object tracking challenge, and multi-object tracking challenge. The images contain various targets such as pedestrians, cars, bicycles, and tricycles [3]. The four samples of VisDrone data set are shown in the Figure 25

The super-resolution model is optimized using TensorRT using FP16 and FP32 option. The results are shown in Table 19. The simulations are done for total 1247 images with 544x960 resolutions. The Peak signal-to-noise ratio (PSNR) value refers to a ratio used as a quality measurement between the original and compressed model. The models compared are as follows:

- ORiginal: Original Model, Full-warp
- Model-1: TensorRT Optimized model, FP32, Full-warp
- Model-2: TensorRT Optimized model, FP16, Full-warp
- Model-3: TensorRT Optimized model, INT8, Full-warp

Table 19: The optimization results of super-resolution model

Model	Speed (fps)	PSNR
Original	3.80	39.79
Model - 1	3.93	39.79
Model - 2	4.18	39.79
Model - 3	4.36	38.86

CHAPTER V

EVALUATION OF OVERALL SYSTEM ARCHITECTURE

The GPU-Based acceleration of video surveillance modules in wide area motion imagery and the overall system architecture is evaluated in this chapter. The project is aimed to design a compact, cheap, and intelligent wide area surveillance system for unmanned aerial devices. The chapter is split into sections to discuss various project results. In the Section 5.1, resource statistics of the proposed embedded system will be discussed. The experimental results of the study will be explained in the Section 5.4, and then the question for how the timing is measured will be answered in the Section 5.2. The overall system designed for the video surveillance module is disclosed in Section 5.3. In the Section 5.5, results of this study and comparison with other works found in the literature will be presented.

5.1 Resources

In this thesis, a system for GPU-Based acceleration of video surveillance modules in wide area motion imagery is designed. Since wide area motion imagery is focused on a large amount of data, to be able to process all the visual data in real-time, a GPU-based system is a need. For the GPU-based system, NVIDIA Jetson AGX Xavier is chosen. The design includes an image stitching algorithm with deep neural network models as components of the entire system. The DNN models are fitted into the GPU, while the image stitching algorithm is fitted into the CPU using multi-core programming. Devices and environments and the data set used in this study will be disclosed in the subsections.

5.1.1.1 Devices and Environments

The complete design is implemented into the NVIDIA Jetson AGX Xavier edge computing device. Jetson AGX Xavier is a device developed especially for AI applications. Since the project aimed to develop a cheap and compact design, Jetson AGX Xavier is the best choice. NVIDIA's Jetson platform is commonly used in computer vision and AI applications. [54, 55, 56] Mainly, Jetson AGX Xavier is chosen for this work because it is perfectly designed for high-performance systems. The Jetson AGX Xavier used in this study is shown in the Figure 26.



Figure 26: NVIDIA Jetson AGX Xavier

NVIDIA has developed various edge computing devices with different specifications for applications requiring high-performance computing. Jetson devices can be flashed with NVIDIA Jetpack SDK and. Jetpack SDK, including the TensorRT, OpenCV, CUDA Toolkit, and CuDNN frameworks and libraries. Also, it has L4T with the LTS Linux Kernel. Table 20 shows the technical specifications of NVIDIA Jetson AGX Xavier. [57] Xavier is a powerful AI workstation with an NVIDIA Volta™ GPU with Tensor Cores. It has two NVDLA engines and an 8-core 64-bit ARM CPU.

Jetson Xavier captured images using multiple cameras. One of the cameras is a high-resolution scanning camera that can scan the area, while the other camera is a wide field of view (FOV) camera on an aerial vehicle. The developed system is responsible for data capturing using cameras and processing the data using various

Table 20: Technical specifications of NVIDIA Jetson AGX Xavier

	AGX Xavier
GPU	512-core Volta GPU with Tensor Cores
DL Accelerator	(2x) NVDLA Engine
CPU	8-core ARM v8.2 64-bit CPU, 8-MB L2 + 4 MB L3
Memory	16 GB 256-bit LPDDR4x 137 GB/s
Storage	32 GB eMMC 5.1
Video Encode	8 x 4K @ 60 (HEVC)
Video Decode	12 x 4K @ 30 12-bit support
JetPack Support	JetPack 4.1.1

applications such as image stitching, object detection, tracking, etc.

For deep neural network model optimizations and training for fine-tuning purposes, a workstation is used. The specifications of workstation are shown in Table 21.

Table 21: Technical specifications of workstation

	Nova Server
GPU	(4x) NVIDIA GeForce RTX 2080 Ti 11 GB
CPU	Intel® Xeon® Gold 6226R CPU @ 2.90GHz
Memory	192 GB
Storage	2 TB
Operating System	Ubuntu 18.04.5 LTS

This study is coded using two different programming languages: *Python* and *C++*. Different libraries and frameworks are installed in the Python environment, such as OpenCV-contrib, TensorRT, Pytorch, etc. The image stitching algorithm is accelerated based on the OpenCV-contrib library, and its CUDA-supported functions. OpenCV is built from the source using OpenCV-contrib repository using Cmake. For the DNN modules, the Pytorch library is used and accelerated,

especially using the TensorRT library. In the C++ environment, the same algorithms are implemented. The OpenCV-contrib repository is built for C++. Also, for DNN modules, LibTorch is used. LibTorch is C++ distribution of Pytorch, and it uses the `torch::Tensor` class. [58]

5.1.2 Data Set

In WAAS applications, a huge amount of data is needed to analyze the field properly. In the overall system, WPAFB 2009 Data set is used in for the simulations. [4] The data set captured in 2009 is contained imagery from an EO platform and six cameras were used to collect this data set. The samples of the data set are shown in Figure 27.

