Exzellenzcluster
Cognitive Interaction Technology
Kognitronik und Sensorik
Prof. Dr.-Ing. U. Rückert

Heterogeneous Computing Systems for Vision-based Multi-Robot Tracking

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

M.Eng. Arif Irwansyah

Referent:	Prof. Dr.-Ing. Ulrich Rückert
Korreferent:	Prof. Dr.-Ing. Franz Kummert

Tag der mündlichen Prüfung: 12.09.2017

Bielefeld / September 2017

Acknowledgments

Firstly, I would like to express my profound gratitude to Prof. Dr.-Ing. Ulrich Rückert and Dr.-Ing. Mario Porrmann for their invaluable support, motivation, and encouragement throughout my doctoral thesis. I have learned and gain many experiences from them during my work in Cognitronics and Sensor Systems Group. I would also like to thank Jens Hagemeyer for his great support and help when I met difficulties in technical and scientific problems. I thank Cordula Heidbrede for her great support and help in all administration and non-technical aspects that I needed. I thank Omar W Ibraheem, for his brothership, companion, and support during my journey in Bielefeld. Our research collaboration that we have done together have significantly accelerated my work.

I would also like to thank Prof. Dr.-Ing. Franz Kummert, who accepted to review my thesis. Additionally, many thanks to the member of the examination committee Prof. Dr. Elisabetta Chicca and Dr. rer. nat. Thies Pfeiffer, for their participations as the chairman and the examiner, respectively.

I am also highly thankful to the past and current members of our research group. I am especially thankful to my colleagues Andry Tanoto, René Zorn, Muhammad Shahzad, and Meysam Peykanu. My sincerest appreciation goes out to all those who have contributed directly and indirectly to the completion of this thesis.

I am as ever, especially indebted to my parents, Mrs. Mardiyah, Mrs. Sa'diyah and Mr. Djauhari for their love and support throughout my life. Without exception, I am also highly thankful to my wife's parent Mr. Rubiyanto and Mrs. Susiyati for their support and prayer. I am really grateful to all of you.

Special thanks to my beloved wife Siti Rokhmawati, for all your love and support. You have shown unrelenting care and support throughout this challenging endeavor. I am also thankful to my precious children Dayyinah, Tsabita, and Hafidz. Finally, Alhamdulillah, all praise to Him, for His guidance and allowing me to achieve all these amazing things.

Bielefeld, September 2017

Arif Irwansyah

Abstract

Vision-based robot tracking is commonly used for monitoring and debugging in single- and multi-robot environments. Currently, most of the established vision-based multi-robot tracking systems are based on the implementations of a general purpose central processing unit (CPU) in the computer. These solutions are not feasible for use-cases with large frame sizes, multiple cameras, and a large number of robots to be tracked. The most common solution to handle the increasing number of cameras and robots is the addition of extra computers. As an alternative, hardware accelerators such as field programmable gate arrays (FPGAs) and general purpose graphic processing unit (GPU) can be used to release the host computer from computation-intensive tasks like vision processing through their high inherent parallelism. FPGAs and GPUs offer different approaches to maximize the performance of a computing system. An FPGA is an integrated circuit (IC) designed to be hardware reconfigurable after manufacturing. It is purpose-built hardware that can be used for specific algorithms according to the user's applications to obtain higher computing performance. Meanwhile, the advantages of the GPU as an accelerator rely on its architecture, which consists of a large number of lightweight cores and applies a single instruction multiple threads (SIMT) model for executing programs. This thesis emphasizes the implementations of two distinct heterogeneous computing systems for a vision-based multi-robot tracking application, encompassing the use of FPGAs and GPUs as hardware accelerators. It aims to determine which architecture offers the optimum solution, in terms of the detection performance, computing performance, and power efficiency.

The proposed heterogeneous computing systems combine the advantages of a CPU with the benefits of an FPGA or a GPU. The designs attempt to efficiently handle computationally intensive vision-based multi-robot tracking algorithms. The FPGA and GPU are utilized as hardware accelerators, processing the portion of the algorithm that is computationally intensive to detect the robots' locations. Meanwhile, the CPU is used as the processor in the host PC for post-processing and display. In the FPGA-based accelerated computing system, a complete design for detecting each robot's location is implemented, comprising a multi-camera frame grabber and IP cores for object segmentation, edge filtering, and circle detection. The number of cameras used in the proposed design is scalable. This design presents three basic configurations, which differ in the number of streaming hardware accelerators and in the parallelism of the implementation. Additionally, two unique architectures for FPGA-based circle detection for multi-robot tracking, using the combination of the circular Hough transform (CHT)-graph cluster algorithm and circle scanning window (CSW) technique-graph cluster algorithm, are proposed and implemented. Regarding the implementation of the GPU as a hardware accelerator, the proposed GPU-based computing system is designed to improve the computational performance by utilizing the benefits of the

GPU's architecture, particularly its thousands of lightweight processing cores. The algorithm's implementation in the GPU includes object segmentation (debayer, RGB to HSV color conversion, and color masking operations), edge filtering, and circle detection (CHT and CSW). The FPGA/GPU performs the computationally intensive tasks for a full resolution image (a maximum of 2048 × 2048 pixels), while the CPU executes the post-processing algorithm for small sub-images (40 × 40 pixels). To obtain the robots' orientations and IDs, the advantage of the multi-core architecture of the CPU is employed to process all of the sub-images in a multi-thread approach.

The results of this thesis show that the FPGA- and GPU-based hardware accelerators greatly enhance the computational performance of the computing system for vision-based multi-robot tracking. The maximum frame rate in the FPGA implementation is optimized by utilizing four streaming hardware accelerators, working in parallel. Meanwhile, the high-performance of the GPU implementation is achieved by employing its many cores. According to the experiments, both the FPGA-based and GPU-based designs present highly accurate performance. The design and its algorithm can provide a highly accurate performance for the localization of multiple robots with a typical detection performance (precision and recall) of 99 %. Additionally, both the FPGA and GPU hardware accelerators offer higher power efficiency than the CPU. They can increase the computation performance per watt of the computing system. Finally, quantitative and qualitative parameters (e.g. computational performance, power consumption, power efficiency, and developing time) are analyzes more details to determine which technology is more suitable for the vision-based multi-robot tracking application.

Contents

1 Introduction

Vision-based localization and tracking is an approach that is frequently used for monitoring and debugging in single- and multi-robot environments, e.g., for the evaluation of navigation algorithms and team behavior in multi-robot experiments. A main feature of a vision-based robot tracking system is its ability to cope with different types of robots, because it can operate without the need for additional components such as electronic sensors to be installed on the robots. Therefore, this system is widely used in robotic laboratories for analyzing and debugging the behavior of multiple robots, for both homogeneous and heterogeneous types of mobile robots.

In general, a vision-based robot tracking system is usually used to provide ground-truth data for performance analysis [103]. This data can be very beneficial for further analysis or research such as to learn the behavior of a robot, measure the performance of the scenario implemented for a group of robots, or simply to test the function of an applied algorithm in a robot. In some applications, a vision-based robot tracking system can also be applied to support robots with accurate position information during runtime because the system emulates the function of an indoor GPS for every robot. In other words, this system is considerably advantageous because it flexibly allows the robots to compute their own positions, which make more resources available for performing their own tasks.

To complete its operation, a vision-based robot tracking system uses one or more cameras as the video input source. The video data are processed by a computing system to extract the useful information (e.g., the locations, orientations, and identities (IDs) of robots). A computationally intensive vision processing algorithms is required to extract the relevant information. Additionally, the computational requirements increase with the number of tracked robots, video frame size, and number of operated cameras. According to the above operations and conditions, the implementation of a vision-based robot tracking system imposes three challenges to be considered. First, rapid processing is needed for applications that require real-time robot tracking. The second challenge is the scalability with respect to the number of cameras. On one hand, scalability is often required to increase the possible field of view, which may be restricted by the environmental conditions of the setup. On the other hand, scalability makes it possible to increase the total resolution, if required. Third, the system must be able to process many robots (more than 50 robots) simultaneously. This is a requirement imposed by various multi-robot experiments.

Previously, most of the established computing systems for vision-based robot tracking were based on the general purpose central processing unit (CPU) in the host PC. These systems focused on developing the software architecture and algorithms implemented in CPU-based computing systems rather than investigating the use of alternative hardware accelerators. For resource-efficient embedded applications or use cases with large frame sizes, multiple cameras and a high number of tracked robots, PC-based solutions are often not feasible. The most common solution to handle the increasing number of cameras and robots is the addition of extra PCs, as was done in [74; 103; 104] to cover a larger robot arena. Unfortunately, this approach can significantly increase the energy consumption, total system complexity, and overall system costs. Therefore, finding an alternative approach that utilizes other hardware architectures has become inevitable. There has been some initial work in vision-based robot tracking using alternative hardware accelerators. Yet, these studies were mostly still in the design or prototyping phase, which involved only a single camera, a low-resolution video input, and a small number of tracked robots. Additionally, such designs did not generally support a comprehensive solution for multi-robot tracking applications, which have been well-supported in CPU-based computing systems. Therefore, this thesis attempts to fill the gap in the area of hardware-accelerated computing systems for vision-based multi-robot tracking by presenting the combination of a CPU and an alternative hardware accelerator.

In the area of vision processing, there are several types of hardware accelerators, e.g., DSPs, GPUs, FPGAs, and multi-core CPUs. Of course, every hardware architecture has distinct advantages and disadvantages, which depend on the application requirements. For instance, a CPU has various advantageous. First, it is ideal for complex scalar processing and very suitable for executing complicated operations on a single or a few streams of data. Second, a CPU is able to accommodate its integration with various operating systems (OS). Third, it also provides a well-known software development environment and I/O port access for sensors and devices (e.g., a camera, display, or network). As a result, CPUs perform essential roles within computing systems, especially in terms of comprehensive vision processing applications. However, despite its positive advantages, a CPU still has a weakness because its parallel processing capabilities are limited by the number of processing cores. In contrast with CPU, GPUs and FPGAs are specialized devices with highly parallel architectures. Both can enhance the computing performance for some vision processing algorithms. The former (GPU) consists of hundreds or even thousands of small yet efficient cores, designed to handle multiple tasks (threads) simultaneously. Meanwhile, the latter (FPGAs) offers a parallel hardware structure that is re-programmable according to a specific user application.

Nowadays, heterogeneous (hybrid) computing systems are being widely used to support highly computationally intensive applications. Heterogeneous computing refers to a system that employs more than one different hardware accelerators or processing core to increase its computational performance. In heterogeneous computing systems, diverse types of processors or hardware accelerators cooperate to accelerate the computational tasks. Heterogeneous computing systems typically combine CPUs with hardware accelerators such as FPGAs and/or GPUs. The collaboration between a CPU as the processor in the host PC and some hardware accelerator (FPGA or GPU) can increase the parallel computational capability of the computing system. These hybrid systems potentially reduce the power consumption and maximize the computing performance.

In the context of heterogeneous computing systems, this thesis focuses on the implementation and evaluation of hardware accelerator (FPGA and GPU) environments rather than the development of an algorithm for a CPU. It emphasizes the implementations of two distinct heterogeneous computing systems for vision-based multi-robot tracking applications, encompassing the use of the FPGA and GPU as hardware accelerators. The main objective is to efficiently handle computationally intensive applications like vision processing through their high inherent parallelism. In particular, this thesis implements FPGA- and GPU-accelerated heterogeneous computing systems, compares the results, and measures the advantages that can be achieved by both computing systems for vision-based multi-robot tracking applications. In doing so, this thesis focuses on the system architecture, detection performance, computing performance, and power efficiency. Based on examinations and analyses, a suitable architecture is proposed for a vision-based multi-robot tracking computing system.

1.1 Contributions

In heterogeneous computing systems, using FPGAs and GPUs as hardware accelerators offers distinctive approaches to maximize the computing performance of systems. FPGAs merely highlight purpose-built and customized design architectures for specific algorithms with low power, low latency, and high computing performance. They deliver hardware that is re-programmable with massive parallel structures according to user applications. An FPGA can also be reprogrammed to have a direct interconnection with an I/O port, including a direct interconnection with a single or multiple cameras. Likewise, the use of a GPU as a hardware accelerator also provides significant benefits. It relies on a large number of lightweight programmable cores (hundreds or even thousands) and is designed to execute programs in a single instruction multiple thread (SIMT) fashion. However, a GPU's architecture is limited by a fixed hardware structure that depends on sequential operations running on those programmable cores with associated register and bus width limitations.

Considering the differences between an FPGA and a GPU, this work aims to compare and analyze FPGA-CPU and GPU-CPU computing systems, to find the optimal system for multi-robot tracking applications. The main contributions of this thesis are as follows:

- An FPGA-based hardware accelerated computing system for multi-robot tracking using multiple cameras.
- A GPU-based hardware accelerated computing system for multi-robot tracking using multiple cameras.
- Two distinct unique architectures for FPGA-based circle detection for multi- robot tracking application. The first one integrates a combination of the circular Hough transform (CHT) and graph cluster algorithms. The second architecture combines the circle scanning window (CSW) technique and graph cluster algorithm.
- Performance analysis and evaluation of the advantages and bottlenecks for FPGA-based and GPU-based multi-robot tracking systems.
- Accuracy and power consumption analysis from both FPGA-CPU and GPU-CPU computing systems to find the optimum architecture.

1.2 Thesis Organization

Chapter 2 presents an overview of the main concept of a vision-based multi-robot tracking system. This chapter also discusses the state of the art of vision-based robot tracking systems, using both CPU-based and hardware-accelerated computing systems. The focus is understanding the design of the existing computing systems, particularly their strengths and weaknesses. This is followed by descriptions of the theoretical backgrounds and architectures of multi-core CPU, GPU, and FPGA hardware accelerators.

Chapter 3 delineates a heterogeneous computing system as an alternative approach for vision-based multi-robot tracking application. Both FPGA-CPU and GPU-CPU architectures are explored to determine their advantages and challenges. Finally, the implementations of algorithms for vision-based multi-robot tracking applications are explored.