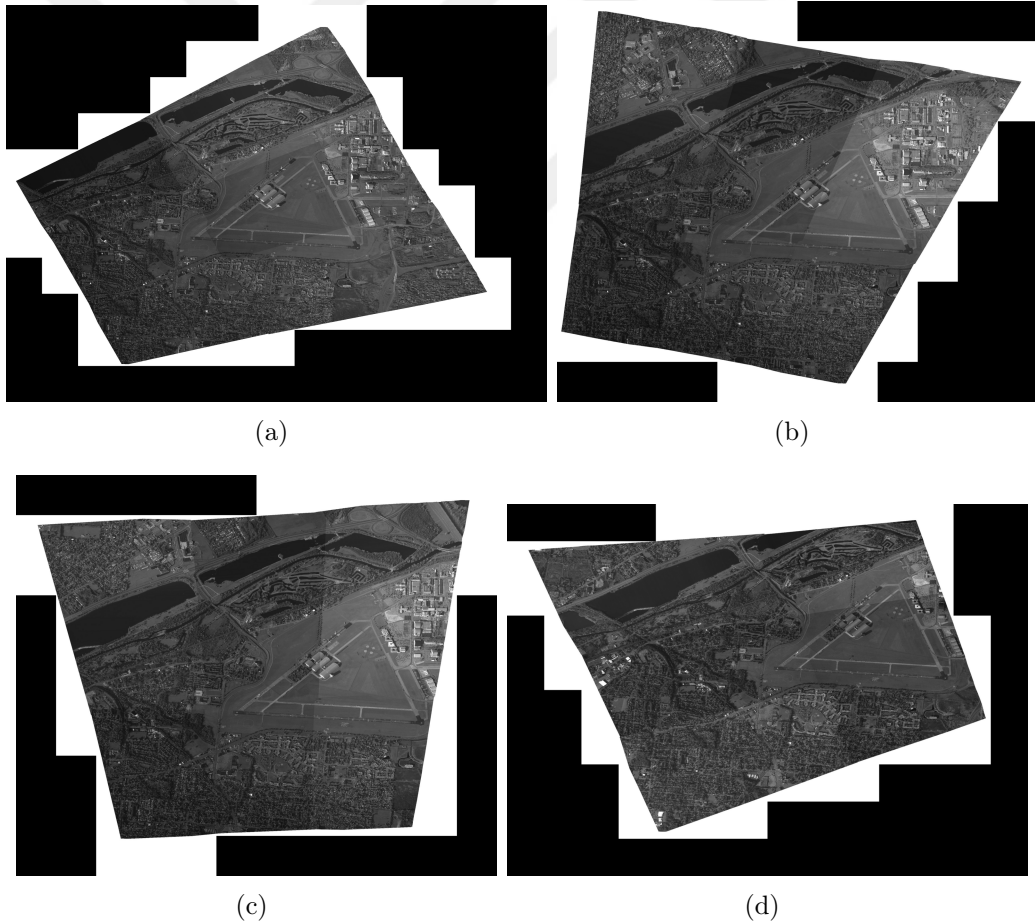


Figure 27: Samples of the WPAFB 2009 Data set [4]

5.2 *How the Timing Measured?*

The answer of the question how the timing measure are answered in this section. Since, and acceleration of video surveillance modules and their execution time is the main focus of this study, measuring elapsed time is become an important step. There are various approaches for measuring execution time.

At the beginning of this study, code profiling is applied to the algorithm to identify the most time-consuming functions. Python has two standard libraries that provide two different implementations of the same profiling template. They are called *profile* and *cProfile*. The *cProfile* is the most recommended one, and it has reasonable overhead compared to the *profile* library. The profiler modules are most suitable for statistical analysis of a given program, and it is not recommended for benchmark purposes. [59] To identify the bottleneck of image stitching algorithm, *cProfile* library of Python is used. After identifying and accelerating the bottleneck function, elapsed time needs to be calculated to analyze the system's performance. The *time* module in the Python standard library is used to calculate the elapsed time. Timestamps are placed at the beginning and end of code blocks that try to find execution time. The timestamps are written using *time.time()* [60] builtin method. The difference between start and end timestamps gives the elapsed time for that code block. The difference results in the total time elapsed to execute the code in seconds. This approach is applied to three main modules in the image stitching algorithm.

For C++ implementation of the algorithm, *VTune* Profiler developed by Intel is used. [61] *VTune* is a profiler developed to examine hardware performance and find and fix the bottleneck of applications. Using *VTune*, the bottleneck and the hot-spots, the most time-consuming parts of the codes, are identified. The results of the *VTune* profiler are shown in Figure 28.

Function	Module	CPU Time ②	% of CPU Time ②
<code>cv::remapBilinear<cv::FixedPtCast<int, unsigned char, (int)15>, cv::RemapVec_8u, short></code>	libopencv_imgproc.so.4.5	20.658s	24.9%
<code>cv::imread</code>	libopencv_imgcodecs.so.4.5	9.491s	11.5%
<code>cv::opt_SSE4_1::WarpPerspectiveLine_SSE4_Impl::process</code>	libopencv_imgproc.so.4.5	9.040s	10.9%
<code>cv::warpPerspective</code>	libopencv_imgproc.so.4.5	6.383s	7.7%
<code>cv::WorkerThread::thread_body</code>	libopencv_core.so.4.5	5.942s	7.2%
[Others]	N/A*	31.315s	37.8%

*N/A is applied to non-summable metrics.

Figure 28: The most active functions in the C++ implementation

5.3 Overall System Architecture

The overall system designed for video surveillance modules is explained in this section. The video surveillance modules used in this system are accelerated in the most conveniently way, and the modules are combined to create an overall system. The Figure 29 shows the overall system designed in this study. The accelerated modules are running on Unmanned Aerial Vehicle. According to the request of the operator in the ground control center, the observation area of the UAV is determined. The information obtained is transmitted to the operator in the ground control center again.

The wide area surveillance systems usually captures frames using multiple cameras. In the embedded UAV system developed in this thesis, two cameras are embedded. One of the cameras is a wide field of view (FOV) camera, and the other is a high-resolution camera that scan the area being observed on the aerial platform. The system contains some tasks such as capturing the images using cameras, pre-processing (image stitching, feature matching etc.), object detection and tracking, produce high-resolution images with higher frame rates, and transferring those captured images and information to the central control center on the ground. Image processing and deep learning techniques are used to analyze the data. The main goal is to design an innovative, cheap, compact, and real-time processing performance using GPU based embedded system with advanced image and video analytic capabilities.