Chapter 4 presents an implementation of heterogeneous FPGA-CPU computing systems for vision-based multi-robot tracking. The advantages of the massive parallel structure and customizable design of the FPGA architecture are used to increase the computing performances. Three basic configurations for FPGA-based video processing are presented, which differ in the number of hardware accelerators and thus in the parallelism of the implementation. Some video processing modules are implemented on the FPGA to ensure the complete proposed system. These modules include multi-camera frame grabber, object segmentation, edge filter, and circle detection modules as FPGA hardware accelerators to obtain the maximum advantages of using the FPGA technology. Two unique architectures for FPGA-based circle detection for multi-robot tracking are presented and evaluated. The first integrates a combination of the CHT and graph cluster algorithms. The second architecture combines the circle scanning window (CSW) technique and a graph cluster algorithm.

Chapter 5 proposes the implementation of vision-based multi-robot tracking in heterogeneous GPU-CPU computing systems. The discussion in this chapter begins with descriptions of the proposed GPU-CPU hardware architectures. It is followed by a presentation of the algorithm and its implementation on a GPU using CUDA kernels. This implementation includes object segmentation (debayer, RGB to HSV color conversion, and color masking operations), edge filter, and circle detection algorithms.

Chapter 6 shows the analysis results and comparisons of both FPGA- and GPU-accelerated computing systems. The analysis and comparisons focus on the computing performances, detection performance, and power efficiency. Additionally, a comparison with some related work is also presented.

Finally, chapter 7 summarizes the proposed designs and implementation reports presented in the previous chapters. This chapter also provides a conclusion and an analysis based on the experience obtained during this thesis work.

2 Vision-based Robot Tracking Computing System

This chapter presents a literature review on vision-based multi-robot tracking computing systems and the background concepts of different hardware accelerators used for vision processing applications. The discussion begins with the basic concept of vision-based robot tracking, which is followed by a review of related works on CPU-based and hardware-accelerated computing platforms. Because one of the goals of this thesis is finding the most suitable computing system and optimizing vision-based robot tracking using an existing hardware accelerator, the architectures of multi-core CPU, GPU, and FPGA systems are also outlined in this chapter. These subjects are very important to provide a complete understanding of their individual costs and benefits.

2.1 Basic Concept of Vision-based Robot Tracking System

The main advantage of using a vision-based robot tracking system is that there is no need to install additional components such as an electronic sensor on the mobile robot. The system uses a camera as a video input source, while robots are individually labeled with a specific marker so that each of them can be recognized by the computing system (e.g., computer) through the camera. The computing system processes the video data to extract the useful information (e.g., location, orientation, and ID of the robot). With this advantage, the system is well-matched and able to cope with different types of mobile robots.

To extract the useful information from the video data, some vision processing algorithms must be utilized. Thus, these algorithms become the fundamental operations to identify objects or interpret the content in the video. As shown in Figure 2.1, there are three main steps for the vision-based robot tracking algorithms: object segmentation, robot detection, and post-processing [62]. Indeed, these vision processing algorithms often involve highly computationally intensive operations. Some segmentation algorithms such as for color space conversion, color masking, thresholding, and background subtraction could be implemented to distinguish objects from a background. Subsequently, some shape detection, blob detection, or contour detection algorithms can be applied to detect the robots. Finally, several post-processing operations such as computing the robot orientation and decoding the robot ID can be applied to obtain more detailed and accurate information, as well as an additional function, e.g., recording the video or storing the computed data.

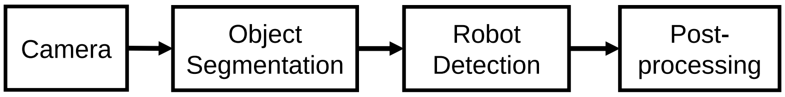


Figure 2.1: Top-level block diagram of a vision-based robot tracking method.

Figure 2.2 illustrates a typical configuration for a vision-based robot tracking system. The system consists of a robot field (arena), robots with markers, a static camera, and a computing system. First, the robot field (arena) refers to the location where the robots are moving or the experiments take place. This arena is typically located indoors and equipped with sufficient lighting. A well-defined lighting condition is very important because this setting frequently influences the capability of the system to detect the robot. Second, a robot marker is a custom symbol with a predefined shape and patterns for the identification of individual robots. Third, the static camera that is attached to the ceiling of the robot lab plays a role in capturing video frames from a top-view perspective. Fourth, the computing system is a set of hardware used to process these video frames by executing the robot tracking algorithms. The camera and computing system are connected with a cable interface, which depends on the type of camera. As an example, a GigE Vision camera is connected to the computing system using an Ethernet cable.

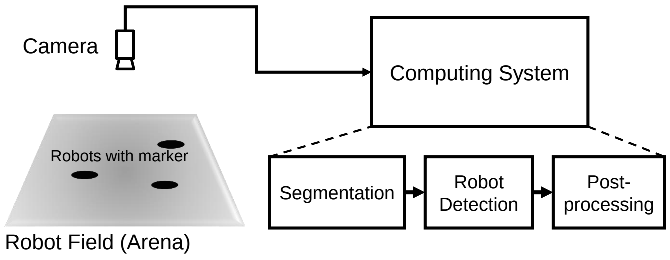


Figure 2.2: Typical configuration of vision-based robot tracking system [62].

Previously, most of the established computing systems for vision-based robot tracking were based on a general purpose CPU. This was because a CPU has various advantageous such as the flexibility to be integrated with an operating system (OS) and a well-known software development environment, along with easy access to I/O ports, sensors, and devices (e.g., camera, display, and network). Accordingly, these existing systems focused on developing a software architecture and implementing algorithms on CPU-based computing systems instead of investigating the use of alternative hardware accelerators. However, some preliminary work has been performed on vision-based robot tracking using alternative hardware accelerators, particularly FPGA- and GPU-based computing systems for accelerating computationally intensive tasks. Therefore, the following sections discuss some of the related work on vision-based computing systems.

2.2 Related Work

Many studies [10; 74; 75; 84; 98; 104] have proposed various vision-based robot tracking systems. They offer different methods to track robots and support different numbers of robots, video frame resolutions, and numbers of cameras. All of the systems referenced above are established systems implemented on CPU-based platforms. Some of the systems that are implemented on FPGA- and GPU-based platforms are mostly still in the design or prototyping phase, which involves the use of only a single camera, low-resolution video input, and small number of tracked robots. FPGA-based system designs can be found in [9; 17; 42; 92; 120], while GPU-based systems are presented in [45; 123]. The following subsections elaborate on the related work in more detail.

2.2.1 CPU-based Computing System

In the first developments of vision-based robot tracking systems, several researchers [10; 75; 98] used a single camera as an input with a low or medium resolution and a small number of tracked robots. These were restricted by the limitations of the camera resolution and CPU performance for processing robot tracking algorithms in real-time.

Lund et al. introduced a simple real-time mobile robot tracking system using a CCD camera, frame grabber card, and tracking algorithm running on a CPU [75]. The system worked by placing the camera above a test field and mounting two LEDs on top of a robot to enable the easy detection of the position and orientation of the robot. This early generation of vision-based systems only supported the tracking of a single robot.

Then, Sirota developed a system to track multiple robots called RoboTracker [98]. It used a camera with a resolution of 1024 × 768 pixels. The CPU-based computing system implemented vision processing algorithms to determine the individual locations and identities of the robots. Each robot was marked with a color-coded marker that uniquely distinguished one robot from another. However, the system did not support orientation detection of the robots.

The Cognachrome Vision System [84] is a low-cost embedded system platform for vision-based tracking. The default tracking resolution is 200 × 250 pixels at 60 frames per second (fps). The system is based on a 32-bit microcontroller (MC68332) connected to a host computer to establish the complete vision system. One of its applications is micro-robot soccer tracking.

Balch et al. presented a system for tracking small insects such as ants [10]. The system is running on a CPU-based platform. It is equipped with a color video camera and a wide-angle lens, as well as a video capture card that can provide 640 × 480 pixel images at 30 fps. A hybrid vision algorithm is used to track multiple ants simultaneously. The system combines color-based tracking and movement-based tracking to detect the insects.

All of the previously discussed vision-based robot tracking systems are designed for a single camera. These designs only allow small numbers of robots to be tracked and support small robot fields. To consider the requirements for tracking many robots in a larger environment, the advanced generation of vision-based robot tracking systems intends to deliver systems that are scalable with respect to the number of cameras and capable of tracking a larger number of robots. Figure 2.3 shows an existing vision-based robot tracking system configuration that utilizes multiple (two) cameras. In this system configuration, the robots in the arena are tracked by more than one overhead camera. Each camera is handled by one CPU-based computing system, and the outputs from the individual computing systems need to be merged for final processing. Several examples of these systems are presented in the following.

Lochmatter et al. developed SwissTrack [74], a vision-based solution for multi-agent tracking. One of its distinctive features is its modular software architecture. It has the ability to add customized modules using the provided interface. These modules extend the functionality of the existing components. The SwissTrack system is capable of tracking up to 50 robots, as well as many insects (e.g., cockroaches) in both single and multiple camera configurations. Its configuration for a single camera consists of a GigE Vision camera with 1032 × 778 pixels and a CPU-based computing system. This computing system is used to process the algorithms for detecting the locations, IDs, and orientations of the robots. It utilizes a blob detection algorithm to detect the locations of the robots, and then implements a nearest-neighbor tracking algorithm to track the

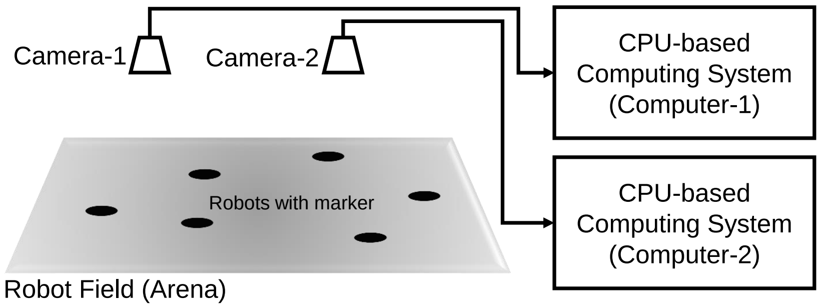

Camera-1
Camera-2
CPU-based Computing System (Computer-1)
CPU-based Computing System (Computer-2)
Robots with marker
Robot Field (Arena)

Figure 2.3: Block diagram of existing vision-based robot tracking using two cameras.

robots. Additionally, SwissTrack supports a multi-camera configuration for a larger arena. To achieve parallel video processing from two cameras, two computers are required, with each running an instance of SwissTrack. Each camera is handled by one computer, while a simple script captures and merges the outputs of the instances, along with recording the merged output video. Although the number of cameras in the system is scalable, each additional camera requires an extra computer.

Another related study was conducted by Zickler et al. They proposed SSL-Vision [123], a vision-based multi-robot tracking system that was intended to be used in the Small Size League (SSL) of RoboCup-Soccer. The SSL-Vision system uses a multi-thread approach on a multi-core CPU, as illustrated in Figure 2.4. The system configuration consists of two Firewire 800 cameras (AVT Stingray F-46C), which provide a 780×580 video stream at 60 Hz, and a multi-core CPU as the main computing system. By default, the number of cameras is two, but it can be extended according to the dimensions of the robot arena. SSL-Vision supports a smaller number of robots compared to the SwissTrack system because it is intended to track the robots in RoboCup Soccer (12 robots). SSL-Vision only uses a single computing system to support the simultaneous image processing of videos from multiple cameras. For processing parallel video frames from multiple cameras, the application is divided into a main thread and several camera threads. The main thread is responsible for the graphical user interface, while each camera thread runs the vision processing algorithm on the respective camera video frames to track the robots. SSL-Vision utilizes the CMVision library [24] to implement color segmentation for robot marker detection.

Faigl et al. [32; 40; 67] introduced SyRoTek, a platform for practical verification in the fields of Robotics and Artificial Intelligence. SyRoTek consists of an arena with real autonomous mobile platforms, communication infrastructures, and a main control

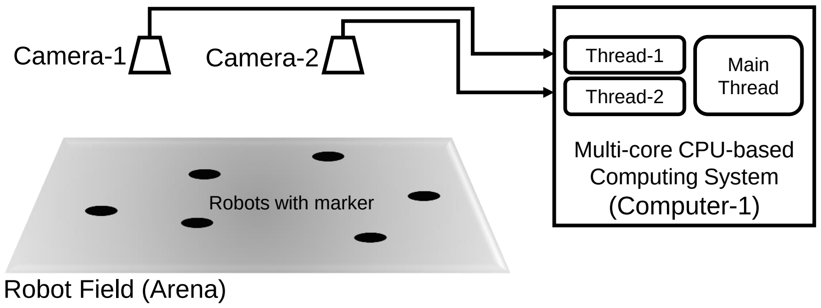


Figure 2.4: Vision-based robot tracking system with multi-thread approach on multi-core CPU.

computer that is accessible from the Internet. The robot localization, orientation, and identification are performed based on robot markers using vision-based multi-robot tracking algorithms, executed on a CPU-based computing system. The robot marker identification supports up to 16 robots. The system configuration uses a FireWire interface camera with a resolution of 1600 × 1200 pixels at 12 fps. An additional CPU-based video server and cameras are used to provide visualization of the real scene and recorded video.

Tanoto et al. introduced Teleworkbench [104; 111], as a scalable and flexible vision-based multi-robot tracking system. The infrastructure was built in the robotic laboratory at Bielefeld University for various mobile robot experiments. It offers a software architecture for a vision-based robot tracking system, which can be adapted to different requirements and is easily extensible for additional functionalities [104]. Teleworkbench provides precise position information, as well as identification, for up to 64 robots and supports a large robot arena using multiple cameras. Compared to SwissTrack and SSL-Vision, it supports a higher resolution of video cameras and a larger number of robots. Figure 2.5 and Figure 2.6 show the Teleworkbench environments with multiple cameras and a multi-server. Regarding the computing hardware, a server equipped with an Intel core i7 940 CPU (quad-core with Hyper-Threading and 2.93 GHz clock speed) is used. Teleworkbench uses one video server for each camera [104] to achieve real-time processing. However, this architecture is considered to be a high-cost solution with a high energy consumption and very complex system maintenance. Thus, in the second generation of Teleworkbench [103], optimizations utilizing a multi-thread approach on a multi-core CPU have been applied. Using this approach, two cameras can be handled by one video server.