The control center determines an area of interest (AOI) for further analysis

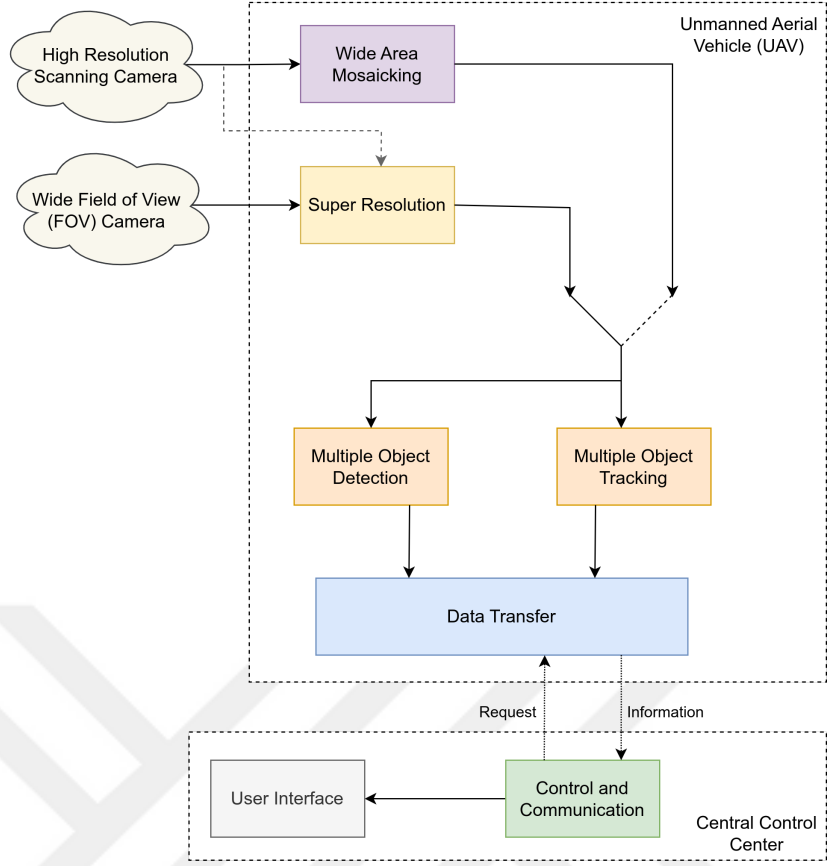


Figure 29: The overall system on UAV

and sends its coordinates to the embedded system. The WAAS system crops this AOI from matched images of both the FOV and PTZ cameras. Since the PTZ camera is scanning the full wide area, producing a lower frame rate sequence for the AOI. The frame rate of the high-resolution camera is increased by the super-resolution module, which takes matched neighboring LR and HR frames as input. In the proposed design in this study, the ratio between frame rates of the cameras assumes two.

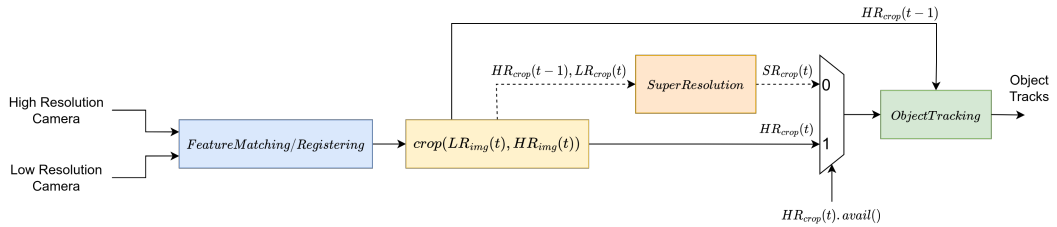


Figure 30: The overall system processing flow

In the Figure 30, overall system architecture and the modules that run on the

system at a given t time are shown. The images were captured from the HR (high-resolution) and LR (low-resolution) cameras and sent to the modules. The frames that come from the scanning camera are matched with the frame that comes from the FOV camera to determine the matching coordinates to be used for cropping the AOI in both camera outputs. Here, the feature matching module that described in the Chapter 3 is used. According to the coordinates that are computed, the frames are cropped to identify the surveillance coverage. Then, cropped frames are sent to the object tracker module. At this stage of the process, since AOI refresh rate of the high-resolution camera is low, the super-resolution module is optional. If at a given t time, a high-resolution frame is available for the AOI, the frame is directly sent to the tracker module. If not, in order to produce a high-resolution frame at t time, low-resolution frame for t time and a reference high-resolution frame for $t - 1$ time are sent to the super-resolution module as input. The $t - 1$ refers to the previous frame at a given t time. Afterwards, produced high-resolution frame is sent to the tracker module. The inputs of the tracker modules are high-resolution images for t and $t - 1$ times. This process is repeated throughout the entire wide area surveillance process. The flow shown in the Figure 30 is defining the process a one frame. This is repeated until the camera reads frames.

5.3.1 Multithreaded Overall System

This section discusses the multithreaded version of the overall system architecture. The proposed WAAS system is designed in a multithreaded fashion in order to meet real-time requirements. The main modules discussed in the Section 5.3 are offloaded to relative threads to create a pipeline system. The multithreaded overall system architecture is shown in the Figure 31.

The overall system efficiency is improved using multithreading programming and queues for data sharing between threads. Main modules are assigned to relative threads to be able to run them concurrently. There main threads are: *im-Read_thread*, *featureMatcher_thread*, and *superResolutionObjectTracking_thread*.

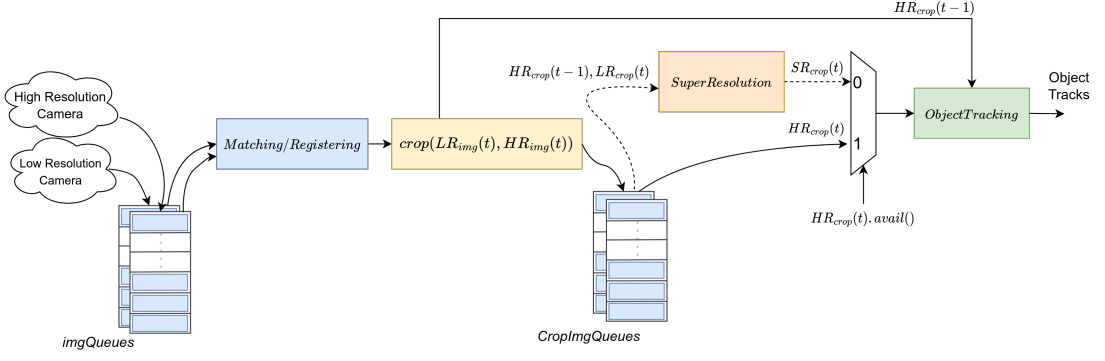


Figure 31: The multithreaded overall system architecture

Different queues are created for data sharing between threads. The built-in Queue object has *put()*, and *get()* methods, and these methods have their signaling operations. The *put()* method waits until there is enough slot to put the data in the queue and the *get()* method waits if there is no available data in the queue. Therefore, race conditions that are most likely to occur between threads are prevented.

5.3.2 Graphical User Interface for WAAS System

This section introduces the Graphical User Interface (GUI) designed for the proposed WAAS system. The graphical user interfaces are used to allow users to interact with programs through buttons, labels, and windows. The designed GUI for the proposed WAAS system includes buttons, a checkbox, drop down menu to select various parameters such as camera paths, running as serial or multithreading, and selecting if the *Feature Matcher* module is going to run on GPU. Also, there are two labels that show wide area images and area of interest selected for monitoring. The top left coordinates of the area of interest are defined by the user. The screenshot of the interface and a sample result is shown in the Figure 32.