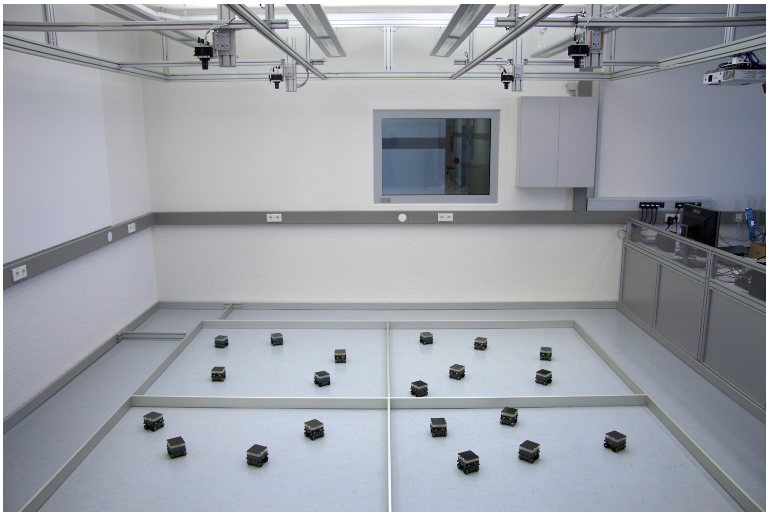


Figure 2.5: Teleworkbench: vision-based multi-robot tracking environment.

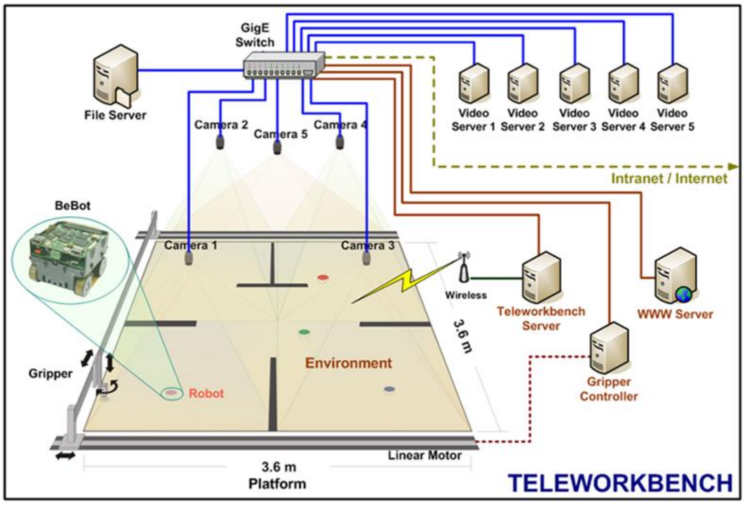


Figure 2.6: Configuration of the Teleworkbench system [103].

2.2.2 FPGA Accelerated Computing System

The initial work on vision-based robot tracking using an FPGA as a hardware accelerator was done by Bianci and Costa in 2002 [17]. They proposed the use of an FPGA in the vision system of a soccer robotic team. A computer vision algorithm that includes thresholding, edge detection, and chain-code segmentation was designed for the FPGA using VHDL. The implementation was simulated for an input image of 320×240 pixels. Little information can be found because it was in the prototyping and simulation phase. The case study only showed the ball detection process.

Rinnerthaler et al. [92] introduced a method called Resource Optimized Co-processing to boost the performance of a DSP-platform for an embedded vision application. The system consists of a DSP coupled to an FPGA. The workload to be processed is distributed between the DSP (TI TMS320C6414) and FPGA and processed in parallel. As a case study, the system was used for robot soccer. The tracking algorithm consisted of Bayer interpolation, background filtering, HSV-based segmentation, color-based classification, and region-based detection to identify the ball and the robot positions. The design achieved a performance of 116 fps for an image resolution of 640×480 pixels using a single camera. However, the concept has not been implemented on a running system.

Ghorbel et.al. [42] proposed a HW/SW implementation on an FPGA for robot localization. They used an Altera FPGA integrating a NIOS-II softcore processor coupled to a hardware accelerator. Most parts of the video processing chain were performed on the NIOS-II processor; only a Sobel filter was implemented in the FPGA fabric. This design required 3.89 s to process one video frame. For the second generation of their design [43], the authors implemented the system on a Xilinx Virtex-5 FPGA, using the embedded PowerPC-440 processor with the Xilinx Floating Point Unit (FPU) coprocessor to process the complete chain, except the Sobel filter. This design required only 30 ms per frame for a video resolution of 640×480 pixels, but supported only the tracking of single robots.

Yu et al. [48; 119; 120] explored the feasibility of using FPGAs in multi-robot formation control applications. Their system used a single digital camera for tracking color markers on moving robots. The monitoring area was $1.2\ m \times 1.6\ m$ in an indoor environment. Each robot was marked with the same marker. A dual color bull's eye marker was used to easily distinguish the robots from the background. To detect the locations of the robots in real-time, a series of image processing algorithms such as image demosaicing, color detection, relative distance estimation, and moving object tracking were implemented in the FPGA. The design achieved a performance of 34 fps using a resolution of 1280×1024 and directly displayed the results on a monitor. Unfortunately, the use of the same color marker on each robot limited the information that could be extracted from the video data, making it difficult to obtain information such as the IDs and directions of the robots.

In 2013, Bailey et al. [9; 34; 35] proposed an FPGA-based smart camera for robot soccer applications. An input resolution of 640 × 480 pixels at 127 fps was used to cover a robot arena of 1.5 $m \times 1.3$ m. Robots were individually labeled with a specific marker, where the ID of the robot was identified based on the marker color and shape (circle, square, and rectangle). Therefore, the number of tracked robots was limited but sufficient for robot soccer applications, as shown in Figure 2.7. A pipelined processing approach was used to obtain the maximum performance on the streamed video data from the camera sensor. Color segmentation and connected component labeling were successfully implemented in the FPGA. However, the final stage, which is to group the blobs into individual robots and calculate their locations and orientations, has not yet been completely implemented. As a system, the design offers significant acceleration for multi-robot tracking because most parts of the video processing chain for detecting the location of the robot are designed to be executed in the FPGA. However, it offers a restricted number of tracked robots, which means that only limited applications can be supported.

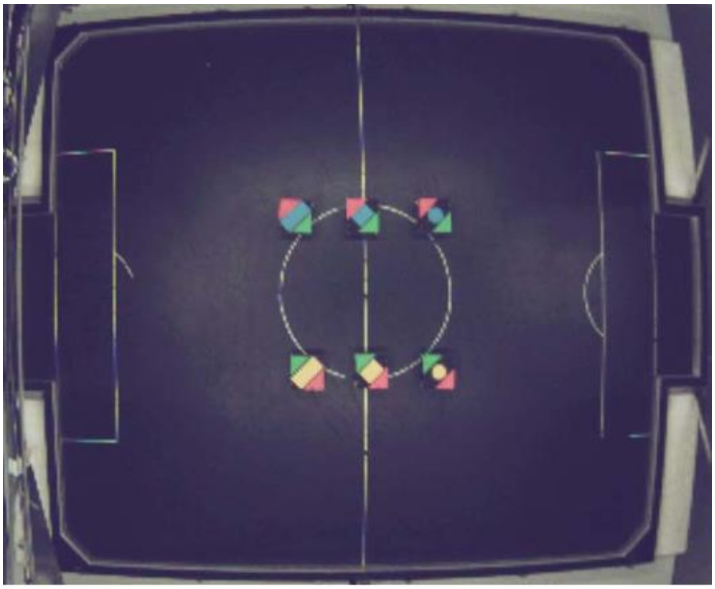


Figure 2.7: Robot marker in FPGA-based smart camera system using shape feature [35].

2.2.3 GPU Accelerated Computing System

The initial work implementing a GPU as a hardware accelerator for vision-based robot tracking application was performed by Zickler et al. [123] in 2009. They conducted an investigation on the potential of using a GPU to improve the computing performance of their SSL-Vision system, a vision-based multi-robot tracking platform for the SSL of RoboCup-Soccer. Using an NVIDIA Geforce GTX 7800 GPU, their GPU-based computing system was 100 times faster than the CPU implementation for a thresholding algorithm. Unfortunately, this approach introduced bottlenecks in the upload and download times between the GPU memory and system memory. Thus, in total, this GPU-based approach was more than four times slower than the implementation of the same thresholding algorithm on the CPU. To solve the bottleneck problem, the authors recommended moving most or all other image processing tasks (in addition to the thresholding) into the GPU. They planned to implement this approach in their future work. However, currently, there is no new documentation on this GPU implementation from the authors. A system for rescue robot competition [45] in 2013 also reported the use of a GPU as a hardware accelerator in its vision box computing system. Unfortunately, there is no further documentation on the detailed implementation and experimental report.

While there has only been a small number of GPU implementations of vision-based multi-robot tracking applications, GPUs have been widely used for accelerating various object tracking algorithms. Some examples are the works presented in [70; 72; 83; 93]. To improve the performance of a six degree-of-freedom pose tracking image processing algorithm, Ruiter et al. [93] utilized a GPU as a hardware accelerator for the blurring and derivative filter. Liu et.al. [72] presented a stereo-vision based framework for tracking the motion of a table-tennis ball in motion-blurred images. GPU-based image processing and a multi-thread technique were used to reduce the latency of the vision system. Limprasert et al. [70] proposed an approach to track people from multiple cameras. They employed a GPU to accelerate the multi-camera tracking process for the overlapping case. A GPU-based system for pedestrian detection using stereo vision on a mobile robot is proposed in [83]. All of the above GPU implementations for object tracking applications show the potential of using the GPU for accelerating vision-based multi-robot tracking applications.

2.2.4 FPGA-GPU Accelerated Computing System

A computing system platform that uses an FPGA, a GPU, and a CPU for a wireless locating system was developed by Alawieh et al. [4]. It is intended for real-time sports analysis applications. Their system uses a radio-based approach rather than a vision-based approach. To detect the locations of the players, each player is equipped with a transmitter device. The FPGA is used for data acquisition, and the GPU is utilized for accelerating the computation of the algorithm to track players.

While there is no implementation of vision-based robot tracking that uses both an FPGA and a GPU, an FPGA-GPU combination hardware accelerator has been implemented for accelerating image processing algorithms in a medical application. Meng et al. [79] proposed an implementation of the FPGA-GPU hardware accelerator for a Cardiac Physiological Optical Mapping application. This implementation shows the potential of using the FPGA-GPU for accelerating the video processing algorithm such as for multi-robot tracking applications.

According to the related work, discussed above, most of the established computing systems for vision-based robot tracking are based on general purpose computers. These systems were focused on developing the software architectures and algorithm implementations on CPU-based computing systems rather than investigating alternative hardware accelerators. The common solution to handle the increasing numbers of cameras and robots is to add extra PCs. Unfortunately, this approach significantly increases the energy consumption and the entire system's complexity. There has been initial work to accelerate the computing performance using an FPGA or a GPU. However, this work is mostly still in the design or prototyping phase, and also use only a single camera, low resolution, and small number of tracked robots. Therefore, an alternative design is proposed in this thesis, using FPGA and GPU implementations for the most computationally intensive tasks of the application, supporting high-resolution video, real-time processing, scalability of the number of cameras, and multi-robot tracking. The Teleworkbench environment, discussed above, was used as the basis for evaluating our implementation. Considering the difference between the FPGA and GPU characteristics, this work focuses on the system architectures, accuracies, computing performances, and power efficiencies of two distinct architectures: FPGA-accelerated and GPU-accelerated computing systems for vision-based robot tracking applications. The FPGA-GPU combination architecture is yet not part of this thesis. It is intended to provide a detailed elaboration of the advantages and disadvantages of the implementations of FPGA- and GPU-accelerated computing systems.

2.3 Hardware Accelerators in Vision Processing

Nowadays, hardware accelerators such as multi-core CPUs, FPGAs, and GPUs have been widely used to support vision processing algorithms, which require highly computationally intensive operations. Each of these hardware accelerators has different advantages compared to the others. In this section, the benefits and drawbacks of multi-core CPU, GPU, and FPGA hardware accelerators are discussed. This information is crucial to propose some alternative hardware accelerators that are able to enhance the computing performance of vision-based multi-robot tracking systems.

2.3.1 Multi-core CPUs

A CPU is a general purpose processor that executes an instruction in a computer program, such as a computational operation, along with input/output operations. The development of the CPU was strongly influenced by the evolution of transistor and integrated circuit (IC) technology. In 1965, Intel co-founder Gordon Moore predicted that the number of transistors on a chip would double approximately every two years. This prediction is the so-called Moore's Law. Figure 2.8 [112] depicts the CPU technology evolution and exponential growth of the number of transistors integrated into a CPU's chip, which follows Moore's Law.

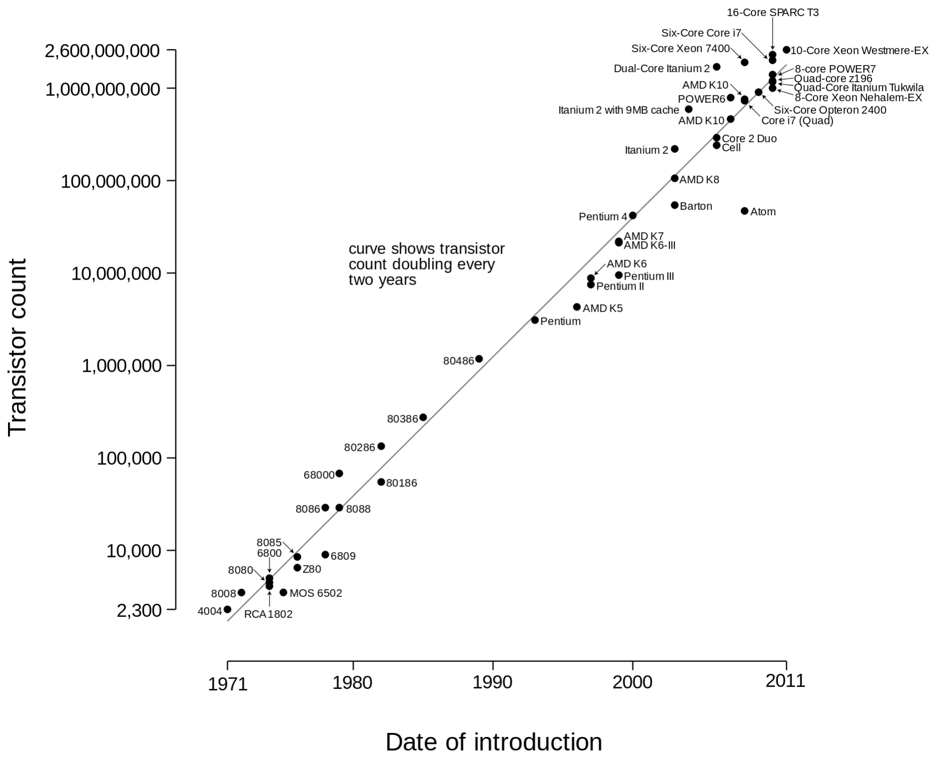


Figure 2.8: Processor transistor counts and Moore's Law [112].

Initially, a CPU had a single core. It could only execute one task at a time. Manufacturers tended to increase the speed at which the processor's clock operated to maximize the CPU's performance. This approach still exists almost three decades after the introduction of the first generation of CPUs. The development of CPU performance is illustrated in Figure 2.9. Until 2003, significant increases were seen in a CPU's clock rate and performance. Then, the limits on the power and available instruction-level parallelism slowed down the performance of a single processor [90]. The single-core CPU that pushed for higher and higher clock speeds reached the point of weakening returns.

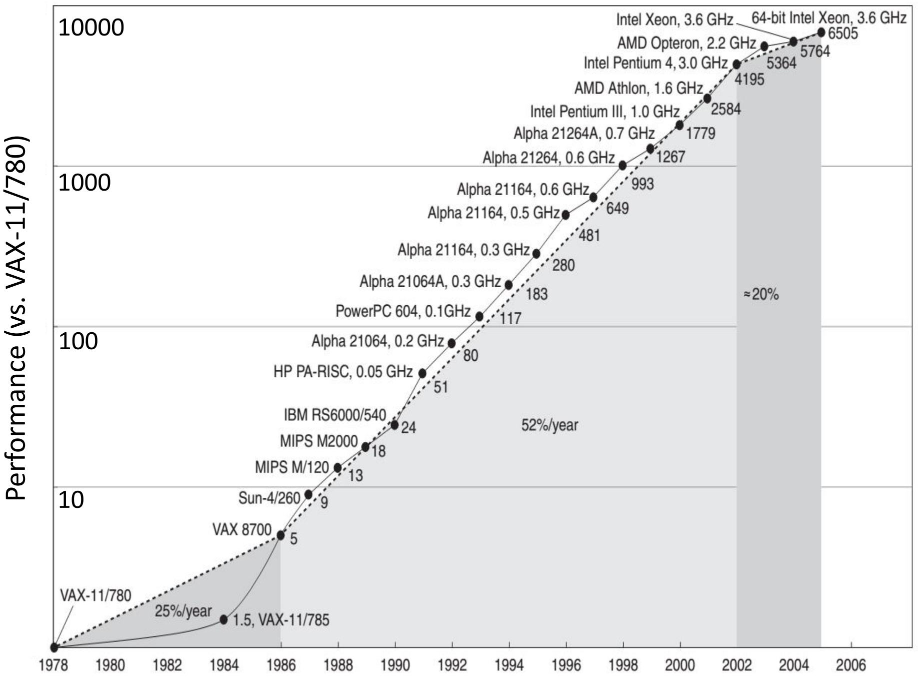


Figure 2.9: Growth in processor performance [50].

Manufacturers discovered an alternative solution to increase the performance by adding more "cores," or central processing units (CPUs) in a single chip. In 2006, Intel introduced the first multi-core CPU. Since then, the evolution of the CPU has been based on multi-core architectures such as dual-cores, quad-cores, and octa-cores. An N-core CPU chip has an N number of physical processor units that function to execute different instructions at the same time. As a result of this parallelism, a multi-core CPU has significantly higher performance than a single-core CPU. An example of a CPU with four processor cores is the Intel i7-4770K CPU, which is manufactured using 22 nm transistor technology. It is the fourth generation of the Intel i7 family of processors with a Haswell architecture. Figure 2.10 shows the top level die layout architecture of the Intel i7-4770K CPU with its four processor cores. Each processor core has 32 KB of L1 cache memory and 256 KB of L2 cache. In addition, another 8 MB of L3 cache is shared across all four cores. As a result, this multi-core architecture delivers independent processing on each processor core and increases the parallelization of the computations.

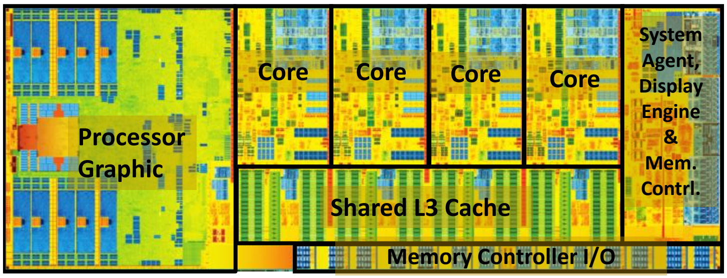


Figure 2.10: Actual die layout of fourth generation Intel i7-4770K CPU with its four processor cores [60].

Additionally, Intel employs hyper-threading technology (Intel HT technology) [77] to maximize the utilization of CPU resources. This technology was initially introduced in 2002 on Xeon server processors and Pentium 4 desktop processors. It is currently implemented on almost all of the new generation of Intel CPUs HT-technology provides more efficiency for processor resource utilization, enabling multiple threads to run on each core. As a performance feature, it increases processor throughput, improving overall performance on threaded software.

Hyper-threading technology allows a single physical processor to appear as two logical (virtual) processors in the operating system; the physical execution resources are shared, and the architecture state is duplicated for the two logical processors. Each logical processor has an architecture state that contains general purpose registers, the control registers, advanced programmable interrupt controller (APIC) registers, and some machine state registers [77]. An illustration of this HT-Technology in the processor is shown in Figure 2.11. It shows that every physical processor core with HT-Technology has two architecture states, while a processor without HT-Technology only has one architecture state. Consequently, according to the software or architecture perspective, operating systems and user programs are able to schedule processes or threads to logical processors as they would on multiple physical processors. From a microarchitecture perspective, it shows that instructions from both logical processors will persist and execute simultaneously on shared execution resources [77].

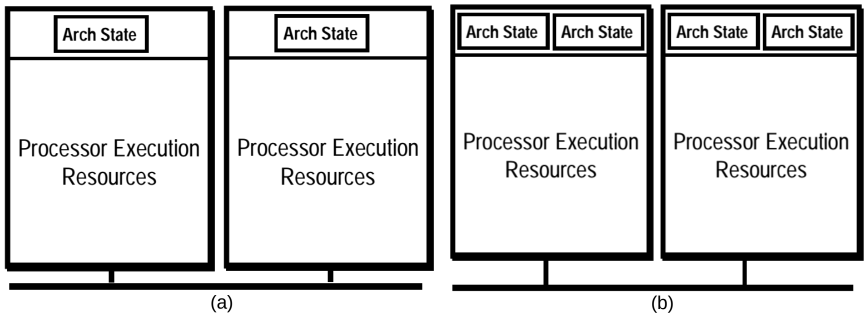

Arch State
Processor Execution Resources
Arch State
Processor Execution Resources
(a)
Arch State
Arch State
Processor Execution Resources
Arch State
Arch State
Processor Execution Resources
(b)

Figure 2.11: (a) Processor without HT-technology and (b) processor with HT-technology [77].

The evolution of the CPU has changed its design from a single to multiple core processor, along with introducing the multi-thread approach in CPU programming. The multi-core architecture and hyper (multi)-thread technology has effectively increased the performance of a CPU by escalating its capability on parallel computations. As a general purpose processor, the benefits of parallel processing are compatible with its advantage in flexibility for application design and implementation. A CPU offers the flexibility to build an ecosystem or a software design architecture, along with convenience in accessing the I/O port and the steadiness of a well-known operating system (OS) such as LINUX or Windows. Figure 2.12 shows a block diagram of the Intel (i7) Haswell platform, one of the existing modern CPU. As can be seen, the processor has a direct connection to the system memory, PCIe interface, and digital display interface, as well as a connection to the platform controller hub (PCH). This PCH provides an interface between the CPU and important ports such as USB 3.0/USB 2.0, SATA 6, High Definition Audio, VGA, integrated LAN, PCIe 2.0, TPM 1.2, and Super IO/EC. All of these interconnections and interfaces enhance the high flexibility of the CPU. As a result, the CPU has the necessary compatibility to work with different operating systems, smoothly implement many software/applications, and conveniently access the I/O ports. Because these advantages cannot all be found in other hardware accelerators, the CPU plays a very significant role in many different applications as a computing platform.

Based on its architecture and platform, a CPU is ideal for complex scalar processing and I/O port access for a sensor or device (e.g., camera, display). It is very suitable for executing complex operations on a single or a few streams of data. For parallel processing, a multi-core CPU processes parallel computations, as many as the number

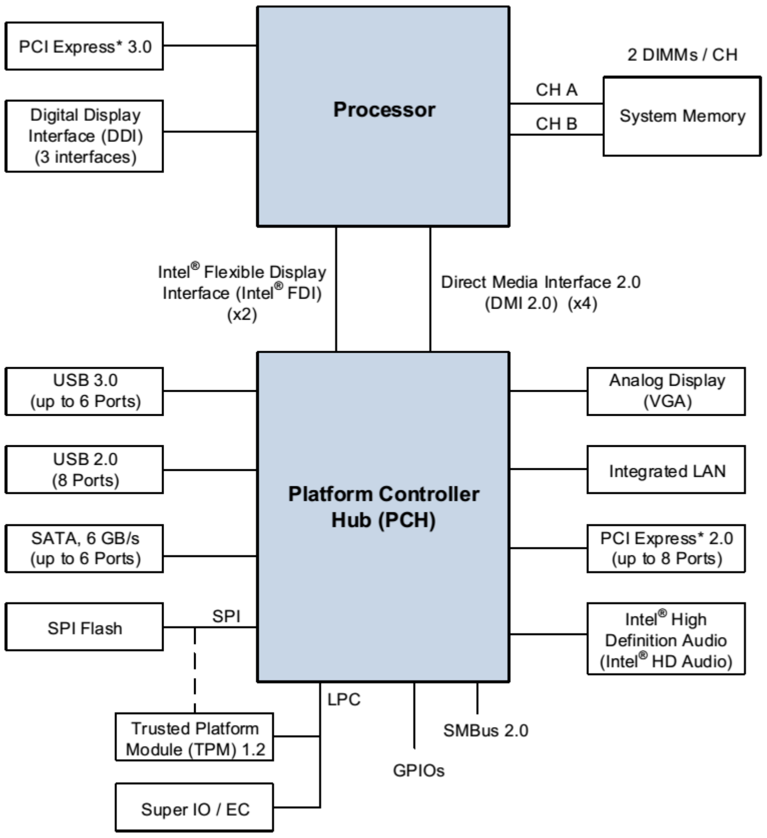

PCI Express* 3.0
Digital Display Interface (DDI) (3 interfaces)
Processor
2 DIMMs / CH
CH A
CH B
System Memory
Intel® Flexible Display Interface (Intel® FDI) (x2)
Direct Media Interface 2.0 (DMI 2.0) (x4)
USB 3.0 (up to 6 Ports)
USB 2.0 (8 Ports)
SATA, 6 GB/s (up to 6 Ports)
SPI Flash
SPI
Platform Controller Hub (PCH)
Analog Display (VGA)
Integrated LAN
PCI Express* 2.0 (up to 8 Ports)
Intel® High Definition Audio (Intel® HD Audio)
LPC
SMBus 2.0
GPIOs
Trusted Platform Module (TPM) 1.2
Super IO / EC

Figure 2.12: Processor (Intel i7) platform block diagram [59].

of cores available. In other words, the capability of a CPU's parallel computation is limited by the number of cores. Because there are some applications that require an excessive number of parallel computations, they cannot typically be implemented using only a CPU. Therefore, the use of an alternative hardware accelerator is taken into account to complement the weakness of a CPU and is combined with a CPU to maximize the computational performance. The great flexibility of the CPU is always needed. It plays an essential role in the complete system of a high computing platform.

2.3.2 Graphic Processing Unit (GPU)

The previous section already described how the performance of a CPU can be enhanced by increasing the number of processor cores in a CPU chip. The enhancement is obtained by scaling up the parallel processing capability, which is nearly equivalent to the growing number of cores in a single CPU chip. Therefore, some researchers believe that the development of future microprocessor industries will continue to focus on adding cores rather than increasing the single-thread performance [30].

Since the early 2000s, the semiconductor industry has generated two primary but different philosophies in microprocessor design [57]. These are the multi-core architecture and many-core architecture. First, the multi-core CPU architecture is a design approach that attempts to optimize the execution speed of sequential programs in every single thread. This method minimizes the latency in the processor by extending its main units such as the on-chip cache unit, control logic unit, and arithmetic-logic unit (ALU). The units' extensions require larger chip areas and higher power consumptions. This means that a CPU core is considered a heavy-weight design. Consequently, the number of cores in a CPU is limited, with a current maximum of 18 cores (Intel E7-8870V3). Unlike the multi-core CPU architecture, the second microprocessor architecture used is a many-core GPU, which merely focuses on improving the throughput of concurrent kernel executions. It utilizes the chip area and power resources to increase the throughput performance. As shown in Figure 2.13, a GPU uses fewer resources (transistor) than a CPU for the on-chip cache, control logic, and arithmetic logic units. Additionally, a GPU dedicates more transistors to data processing (in ALU) rather than data caching and flow control (in on-chip cache and control logic units). As a result, its architecture has a large number of processing cores.

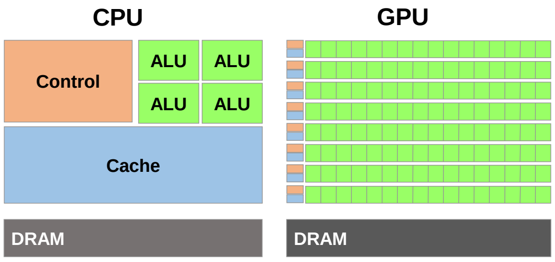


Figure 2.13: CPU vs GPU architectures [85].