5.4 Experimental Results

The experimental setup and the results obtained from the overall system are discussed in this section. The experimental setup for this study consisted of an

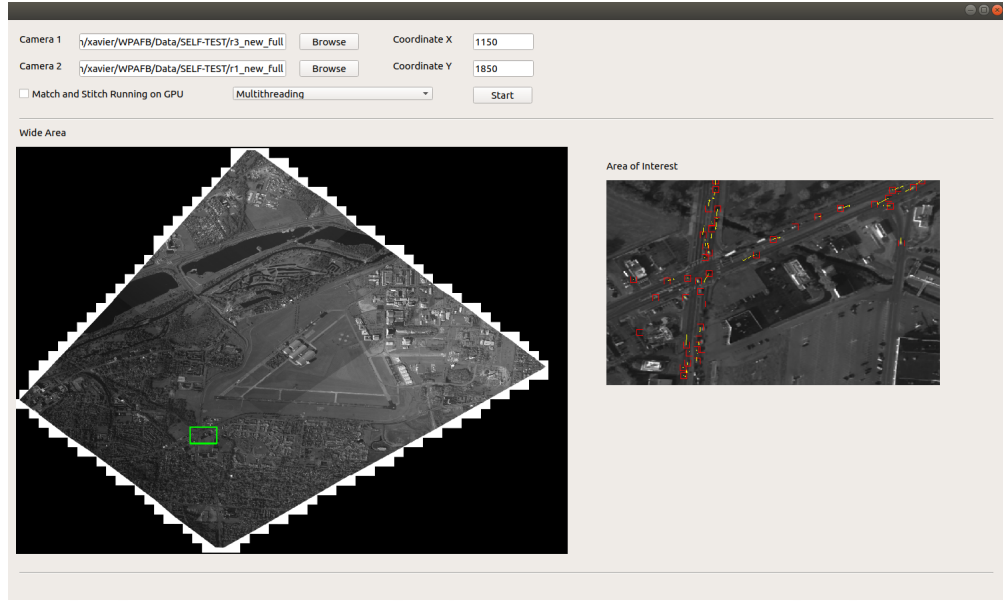


Figure 32: The graphical user interface

NVIDIA Jetson AGX Xavier edge computing device, which was used to implement the accelerated image processing module. The device was configured with a eight of CPU cores and a 512-core Volta GPU with Tensor Cores. The image processing algorithms were implemented in Python programming language , and the performance of the system was evaluated using three different data set including WPAFB 2009 images, drone camera and GoPro images captured at Medipol University and its surroundings and data set providing by TUSAŞ. Throughout the thesis, “*GoPro Dataset*” refers to the images captured at Medipol University using drone and GoPro and “*Small Dataset*” refers to images providing by TUSAŞ. The tracking results and fps comparisons will explained in Section 5.4. The sample of the WPAFB 2009 data set used for the evaluation of the overall system are shown in the Figure 33



Figure 33: Sample of the WPAFB 2009 Data set [4]

The comparisons of serial and multithreaded overall system are shown in Table 22. The total elapsed time of serial version of overall system is 289 second while the multithreaded version is approximately 189 seconds for processing 261 frames. The frame rate of serial version is 0.9 fps and the frame rate of the multithreaded version is 1.4 fps. The simulations are done on Xavier. Also, the feature matcher module can be run on both CPU and GPU.

Table 22: The comparisons of serial and multithreaded overall system

	Serial WAAS System	Multithreaded WAAS System
Feature Matcher on CPU	0.9 fps	1.38 fps
Feature Matcher on GPU	1.02 fps	1.31 fps

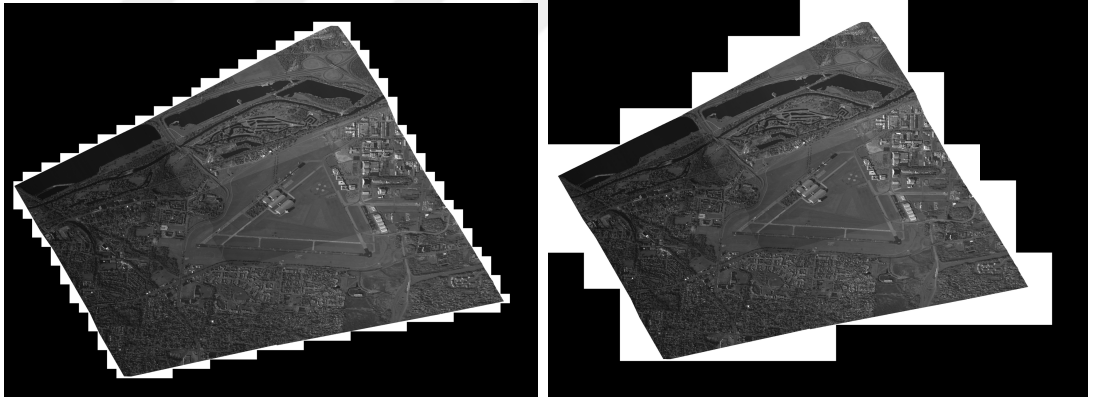
Both super-resolution and object tracking models are optimized using TensorRT. Optimized models are integrated to overall WAAS system. The results and comparisons of overall WAAS system with original model and optimized models are shown in Table 23. The frame rate of slowest system which is the serial version using original models is 0.9 fps whereas the frame rate of fastest system which is multithreaded version using optimized model with Tensorrt fp16 is 2.58 fps. The

results in the Table 23, the resolution of the area of interest is 432x704 and the data set used for simulation is WPAFB 2009 [4].

Table 23: The comparisons of serial and multithreaded overall system

Model	Serial		Multithreaded	
	CPU	GPU	CPU	GPU
Original	0.9 fps	1.02 fps	1.38 fps	1.31 fps
TensorRT Static	1.23 fps	1.43 fps	2.08 fps	2.15 fps
TensorRT FP16	1.36 fps	1.63 fps	2.49 fps	2.58 fps

The registered frames are shown in the Figure 34 and the cropped frames are shown in Figure 35. In order to align and crop frames together and identify the coordinates to crop the frames, the feature matching module is used. Using the estimated homography, frames are overlapped each other and cropped the are being observed.