A GPU core is a lightweight design that is dedicated to data-parallel tasks. Therefore, an individual thread in a GPU likely needs a much longer execution time than a CPU. However, by employing its many cores, a GPU can process thousands of threads simultaneously. In other words, a multi-core CPU uses the advantage of its heavy-weight cores to process some computation tasks, while a GPU effectively handles tasks using its hundreds or thousands of lightweight cores [102].

Indeed, the presences of GPU computing is not intended to replace the complete function of CPU computing. The two approaches have their own advantages and both are useful for different types of applications or tasks. The GPU technology is very suitable for a program with a huge number of threads or data-parallel computation-intensive tasks. Meanwhile, a CPU with its much lower latencies can achieve higher performance for a program that has few threads but requires control-intensive tasks. In other words, a complementary GPU and CPU combination potentially generates significant improvements in many applications.

To explore the potentialities and characteristics of the GPU in more detail, including its benefits in parallel computing, the next subsection describes two aspects of GPU computing technologies. These include the GPU architecture as the hardware aspect and parallel programming as the software aspect.

2.3.2.1 GPU Architecture Overview

This work emphasizes NVIDIA GPUs with their Compute Unified Device Architecture (CUDA) programming platform. Figure 2.14 illustrates the top level block diagram of a modern CUDA-supported GPU architecture, which consists of a cache memory, memory controller, host interface, GigaThread Engine, and numerous streaming multiprocessors. The cache memory refers to an on-chip memory that is allocated from among the streaming multiprocessors. A memory controller is a unit to access an external memory (global memory). The host interface has functions for communication and transferring data to the host PC, whereas the GigaThread Engine schedules thread blocks to various streaming multiprocessors.

The streaming multiprocessor (SM) is the most important part of the GPU. A single GPU consists of numerous SMs (e.g 12, 15). As shown in Figure 2.14, the GPU architecture replicates the SM architecture building block. This approach aims to obtain high parallel computing capability since all of the SMs can run simultaneously. Each SM in a state of the art GPU comprises up to hundreds of computing cores (CUDA cores), as illustrated in Figure 2.15. Considering that each SM is able to support the concurrent execution of hundreds of threads, one GPU can concurrently execute thousands of threads [30].

Some examples of well-known GPU architectures that consist of SMs are the Fermi and Kepler architectures. In this thesis, the GTX-580 and GTX 780 NVIDIA GPUs are used

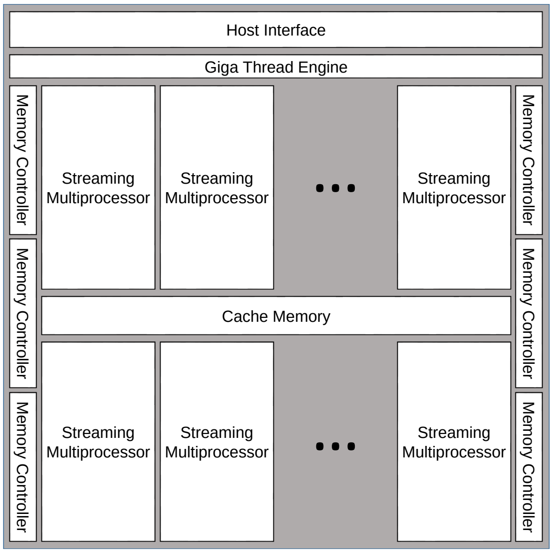


Figure 2.14: Top-level block diagram of modern GPU, modified from [30].

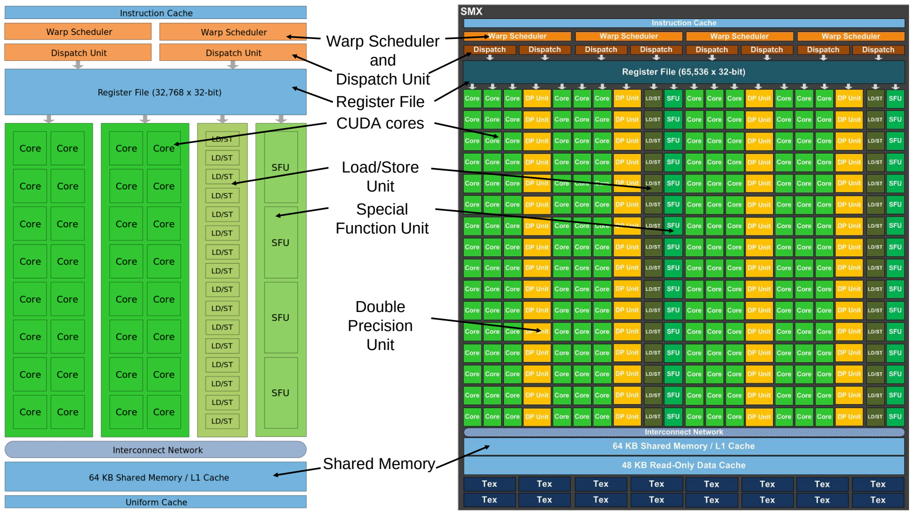


Figure 2.15: Streaming multiprocessor: Fermi (left) and Kepler (right) architectures, modified from [86; 87].

to represent the Fermi and Kepler architectures, respectively. The SM architectures of both Fermi and Kepler are shown in 2.15 and their complete architecture are illustrated in Figure 2.16 and Figure 2.17, respectively. The GTX-580 Fermi based GPU [86] is fabricated using approximately 3.0 billion transistors and features 512 CUDA cores. Meanwhile, the GTX 780 uses around 7.1 billion transistors and features 2304 CUDA cores. Furthermore, the GTX-580 organizes its 512 CUDA cores in 16 SMs of 32 cores each; whereas the GTX 780 arranges its 2304 CUDA cores in 12 SMs (from the maximum of 15 for the Kepler architecture) of 192 cores. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and and floating point unit (FPU) [86]. Thus, it is able to execute a floating point or integer instruction per clock for a thread.

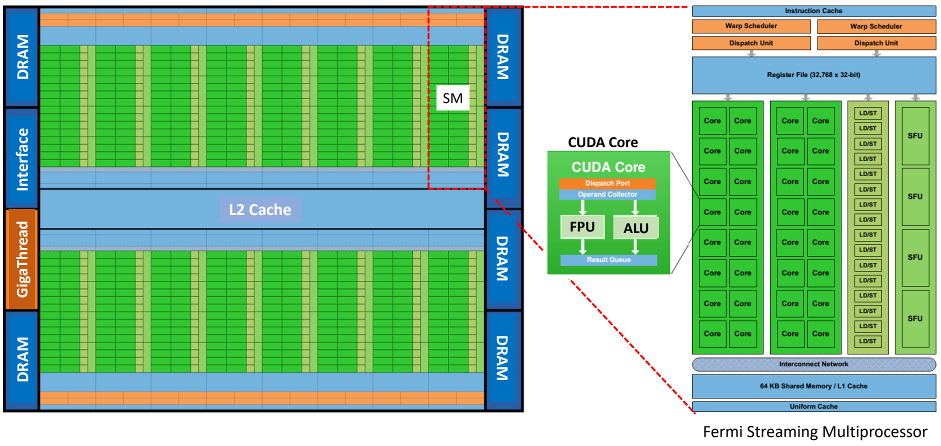


Figure 2.16: NVIDIA GTX580 Fermi architecture [86].

In a CUDA-supported GPU, an SIMT approach plays a role in handling and executing many threads. All of the threads are processed in a group by group fashion. All of the threads in the same group execute the same instruction simultaneously. The warp scheduler and dispatch unit in the SM determine the threads in groups of 32 parallel threads called warps. In the GTX 580, each SM has two warp schedulers and two instruction dispatch units, which allow two warps to be issued and executed concurrently. The GTX 780 supports a higher number of warp schedulers, where each SM features four warp schedulers and eight instruction dispatch units. It generates four warps to be issued and executed concurrently. Unlike the GTX 580 Fermi architecture, which does not permit double precision instructions to be paired with other instructions, the GTX 780 Kepler architecture allows double precision instructions to be paired with other instructions [87]. Both the GTX 580 and 780 GPUs have six 64-bit memory

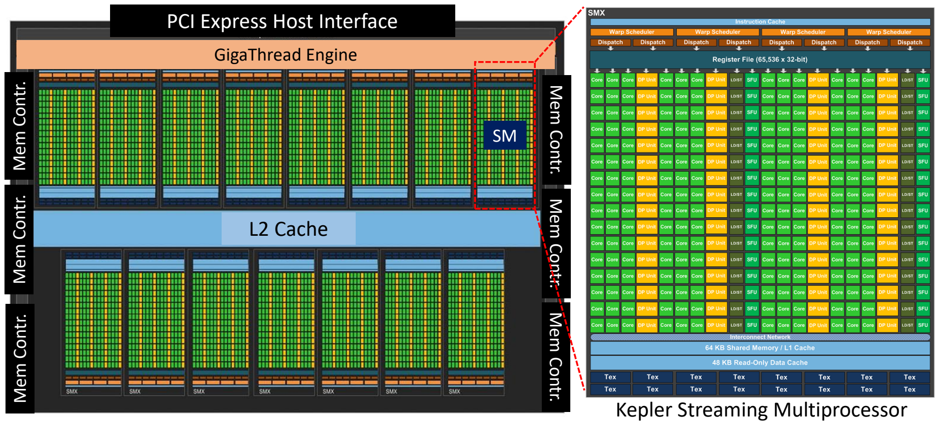


Figure 2.17: NVIDIA GTX780 Kepler architecture [87].

partitions, for a 384-bit memory interface, approximately supporting a total maximum of 6 GB of GDDR5 DRAM memory. However, the GPU for this work only uses 1.5 GByte for the GTX-580 and 3 GByte for GTX 780.

Figure 2.18 shows a block diagram of the CUDA device memory model, including its association with the threads and SM. The block diagram uses only two thread blocks, which are located in a distinct SM to represent the relationship between the threads and SM in a simple way. In the CUDA-supported GPU, all of the threads located in the same block are executed in one SM. Therefore, these threads can be synchronized and utilize the same shared memory. In contrast, the threads in different blocks are executed in separate SMs. They operate independently and use a distinct shared memory. This condition prevents the different threads in blocks from cooperating with each other.

Based on the thread's accessibility to the data in memory (as illustrated in Figure 2.18), there are three memory groups in a GPU device:

- The thread level refers to data stored in the memory that is accessible only by the thread that writes them. In this category, there are registers with low latency (fast accessing time) and local memory with high latency (slow accessing time).

- The SM level is data stored in the memory that is only accessible by the threads that are located in the same block and executed in the same SM. The shared memory unit located in the SM is also accessible.

- The device level refers to data stored in the memory that are accessible by all of the threads in a kernel. Additionally, the data at the device level are also

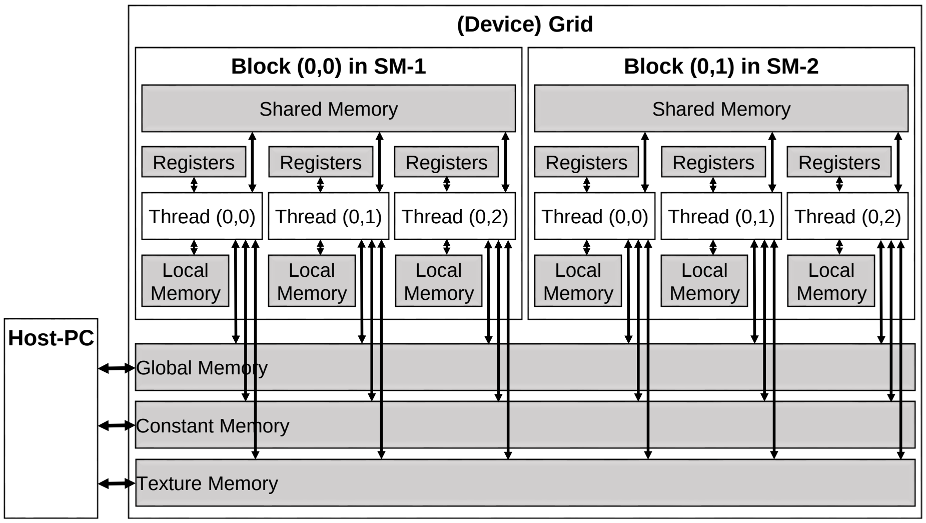


Figure 2.18: Block diagram of CUDA device memory model, modified from [30].

accessible by the host PC. This category includes the global memory, constant memory, and texture memory.

The global memory is an external DRAM with a high latency access time. However, it has a huge amount of storage and is accessible by all of the SMs in the GPU. Meanwhile, constant and texture memory are beneficial for very specific types of applications such as for data having fixed values during a kernel execution. In contrast to the global memory, the shared memory is a programmable on-chip memory with very low latency and high bandwidth. It exists on every SM with a limited memory. It is shared and partitioned among the thread blocks in a specific SM. A shared memory is not accessible between different SMs. In some tasks or programs, it functions as a data buffer of the global memory, reducing the data transfer latency between the CUDA core and global memory.

2.3.2.2 CUDA software on GPU

CUDA is a general purpose parallel computing platform and programming model invented by NVIDIA. It drives the parallel computing engine in NVIDIA GPUs, to support various computationally intensive applications on GPU-accelerated computing systems. Many algorithms and applications that can be formulated as data-parallel computations perform well in CUDA-supported GPUs [30]. Using CUDA, a programmer is able to implement parallel computing in a more efficient approach.

A CUDA program consists of a combination of two different parts that are executed on either a CPU (host PC) or GPU. It makes it possible to execute programs or applications on heterogeneous computing systems. In CUDA programming, as illustrated in Figure 2.19, the parts that comprise few or no data-parallel (host code) operations are executed in the host PC (CPU), whereas the parts that hold a huge number of data-parallel (kernel GPU code) operations are performed in the GPU device [65].

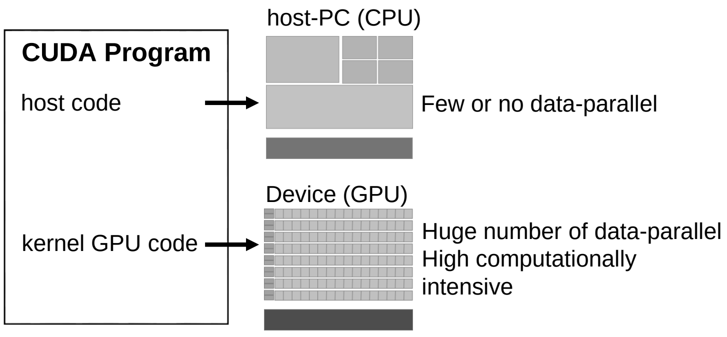


Figure 2.19: CUDA program structure.