(a) High-resolution Frame

(b) Low-resolution Frame

Figure 34: Registered WPAFB 2009 images

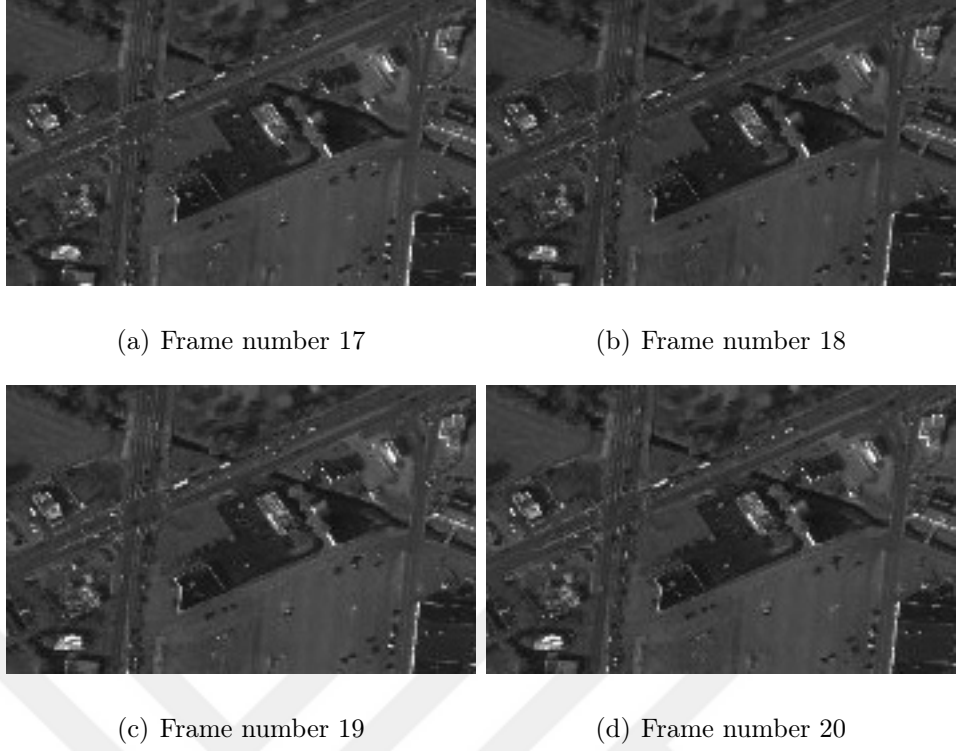


Figure 35: Cropped area of interest

The output of the super-resolution for the frame number 85 is shown in Figure 36. The detected moving object in high-resolution frames are shown in the Figure 37. The object tracking model takes high-resolution frames as an input. The inputs are for time t and $t - 1$.



Figure 36: The super-resolution output for frame number 85

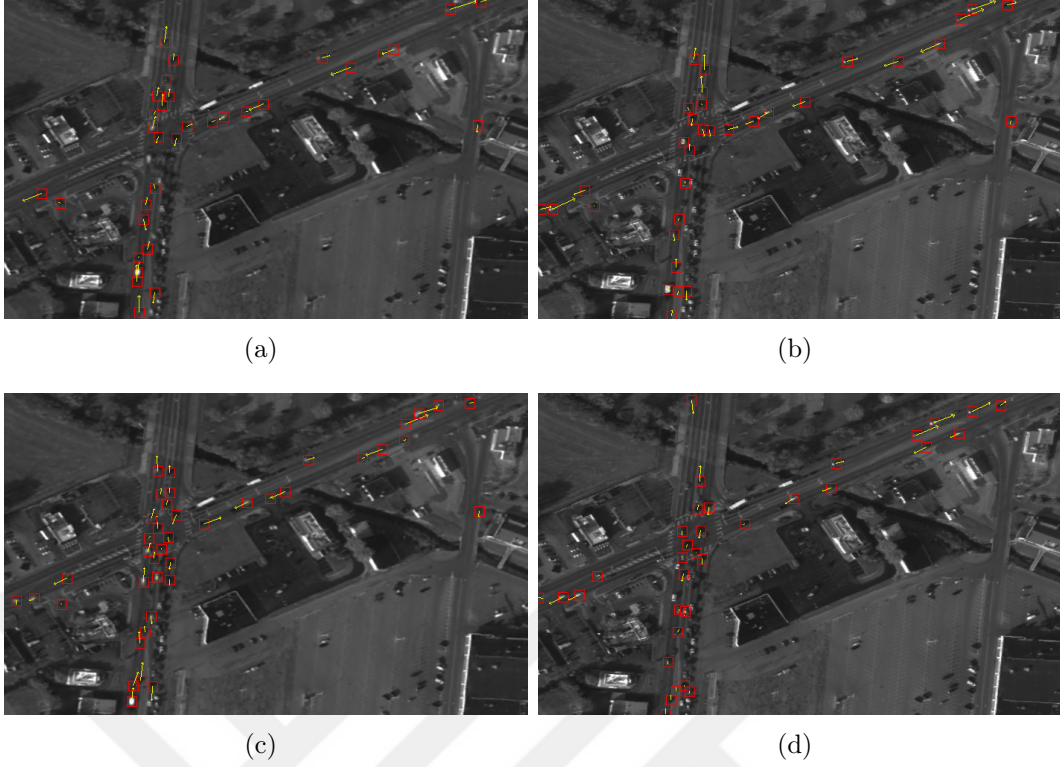


Figure 37: Tracking results of cropped area

Afterward, both *GoPro Dataset* and *Small Dataset* were used in the simulations of the overall system. The comparison of both results are shown in Table 24. The simulations are performed at Xavier and the results are for GPU-Multithreaded version of the overall system.

Table 24: The comparisons of GoPro Dataset and Small Dataset

Dataset	Multithreaded Overall System	
	CPU	GPU
GoPro Dataset	1.51 fps	1.38 fps
Small Dataset	1.76 fps	1.56 fps

The four sample of tracking results of GoPro Dataset are shown in Figure 38 and four sample of tracking results of Small Dataset are shown in Figure 39. The stitched mosaic image produced using Small Dataset is shown in Figure 40.

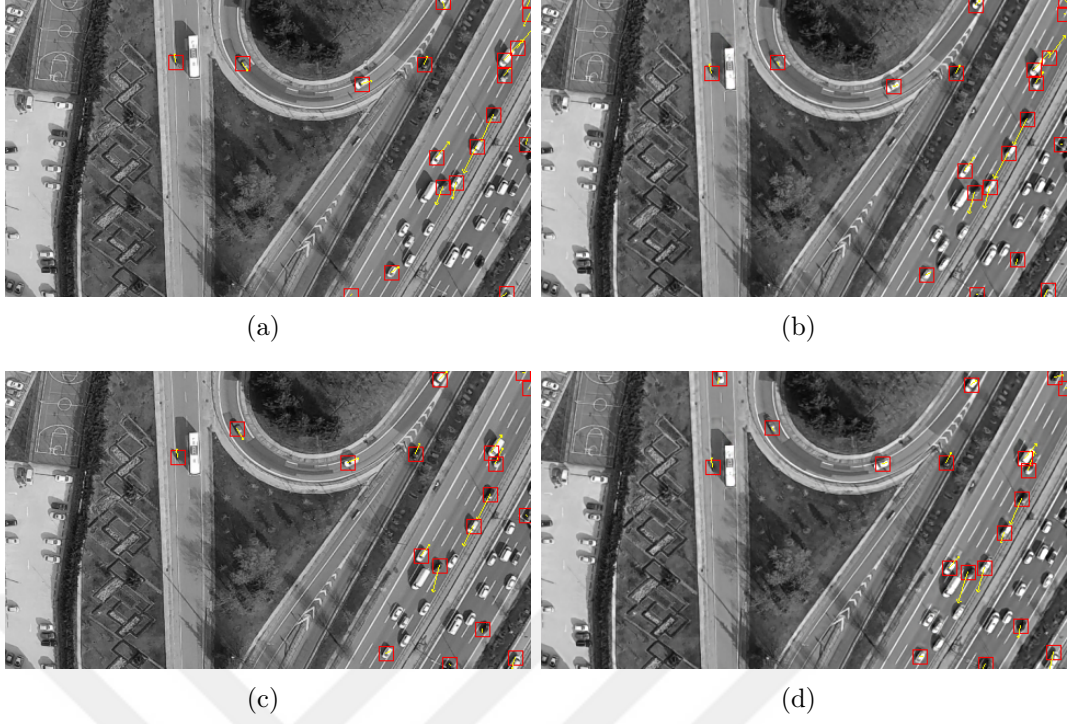


Figure 38: Tracking results of cropped area of GoPro Dataset

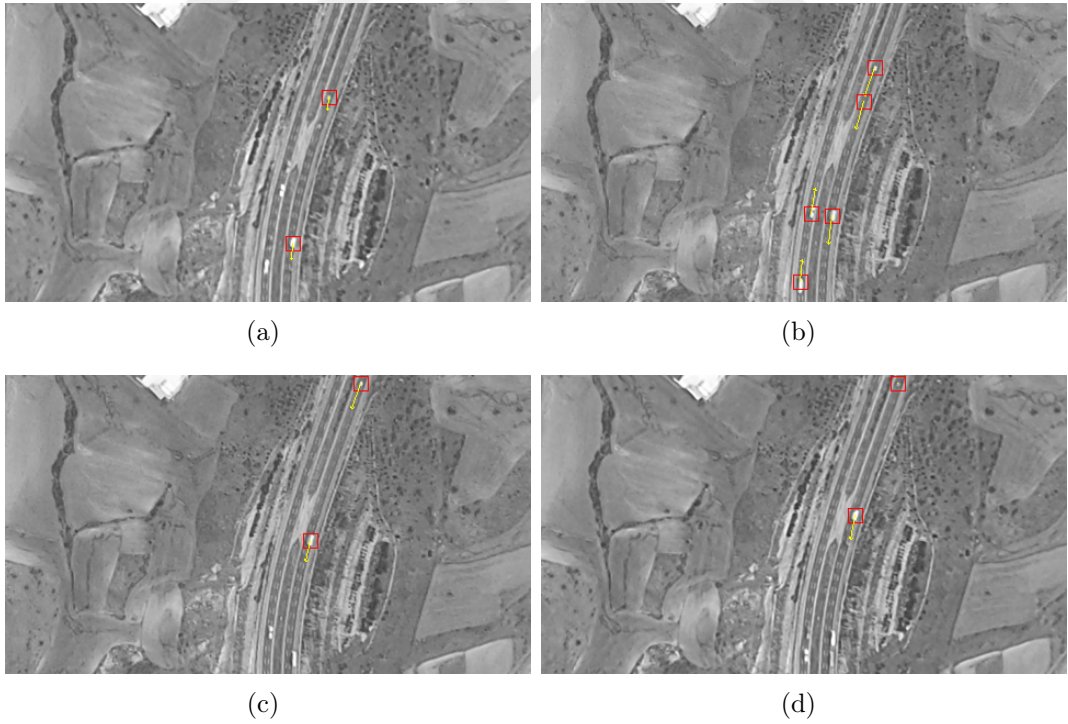


Figure 39: Tracking results of cropped area of Small Dataset



Figure 40: The mosaic image produced using small dataset

5.5 Comparing with Other Works

The comparing results obtained in this thesis study with others works are discussed in this section. In [10], Zhou et al. propose a UAV patrol system based on panoramic image stitching and object detection system. Their system proposed in [10] can not process video frames captured from a moving drone. The experiments setup mentioned in the paper is not an on-device UAV system. The study does not mention if any acceleration methods were implemented or their modules are

running in real-time. The image stitching method that they used, is not give a good results when the objects are moving. Clearly seen in the results they presented, when the objects are moving there are ghosts of the moving object in the panoramic image. In their study, object detection module runs at *7FPS* on a computer with an Intel Core i7-8700k, 16GB RAM and an NVIDIA GeForce GTX1080Ti GPU. The system designed in this study is runs an on-device WAS system which contains NVIDIA Jetson AGX Xavier edge computing device in real-time. .

The study done by Wei et al. proposes an UAV platform that includes image processing modules such as video frame registration, moving object detection and panoramic image stitching in [9]. Their experiments are implemented in MATLAB R2016a on an Intel Core i5-7300HQ, 2.5GHz CPU, 8GB RAM computer. The image registration is aimed for identify the coordinates of the corresponding feature points in consecutive images based on feature matching approach. Then, the homography matrix is obtained between two frames to aligning and stitching frames together. Using the stitched panoramic video frames, the moving objects are detected in the large area. They did not provide any numerical information related to the frame rate or accuracy of object tracking module. They only show the experiment results in visually and show that their proposed method can work in both less-textured and well-textured backgrounds. In this study, in order to increase the resolution of the frames of the observed area, the super-resolution module is implemented. This WAAS system proposed in this thesis runs in real-time and the results are in high-resolution.

CHAPTER VI

CONCLUSION

This thesis proposes a wide area of aerial surveillance systems, including accelerated image processing modules. The modules in the WAAS system are image stitching, object detection and tracking, and super-resolution. GPUs and multi-core programming have proven to be effective in accelerating the image processing module of a wide area aerial surveillance system. The on-device system designed with the NVIDIA Jetson AGX Xavier edge computing device was able to process a large amount of aerial imagery in real-time, meeting the system's requirements.

This approach has several advantages, including the ability to perform image processing on-device without the need for a remote server, which can reduce latency and improve the overall performance of the surveillance system. Using GPUs and multi-core programming also allows for better utilization of the available computing resources, enabling the system to handle larger volumes of data and more complex algorithms.