A kernel to be executed in the GPU device holds a large number of threads to process the data using an efficient concurrent approach. A programmer can write a sequential program for a single thread, whereas the CUDA platform will manage the scheduling for all the GPU threads. Figure 2.20 illustrates the organization of the many threads in a CUDA-supported GPU. It applies two-level block and grid hierarchies. All of the threads generated by a kernel are arranged into a grid. They are organized into blocks of threads, and all of the blocks are organized into a grid. Based on this hierarchal organization, CUDA provides a unique identity for each thread. CUDA uses block index coordinates within a grid (blockIdx) and thread index coordinates within a block (threadIdx) to identify all of the threads. Based on the identities (coordinates) of all the threads, a programmer can define portions of data to different threads.

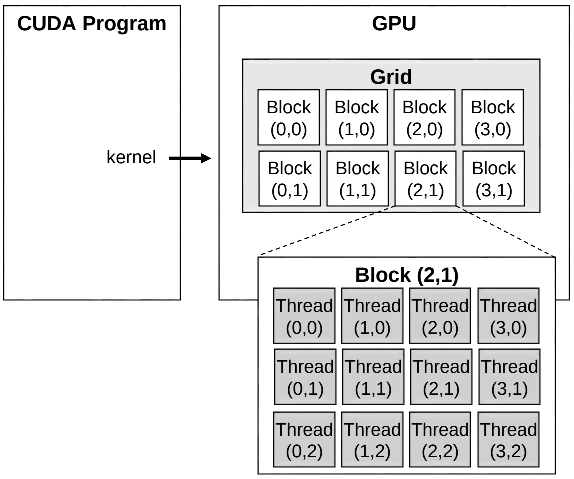


Figure 2.20: Thread hierarchy in CUDA-supported GPU, modified from [30; 85].

After understanding how all the threads are organized, it is important to know how they are processed from a hardware perspective. Figure 2.21 illustrates the corresponding hardware component for each hierarchy from the logical perspective of the software (kernel). Every single thread is executed sequentially in a CUDA core, whereas all the threads in the same block (thread block) are executed simultaneously in the same single SM. A kernel in the CUDA-supported GPU is performed using the SIMT execution model. When this process is started, the kernel generates all the threads and organizes them into a thread block grid. Afterward, the GigaThread engine (Figure 2.14) schedules and distributes the grid of thread blocks to the SMs.

As described in the previous section, all of the threads in a group are processed in a group fashion. The warp scheduler and dispatch unit in an SM the threads into groups of 32 parallel threads called warps. The number of active warps is restricted by the SM's resources such as the registers and shared memory. These resources are shared among warps and blocks. Therefore, not all of the warps are active. The ratio of active warps to the total number of available warps is called the occupancy. A higher warp occupancy means a better utilization of GPU computation resources [28].

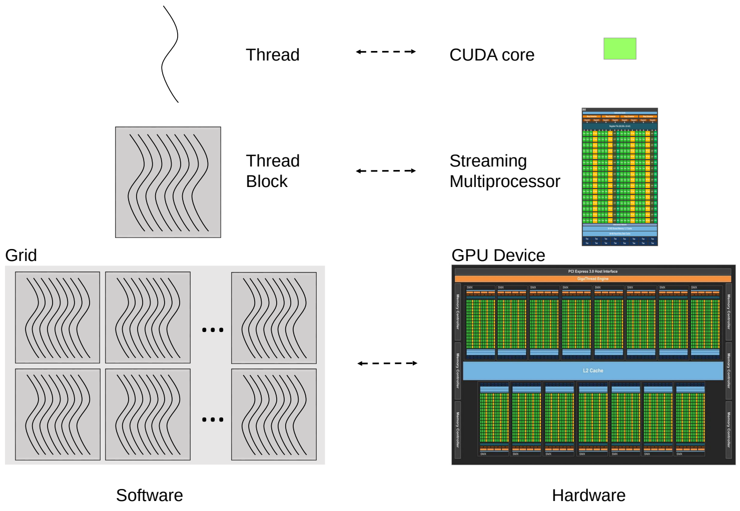


Figure 2.21: Illustration of logical view corresponding to hardware view, modified from [30].

2.3.3 Field Programmable Gate Arrays (FPGAs)

In this section, the basic architectural features of FPGAs are explored to understand their architectural benefits. An FPGA is a type of prefabricated integrated circuit that can be re-programmed for different digital circuit or system functions. Some modern FPGA devices consist of up to two million logic cells that can be configured to implement a variety of software algorithms [115]. When an FPGA is configured, the internal circuitry is connected in a way that creates a hardware implementation of the software application. In a general purpose processor, an algorithm is executed as a sequence of instructions by utilizing its fixed architecture. In other words, with a processor, the computation architecture is fixed, and the best performance is obtained by following the available processing structures. In this case, the performance is a function of how well the algorithm maps to the capabilities of the processor [115]. Unlike general purpose processors, FPGAs use dedicated/customized hardware for processing algorithms and do not have an operating system [1; 23]. An algorithm in an FPGA is implemented by building separate hardware for each function using the FPGA's logic cells and components. This approach, which is inherently supported by the FPGA's architecture, allows a hardware design to have a parallel speed performance while retaining the reprogrammable flexibility of software at a relatively low cost [8].

The basic architecture and components of a generic FPGA are shown in Figure 2.22. It consists of an array of configurable logic blocks, programmable interconnects, and input/output (I/O) blocks. Logic blocks are used to implement the logic of a custom algorithm or function. Each of these uses a look-up-table (LUT) to perform some logic operations and flip-flops to store the result of the LUT. The logic blocks are typically arranged in a two-dimensional matrix array and connected by configurable interconnects. During the FPGA configuration process, this programmable interconnect wire is used to enable the interconnections between the logic blocks. As an interface between the FPGA and external devices, I/O blocks can be configured as input/output ports. To increase the computational density and efficiency of the device, modern FPGA architectures incorporate the above-mentioned basic components along with additional computational and data storage blocks [115] such as DSP48 and Dual-Port RAM, as shown in Figure 2.23. The combination of these components provides more flexibility in the FPGA design, making it possible to implement any software algorithm that typically runs on a processor. More details about these components will be discussed in the following paragraphs.

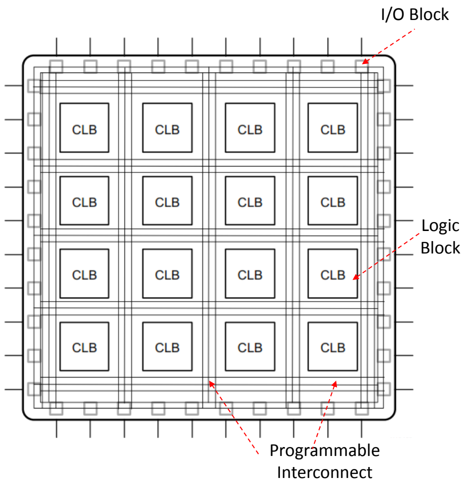


Figure 2.22: Basic FPGA architecture [115].

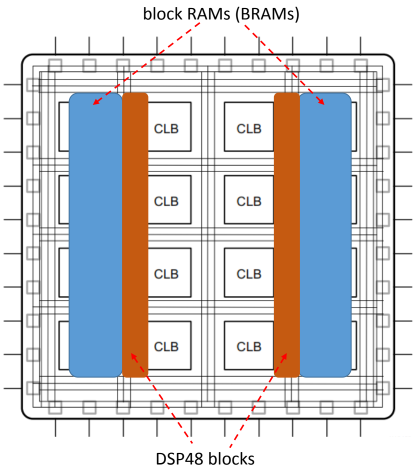


Figure 2.23: Contemporary FPGA architecture.

Each logic block in the FPGA is divided into several logic slices, and each logic slice consists of numerous logic cells, which are the smallest logic unit within the FPGA device. Different FPGA technologies usually have a distinct number of logic slices and logic cells. The basic element inside the logic cell is illustrated in Figure 2.24. As the smallest logic unit, each logic cell typically consists of a LUT and flip-flop. Basically, a LUT is a truth table where different combinations of inputs implement different functions to produce output values. A flip-flop is a basic storage unit for storing the LUT output. The hardware implementation of a LUT can be represented as a collection of memory cells connected to a set of multiplexers, as shown in Figure 2.24-a [115], where the LUT inputs are used as selector bits on the multiplexer to choose the result at a given point in time. Therefore, a LUT can be used as both a function of a computation engine and a data storage element. A LUT and flip-flop combination within a logic cell is illustrated in Figure 2.24-c.

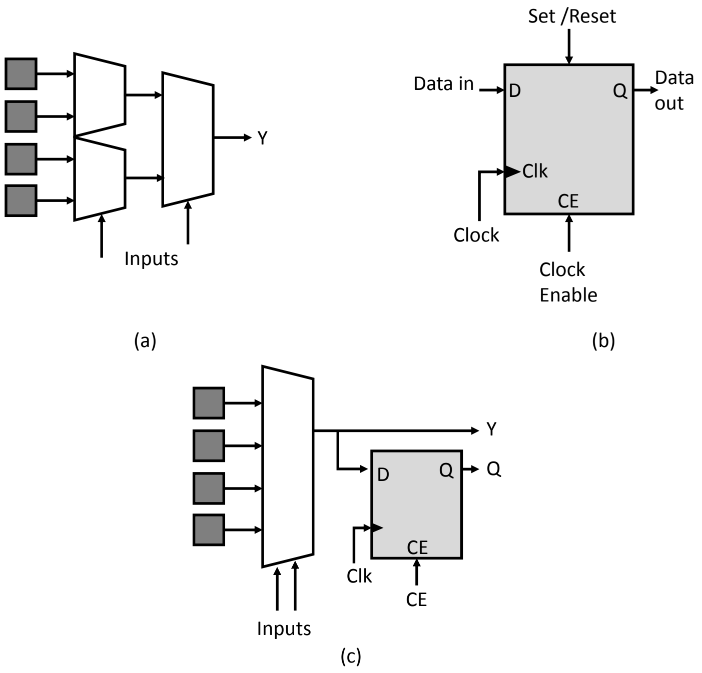


Figure 2.24: Basic elements in logic block of FPGA [115]: (a) functional representation of LUT as collection of memory cells, (b) structure of flip-flop, and (c) structure of logic cell in Xilinx FPGA.

To efficiently support digital signal processing (DSP) applications, which typically use many binary multipliers and accumulators, FPGAs are equipped with a DSP48 block, as shown in Figure 2.25. The DSP48 block is an ALU embedded into the fabric of the FPGA. One DSP48 block could contain two or more slices. Each DSP48 slice supports many independent functions, including a multiplier, multiplier-accumulator (MACC), multiplier followed by an adder, three-input adder, barrel shifter, wide bus multiplexer, magnitude comparator, and wide counter. The architecture also supports connecting multiple DSP48 slices to form wide math functions, DSP filters, and complex arithmetic without the use of a general FPGA fabric [118].

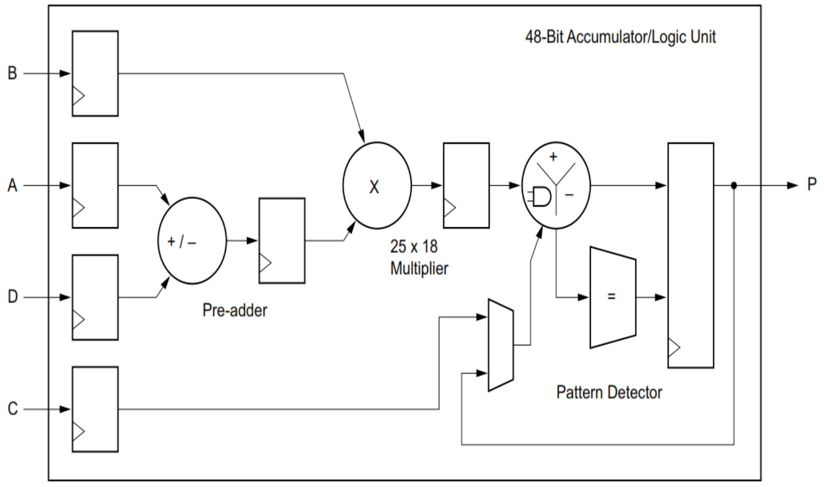


Figure 2.25: Structure of a DSP48 block [115].