Overall, the results of this study that the potential of using GPU-based acceleration and multi-core programming to improve image processing performance in wide area aerial surveillance systems. This approach offers a promising solution for addressing the challenges of real-time image processing in such systems and has the potential to be applied to other applications that require high-speed image processing. The frame rate of slowest system which is the serial version using original models is 0.9 fps whereas the frame rate of fastest system which is multithreaded version using optimized model with TensorRT fp16 is 2.58 fps.

REFERENCES

- [1] S. W. Tesfay, Z. G. Demirdag, H. F. Ugurdag, and H. F. Ates, “Hybrid CPU-GPU acceleration of a multithreaded image stitching algorithm,” in *Proceeding of International Conference on Computer Science and Engineering (UBMK)*, pp. 468–473, 2022.
- [2] “MAMI: Minor Area Motion Imagery Dataset.” Available: <https://www.sdms.afrl.af.mil/index.php?collection=mami2013>, Accessed: 2022-12-05.
- [3] “VisDrone-Dataset: The Vision Meets Drones.” Available: <https://github.com/VisDrone/VisDrone-Dataset>, Accessed: 2022-12-05.
- [4] “AFRL, Wright-Patterson Air Force Base (WPAFB 2009) Dataset.” Available: <https://www.sdms.afrl.af.mil/index.php?collection=wpafb2009>, Accessed: 2022-12-19.
- [5] S. Sangsuwan, P. Maksin, P. Popattanachai, C. Musana, and A. Aobpaet, “The investigation of viewing angle effects on ground sampling distance of thaichote satellite imagery.” Available: <http://learn.gistda.or.th/wp-content/uploads/2017/06/GISTDA-Research-Space-2557-The-investigation-of-viewing.pdf>, Accessed: 2022-12-22.
- [6] “Opencv super resolution with deep learning.” Available: <https://pyimagesearch.com/2020/11/09/opencv-super-resolution-with-deep-learning/>, Accessed: 2022-12-12.
- [7] M. Parker, “Chapter 29 - Implementation With GPUs,” in *Digital Signal Processing 101 (Second Edition)*, pp. 387–393, Newnes, second edition ed., 2017.
- [8] E. Blasch, G. Seetharaman, S. Suddarth, K. Palaniappan, G. Chen, H. Ling, and A. Basharat, “Summary of methods in wide-area motion imagery (WAMI),” *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 9089, p. 90890C, 2014.
- [9] Q. Wei, S. Lao, and L. Bai, “Panorama stitching, moving object detection and tracking in UAV videos,” in *Proceeding of International Conference on Vision, Image and Signal Processing (ICVISIP)*, pp. 46–50, 2017.
- [10] Y. Zhou, T. Rui, Y. Li, and X. Zuo, “A UAV patrol system using panoramic stitching and object detection,” *Computers & Electrical Engineering*, vol. 80, p. 106473, 2019.
- [11] M. Heikkilä and M. Pietikäinen, “An image mosaicing module for wide-area surveillance,” in *Proceedings of the Third ACM International Workshop on Video Surveillance & Sensor Networks*, p. 11–18, Association for Computing Machinery, 2005.

- [12] C. G. Harris and M. J. Stephens, “A combined corner and edge detector,” in *Alvey Vision Conference*, 1988.
- [13] Y. Li, Y. Wang, W. Huang, and Z. Zhang, “Automatic image stitching using sift,” *Proceeding of International Conference on Audio, Language and Image Processing*, pp. 568–571, 2008.
- [14] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, “Speeded-up robust features (SURF),” *Comput. Vis. Image Underst.*, vol. 110, p. 346–359, jun 2008.
- [15] E. Rosten and T. Drummond, “Machine learning for high-speed corner detection,” in *European Conference on Computer Vision*, 2006.
- [16] A. V. Kulkarni, J. Jagtap, and D. V. K. Harpale, “Object recognition with ORB and its implementation on FPGA,” vol. 3, 2013.
- [17] C.-H. Lin, Y. M. Tsai, W. Wang, and L.-G. Chen, “GPU-accelerated high-resolution image stitching with better initial guess,” in *Proceeding of IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–3, 2018.
- [18] O. Nakov, E. Mihaylova, M. Lazarova, and V. Mladenov, “Parallel image stitching based on multithreaded processing on GPU,” in *Proceeding of International Conference on Intelligent and Innovative Computing Applications (ICONIC)*, pp. 1–5, 2018.
- [19] T. Xu, P. Cockshott, and S. Oehler, “Acceleration of stereo-matching on multi-core CPU and GPU,” in *Proceedings of IEEE Intl Conf on High Performance Computing and Communications, Proceedings of IEEE Intl Symp on Cyberspace Safety and Security, Proceedings of IEEE Intl Conf on Embedded Software and Syst (HPCC,CSS,ICSS)*, pp. 108–115, 2014.
- [20] C. Lai and M. Huang, “Hybrid implementation of image stitching on computers with GPUs,” in *Proceedings of the International Conference on Image Processing, Computer Vision, and Pattern Recognition (IPCV)*, p. 3, 2016.
- [21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.
- [22] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [23] M. Dextre, O. Rosas, J. Lazo, and J. C. Gutiérrez, “Gun detection in real-time, using YOLOv5 on Jetson AGX Xavier,” in *Proceeding of XLVII Latin American Computing Conference (CLEI)*, pp. 1–7, 2021.
- [24] Y. Xu, X. Zhou, S. Chen, and F. Li, “Deep learning for multiple object tracking: a survey,” *IET Computer Vision*, vol. 13, no. 4, pp. 355–368, 2019.
- [25] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, “Natural language processing (almost) from scratch,” *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, 2011.

- [26] C. Du, J. Yuan, J. Dong, L. Li, M. Chen, and T. Li, “GPU based parallel optimization for real time panoramic video stitching,” *ArXiv*, vol. abs/1810.03988, 2018.
- [27] A. Asaduzzaman, A. Martinez, and A. Sepehri, “A time-efficient image processing algorithm for multicore/manycore parallel computing,” in *Southeast-Con*, pp. 1–5, 2015.
- [28] M. J. Mišić, M. Đurđević, and M. V. Tomašević, “Evolution and trends in gpu computing,” in *Proceedings of the 35th International Convention MIPRO*, pp. 289–294, 2012.
- [29] M. Hopf and T. Ertl, “Hardware accelerated wavelet transformations,” in *Data Visualization* (W. C. de Leeuw and R. van Liere, eds.), pp. 93–103, Springer Vienna, 2000.
- [30] A. Yaromenok, “Diplib - digital image processing library,” 2003. Available: <http://sourceforge.net/projects/diplib>, Accessed: 2022-12-05.
- [31] J. Fung and S. Mann, “OpenVIDIA: Parallel GPU Computer Vision,” in *Proceedings of the 13th Annual ACM International Conference on Multimedia*, p. 849–852, Association for Computing Machinery, 2005.
- [32] R. Strzodka and A. Telea, “Generalized Distance Transforms and Skeletons in Graphics Hardware,” in *Eurographics / IEEE VGTC Symposium on Visualization* (O. Deussen, C. Hansen, D. Keim, and D. Saupe, eds.), The Eurographics Association, 2004.
- [33] S. Saxena, N. Sharma, and S. Sharma, “Image processing tasks using parallel computing in multi core architecture and its applications in medical imaging,” *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 2, no. 4, pp. 1896–1900, 2013.
- [34] C. G. Kim, J. Kim, and D. Lee, “Optimizing image processing on multi-core CPUs with intel parallel programming technologies,” *Multimedia Tools and Applications*, vol. 68, 2014.
- [35] G. Georgis, G. Lentaris, and D. Reisis, “Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution,” *J. Real-Time Image Process.*, vol. 16, no. 4, p. 1207–1234, 2019.
- [36] J. Zhao, Y. Yang, and G. Li, “Real-time image processing system based on multi-core processor,” in *Proceeding of International Symposium on Intelligent Information Technology Application*, vol. 1, pp. 329–332, 2009.
- [37] G. Wang, Z. Zhai, B. Xu, and Y. Cheng, “A parallel method for aerial image stitching using ORB feature points,” in *Proceeding of IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, pp. 769–773, 2017.

- [38] T. Blattner, W. Keyrouz, J. Chalfoun, B. Stivalet, M. Brady, and S. Zhou, “A hybrid CPU-GPU system for stitching large scale optical microscopy images,” in *Proceeding of 43rd International Conference on Parallel Processing*, pp. 1–9, 2014.
- [39] S. Yu, Z. Yao, A. Gholami, Z. Dong, M. W. Mahoney, and K. Keutzer, “Hessian-aware pruning and optimal neural implant,” *CoRR*, vol. abs/2101.08940, 2021.
- [40] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, “Learning efficient convolutional networks through network slimming,” *CoRR*, vol. abs/1708.06519, 2017.
- [41] S. Lym, E. Choukse, S. Zangeneh, W. Wen, S. Sanghavi, and M. Erez, “Prune-Train: fast neural network training by dynamic sparse model reconfiguration,” pp. 1–13, 2019.
- [42] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: An efficient alternative to SIFT or SURF,” in *Proceeding of International Conference on Computer Vision*, pp. 2564–2571, 2011.
- [43] D. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.
- [44] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, “Parallel computing experiences with CUDA,” *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.
- [45] E. Cervera, “GPU-accelerated vision for robots: Improving system throughput using OpenCV and CUDA,” *IEEE Robotics & Automation Magazine*, vol. 27, no. 2, pp. 151–158, 2020.
- [46] G. Bradski and A. Kaehler, *Learning OpenCV : computer vision with the OpenCV library*. O’Reilly, 1st ed., 2008.
- [47] 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.*, vol. 21, no. 5, 2022.
- [48] H. Motorcu, H. F. Ates, H. F. Ugurdag, and B. K. Gunturk, “HM-Net: A regression network for object center detection and tracking on wide area motion imagery,” *IEEE Access*, vol. 10, pp. 1346–1359, 2022.
- [49] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, “Pruning Filters for Efficient ConvNets,” *CoRR*, vol. abs/1608.08710, 2016.
- [50] Z. Huang, T. Zhang, W. Heng, B. Shi, and S. Zhou, “Rife: Real-time intermediate flow estimation for video frame interpolation,” in *European Conference on Computer Vision*, 2020.

- [51] H. U. Suluhan, H. F. Ates, and B. K. Gunturk, "Dual camera based high spatio-temporal resolution video generation for wide area surveillance," in *18th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–8, 2022.
- [52] Y. He, P. Liu, Z. Wang, and Y. Yang, "Pruning filter via geometric median for deep convolutional neural networks acceleration," *CoRR*, vol. abs/1811.00250, 2018.
- [53] "VisDrone: The Vision Meets Drones." Available: <http://aiskyeye.com/>, Accessed: 2022-12-05.
- [54] Y. Kortli, S. Gabsi, L. F. L. Y. Voon, M. Jridi, M. Merzougui, and M. Atri, "Deep embedded hybrid CNN-LSTM network for lane detection on NVIDIA Jetson Xavier NX," *Knowledge-Based Systems*, vol. 240, p. 107941, 2022.
- [55] J. Jeon, S. Jung, E. Lee, D. Choi, and H. Myung, "Run your visual-inertial odometry on NVIDIA Jetson: Benchmark tests on a micro aerial vehicle," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5332–5339, 2021.
- [56] H. A. Abdelhafez, H. Halawa, K. Pattabiraman, and M. Ripeanu, "Snowflakes at the edge: A study of variability among NVIDIA Jetson AGX Xavier boards," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, p. 1–6, Association for Computing Machinery, 2021.
- [57] "Jetson AGX Xavier Developer Kit." Available: <https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit>, Accessed: 2022-12-05.
- [58] "Libtorch: C++ Distribution of Pytorch." Available: <https://pytorch.org/cppdocs/installing.html>, Accessed: 2022-12-05.
- [59] "Cprofile library documentation." Available: <https://docs.python.org/3/library/profile.html#module-cProfile>, Accessed: 2022-12-06.
- [60] "time module of python documentation." Available: <https://docs.python.org/3/library/time.html>, Accessed: 2022-12-06.
- [61] "Intel vtune profiler." Available: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.kir8q1>, Accessed: 2022-12-06.

VITA

Zeynep Gülbeyaz Demirdağ

Education

- Feb. 2020 to Dec. 2022: M.Sc. in Electrical & Electronics Engineering, Özyeğin University.
- Sep. 2016 to Jun. 2019: B.Sc. in Electrical & Electronics Engineering, KTO Karatay University.
- Sep. 2011 to Jun. 2015: B.Sc. in Metallurgy & Materials Engineering, Selçuk University.
- Sep. 2007 to Jun. 2011: Aydınlikevler Anatolian High School, Ankara.

Publications

- **Hybrid CPU-GPU Acceleration of a Multithreaded Image Stitching Algorithm** S.W. Tesfay, Z.G. Demirdag, H.F. Ugurdag, H.F. Ates
Proceeding of International Conference on Computer Science and Engineering (UBMK), IEEE Xplore, 2022