A BRAM in an FPGA device refers to a dedicated dual-port RAM module, which functions as an embedded memory element. It is used to provide on-chip storage for a relatively large set of data. Each FPGA device usually possesses two types of BRAM memories, which can hold either 18 k or 36 k bits. Indeed, these memory numbers are device specific. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations [115]. In the Xilinx FPGA, five memory types can be generated from these block RAMs. These are single-port ROM, single-port RAM, dual-port ROM, simple dual-port RAM, and true dual-port RAM. The single-port ROM and single-port RAM have only one port to access the memory space. As illustrated in Figure 2.26-a and b, the ROM type only provides read access, while the RAM type uses the same port for both read and write accesses. The dual-port ROM (Figure 2.26-

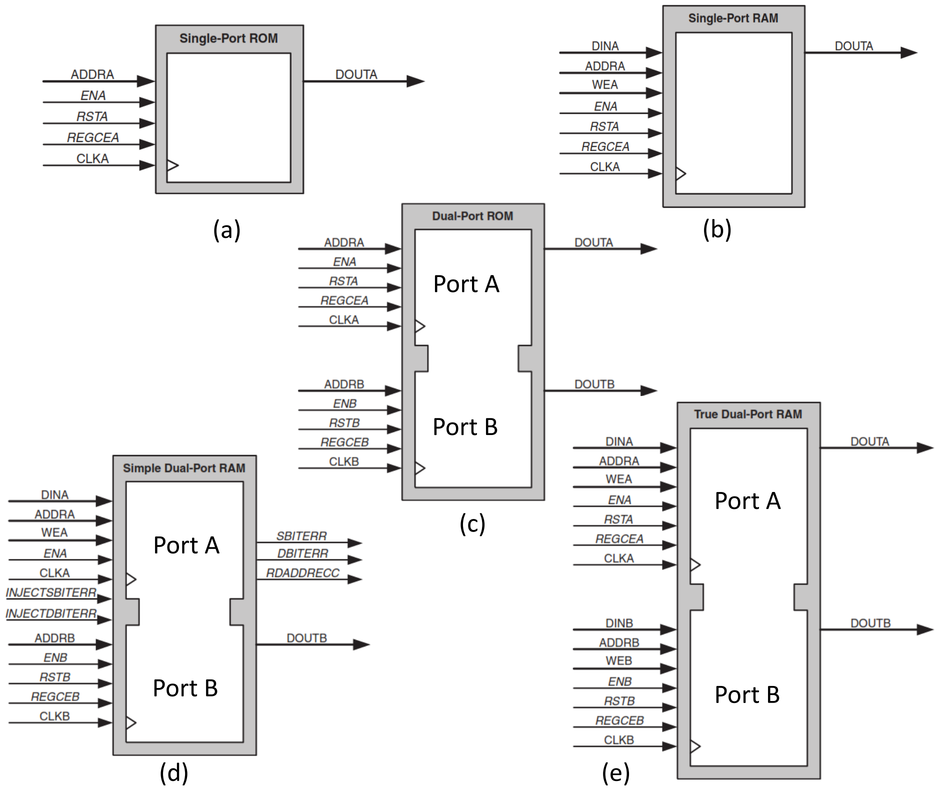


Figure 2.26: Five memory types [114] generated from block RAMs: (a) single-port ROM, (b) single-port RAM, (c) dual-port ROM, (d) simple dual-port RAM, and (e) true dual-port RAM.

c) allows read access to the memory space through two ports. Meanwhile, for the simple dual-port RAM, as illustrated in Figure 2.26-d, the write access to the memory is allowed through port A, and read access is allowed through port B. Lastly, as shown in Figure 2.26-e, the true dual-port RAM allows read and write accesses to the memory on either port A or B [114].

2.4 Summary

This chapter has described the basic concept of vision-based robot tracking systems and the related work in detail. In particular, it showed that most of the established computing systems for vision-based robot tracking focus on developing the software architecture and algorithm implementation on a general purpose processor (CPU) rather than investigating alternative hardware accelerators. Therefore, this chapter has shown the need for a computing system that uses the benefits of the CPU and hardware accelerators (e.g., FPGA and GPU) to enhance the computing performance of a vision-based multi-robot tracking algorithm. Additionally, this chapter has discussed the architectures of the multi-core CPU, GPU, and FPGA to give a complete overview of their costs and benefits. The qualitative comparison between them is summarized in Table 2.1. To obtain an appropriate comparison, similar technology processes were taken into account as important parameters during the comparison.

Table 2.1: Qualitative comparison between CPU, GPU and FPGA, based on [15; 27; 106].

	CPU	**GPU**	**FPGA**
Parallelism	Limited by the number of cores	Supported by SIMT (single instruction multiple threads) approach	High parallelism with a customized design approach
Power efficiency (performance / watt)	Low	High	Very high
Interfaces	Support different interfaces	Limited or dependent on the interface with CPU	Customizable, including direct connection to cameras
Development time	Short	Medium	Long

Table 2.1 shows that each device technology has different advantages and disadvantages in term of its parallelism, power efficiency, interfaces, and development time. The technology selection depends on the architectural design considerations and application requirements [15]. Therefore, the next chapter shows how heterogeneous computing

systems are used in vision-based multi-robot tracking applications. In heterogeneous computing systems, different types of processors or hardware accelerators cooperate to accelerate the computation tasks. The discussions in chapter 3 and the rest of this thesis will focus on the implementations of two distinct heterogeneous computing systems for vision-based multi-robot tracking applications, encompassing the use of the FPGA and GPU as hardware accelerators for a CPU.

3 Vision-based Multi-Robot Tracking with Heterogeneous Computing Systems

As presented in chapter 1, the development of a vision-based multi-robot tracking system using multiple cameras as the video input source began with the objective to deal with a larger robot arena and support various applications. Additionally, the computational intensity of vision processing algorithms increases with the number of tracked robots, video frame size, and number of used cameras. This chapter delineates FPGA-CPU and GPU-CPU heterogeneous (hybrid) computing systems as alternative approaches for vision-based multi-robot tracking applications. It includes a comprehensive elaboration on heterogeneous computing systems, both the FPGA-CPU and GPU-CPU architectures, and vision-based multi-robot tracking algorithms.

3.1 Heterogeneous Computing System

The mainstream computing system for a high-performance computer platform has been rapidly evolving to combine more than one type of processor or hardware accelerator over the last decade, changing from homogeneous into heterogeneous systems. This development means a dynamic breakthrough in the field of high-performance computing systems. For instance, conventional homogeneous computing uses only one or more processors with the same architecture; heterogeneous computing alternately combines the benefits from the different types of processors or hardware accelerators to enhance the computation tasks.

Heterogeneous computing systems have been widely used in many highly computationally intensive applications and successfully provide significant improvements compared to conventional computing systems. At this point, the FPGA and GPU are the most prevalent hardware accelerators for heterogeneous computing systems. An FPGA offers customized design features and a parallel structure, while a GPU provides massively parallel processing using its thousands of cores. As a result, there have been various implementations of FPGA-CPU heterogeneous computing systems, such as for big data applications [29; 108; 121], neural networks [69; 94], and image processing applications. These image processing applications include an avionic test application [2], a face detection algorithm [81], an optical flow algorithm [26], and medical image processing for ultrasound computer tomography [19; 20].

GPU-CPU heterogeneous computing systems are more adaptable and can be implemented in various applications. This is different from FPGA-based systems. One of the reason is because the development time in a GPU is relatively much faster than in an FPGA. Accordingly, many applications have been implemented on GPU-CPU heterogeneous computing systems, including data processing [13; 71], numerical method [3; 82; 109], chemistry [16; 49; 76], bioinformatics [25], electromagnetics [38], physics [51; 56], and image processing applications. These image processing applications include biomedical imaging [68; 73; 96; 110], face detection [47; 88], and optical flow algorithm [89].

The FPGA and GPU hardware accelerators possess their own unique features and advantages. To some extent, experts and researchers have been encouraged to investigate these factors, particularly to identify their benefits and drawbacks. Consequently, some prominent studies on various applications and algorithms have been conducted by experts and researchers, including a heterogeneous computing platform [27; 113], medical imaging [12; 18; 21], and pedestrians detection [22].

From the perspective of heterogeneous computing systems, the CPU and hardware accelerator (e.g., FPGA or GPU) work together to improve the computing performance. As illustrated in Figure 3.1, the hardware accelerator enhances the computing performance of the system by executing massive parallel processing or computationally intensive tasks. Meanwhile, the CPU has an essential task within the application, especially in executing complicated operations on a single or a few streams of data. A CPU also easily accommodates integration between the computing system and operating system (OS) and provides I/O port access to a sensor or device (e.g., camera, display).

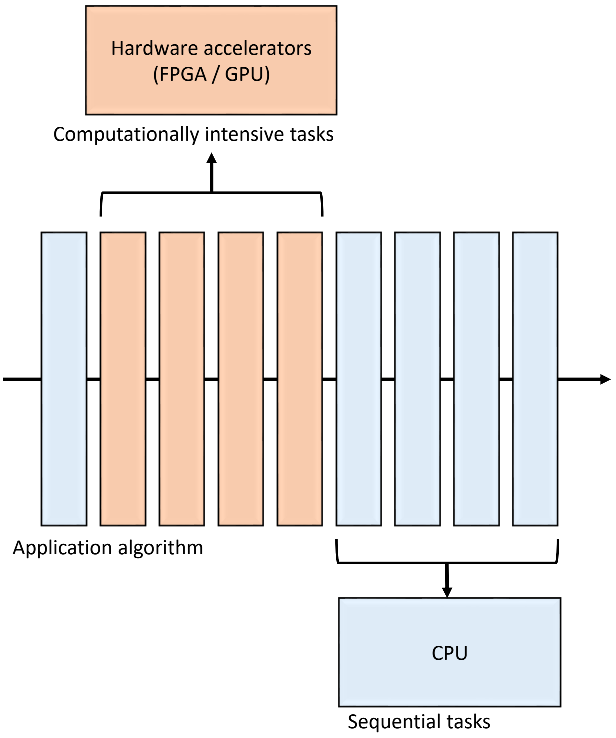

Hardware accelerators (FPGA / GPU)
Computationally intensive tasks
Application algorithm
CPU
Sequential tasks

Figure 3.1: Tasks partitioned in heterogeneous computing system.

At the architectural level, there are two types of architectures, both of which are based on the integration between the CPU and hardware accelerator [80]. They are defined as discrete and integrated heterogeneous systems. The former (discrete) consists of a multi-core CPU and hardware accelerator, both connected to a high-speed bus, and each of them has a distinct memory, as shown in Figure 3.2. In contrast, as illustrated in Figure 3.3, the latter integrates both a CPU and hardware accelerator in a single chip, and it shares the same memory between the CPU and hardware accelerator. This integrated heterogeneous system is also known as a programmable system-on-a-chip (SoC). A low data transfer overhead is the main advantage of an integrated heterogeneous computing system [52; 100]. However, this system normally uses a simpler CPU and hardware accelerator (GPU) architectural design. Hence, it generates a lower performance on computationally intensive tasks than a discrete system.

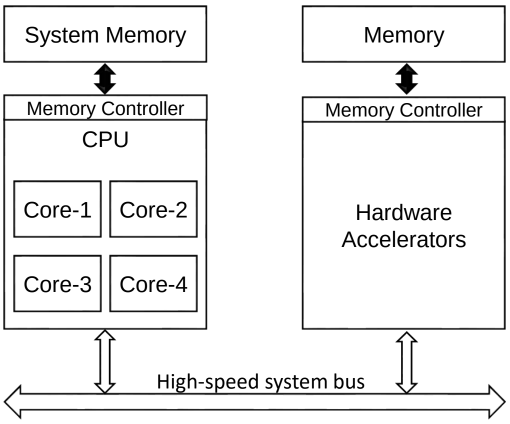


Figure 3.2: Discrete heterogeneous computing system architecture.

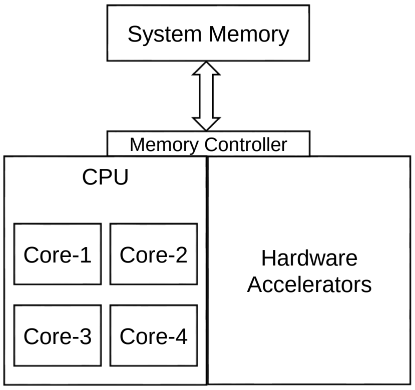


Figure 3.3: Integrated heterogeneous computing system or programmable SoC architecture.

In discrete heterogeneous computing systems, designers can independently combine certain types of CPUs and GPUs to enhance the computing performance. Therefore, this work uses a discrete architecture for the implementation of vision-based multi-robot tracking heterogeneous computing systems. The following section within this chapter discusses the proposed system in more detail.

3.2 Architecture and Design Flow

As mentioned in the previous chapters, the realization of a vision-based multi- robot tracking system imposes three challenges: fast processing for real-time robot tracking using high-resolution images, scalability of the number of cameras to support larger environments, and a system performance capable of simultaneously processing many robots for multi-robot applications. Indeed, most of the established vision-based multi-robot tracking systems are implemented on CPU-based computing systems. Therefore, this chapter explores heterogeneous (hybrid) computing systems, particularly FPGA-CPU and GPU-CPU based computing systems, which aim to increase the computation performance. These will likely become alternative approaches in the field of vision-based multi-robot tracking systems. Basically, the systems combine the advantage of using a CPU as the processor in the host PC with the use of an FPGA or a GPU as a hardware accelerator. The proposed systems have the objective of efficiently handling computationally intensive vision-based multi-robot tracking algorithms.

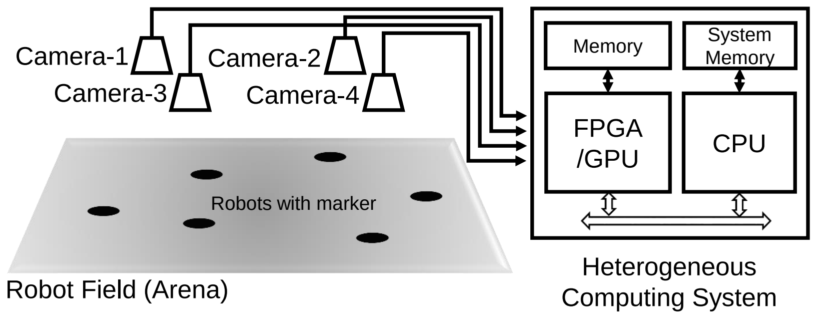


Figure 3.4: Heterogeneous computing system for vision-based multi-robot tracking.

Figure 3.4 depicts the top-level block diagram of the proposed heterogeneous computing system for vision-based multi-robot tracking applications. It uses four cameras to cover a robot arena with a size of 6 $m \times 6$ m and a discrete computing system that consists of the FPGA/GPU and CPU. The FPGA/GPU are utilized as hardware accelerators for processing the computationally intensive tasks of the algorithm to detect the locations of the robots. Meanwhile, the CPU is used as the processor in the host PC for post-processing algorithms and display.

3.2.1 FPGA-CPU Heterogeneous Computing System

This section presents the design flow and architecture of the FPGA-CPU computing system for vision-based multi-robot tracking. Figure 3.5 illustrates the design flow for the implementation of an FPGA-based heterogeneous computing system, which is a modified version of the FPGA-based design process from Bailey [8]. The implementation processes in the CPU and complete (heterogeneous) system are added to the design process. Hence, the design flow consists of five main steps: problem specification, image processing algorithm development, architecture selection, algorithm implementation in the FPGA and CPU, and implementation on a complete (heterogeneous) system.

The problem specification includes at least three aspects [8; 36] that must be considered: the system functionality, which means the expected function or output of the system; the system performance that must be achieved (e.g., frame rate, the number of robots); and the system environment, including the number of cameras, robot arena, etc. In the next step, the problem specification is used to develop the algorithm.

The objective of image processing algorithm development is to find a series of operations that transform the input image into the expected output. In this work, the output is some relevant information related to the tracked robots; for instance, their locations, orientations, and IDs. In the FPGA design, the image processing algorithm cannot be developed directly on the targeted FPGA device. This is because the development cycles (e.g., synthesize, translate, map, place, and route) require too much time, which influences the algorithm implementation. Therefore, it becomes impracticable to have an interactive design. Thus, a software environment such as MATLAB or OpenCV could be used to simulate and develop the algorithm. This approach allows the algorithm to be tested up to the application level and meets the relevant accuracy and robustness performance [8].

When an initial algorithm has been developed, the implementation architecture can be defined, which includes system level and computational level architectures. Referring to the system level architecture, this work uses a discrete heterogeneous computing system, as shown in Figure 3.6. Indeed, the main goal of the computational architecture design is to improve the computation performance, particularly to exploit the parallelism in the FPGA structure and accelerate the execution of the algorithm. According to the context of the heterogeneous computing system, the algorithm is divided and distributed into the FPGA and CPU. Usually, an algorithm requires some modification before it can be efficiently mapped into the FPGA as a hardware accelerator. However, not all algorithms are well transformed into a hardware implementation. Algorithms with complex data-dependent control and possibly frequent memory accesses are merely suitable to be implemented in software [33]. Section 3.4 provides more details on

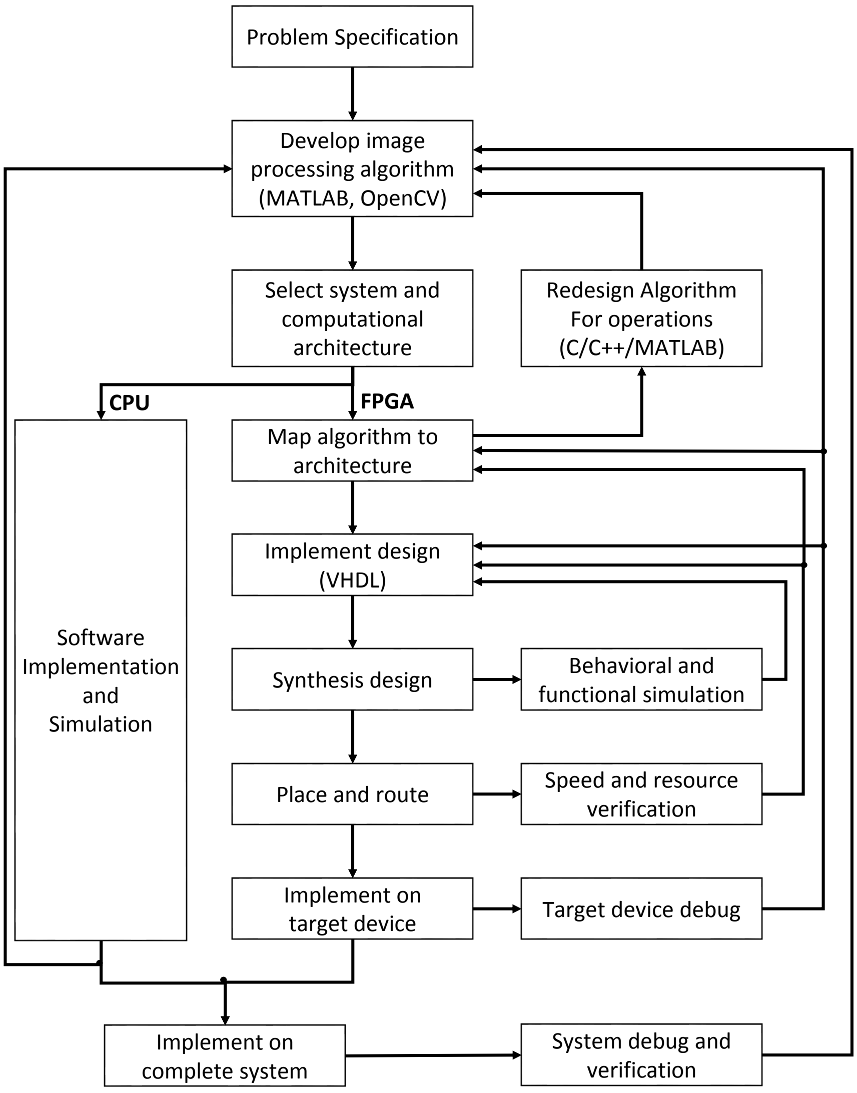

Problem Specification
Develop image processing algorithm (MATLAB, OpenCV)
Select system and computational architecture
Redesign Algorithm For operations (C/C++/MATLAB)
CPU
FPGA
Software Implementation and Simulation
Map algorithm to architecture
Implement design (VHDL)
Synthesis design
Behavioral and functional simulation
Place and route
Speed and resource verification
Implement on target device
Target device debug
Implement on complete system
System debug and verification

Figure 3.5: Design flow for FPGA-CPU implementation. Modified version from [8].

partitioning the vision-based multi-robot tracking algorithm into the hardware and software.

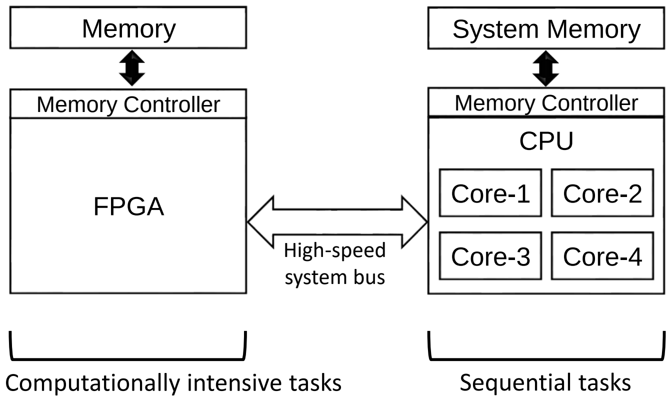


Figure 3.6: Discrete FPGA-CPU heterogeneous computing system architecture.

Once the architecture and algorithm distribution are defined, the next process for hardware implementation in the FPGA and software implementation in the CPU can be started. The FPGA- and CPU-based algorithm implementations can be implemented independently. However, the rules for developing the image processing algorithm, computational architecture design, and implementation in hardware and software are not always straightforward. It is possible to redesign and iterate the process to obtain the desired performance. Additionally, the image processing algorithm implementations in the CPU and FPGA differ significantly. The software implementation in the CPU is mostly coding the algorithm, and it can be simulated interactively. Meanwhile, the FPGA-based implementation requires the design of specific hardware to execute specific operations of the algorithm. Therefore, the hardware design and simulation in the FPGA is mostly performed in a step-by-step way based on a sequence of operations in the algorithm. High performance and a resource efficient design are typically desired, exploiting the benefits of the FPGA parallelism. Finally, after both the CPU and FPGA implementations are completed and integrated into one system, the system is ready to be tested for debugging and verification.

3.2.2 GPU-CPU Heterogeneous Computing System

The design flow for the implementation of a GPU-based heterogeneous computing system is shown in Figure 3.7. It consists of five main steps: problem specification, image processing algorithm development, architecture selection, algorithm implementation in the GPU and CPU, and implementation on a complete (heterogeneous) system. It basically uses the same approach as the FPGA-based system for determining the problem specification and developing the image processing algorithm. The design flow involves a rich vision processing algorithm such as MATLAB or OpenCV library [105] for developing the application algorithm. The use of these kinds of development tools can accelerate the algorithm development process.

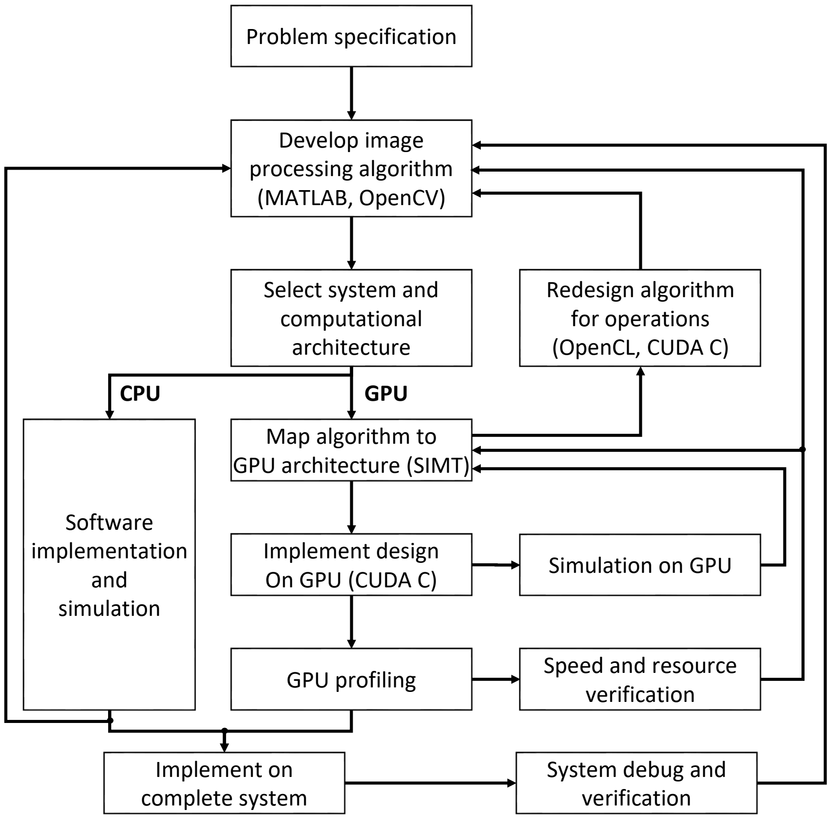

Problem specification
Develop image processing algorithm (MATLAB, OpenCV)
Select system and computational architecture
Redesign algorithm for operations (OpenCL, CUDA C)
CPU
GPU
Map algorithm to GPU architecture (SIMT)
Software implementation and simulation
Implement design On GPU (CUDA C)
Simulation on GPU
GPU profiling
Speed and resource verification
Implement on complete system
System debug and verification

Figure 3.7: Design flow for GPU-CPU implementation.

Figure 3.8 shows the use of a discrete GPU-CPU heterogeneous computing system. At the computational design level, the massively parallel architecture of the GPU, which consists of many cores, is employed to accelerate the execution of the algorithm. The algorithm is partitioned and distributed into the GPU and CPU. This means that the parts that comprise few or no data-parallelism are executed in the host PC (CPU), whereas the parts that take a huge number of data-parallelism are performed in the GPU device [65]. When the architecture and algorithm distribution are defined, the algorithm implementation in the GPU and CPU can be started.

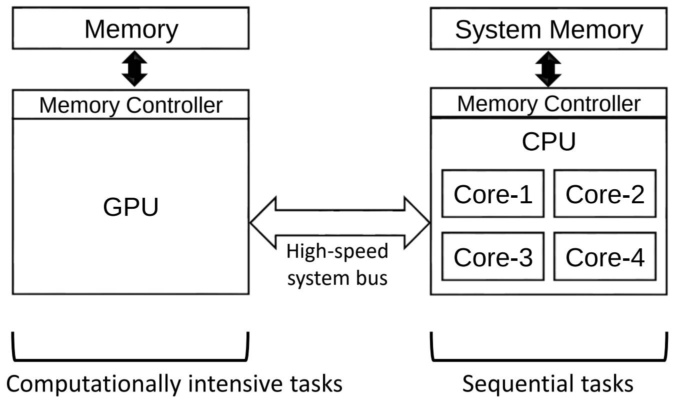


Figure 3.8: Discrete GPU-CPU heterogeneous computing system architecture.

The algorithm implementation in the GPU is different from its implementation in the CPU and FPGA. In dealing with GPU programming, thinking in parallel and acknowledging the basic understanding of the GPU architecture are needed, in order to obtain an optimum computing performance [30]. As described in section 2.3.2, a GPU architecture has hundreds or even thousands of built-in cores. Furthermore, an SIMT method is used to execute a kernel (algorithm) by exploiting its many cores. Figure 3.9 shows that in the CUDA-supported GPU, when a kernel is launched, a grid consisting of numerous threads blocks is scheduled to implement an algorithm. Every single thread executes instructions as the CUDA platform manages the scheduling for all the GPU threads to be executed in a concurrent processing manner.

In contrast to the FPGA, where it is likely unfeasible to perform an interactive simulation, debugging and interactive simulation are fully accommodated in a GPU development system. A GPU development tool such as the NVIDIA CUDA tool provides a profiling tool to examine the speed performance, power consumption, and achieved occupancy of the GPU. Therefore, the algorithm implementation in the GPU is relatively faster and has more benefits than its implementation in the FPGA. Finally, after both the GPU and CPU implementations are completed and integrated as a system, the heterogeneous computing system is ready to be tested for debugging and verification.

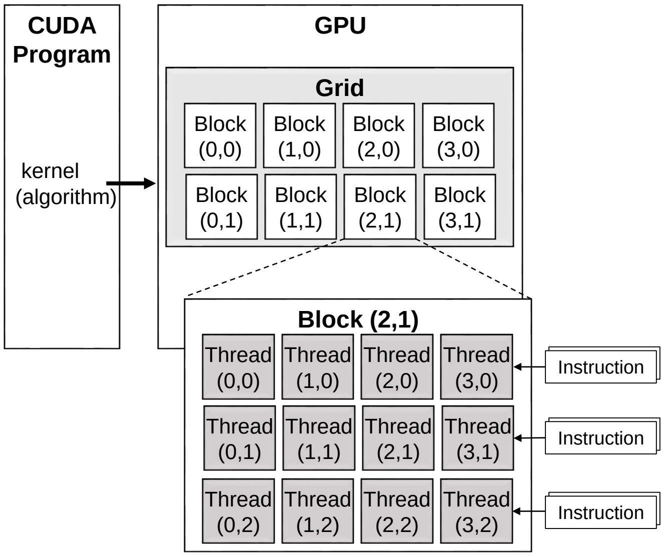


Figure 3.9: CUDA-supported GPU algorithm implementation.

3.3 Vision-based Multi-Robot Tracking Algorithm

The purpose of the computing system in a vision-based multi-robot tracking application is to execute algorithms for extracting useful information (e.g., locations, orientations, and IDs of the robots) from the video. There are at least three main steps for vision algorithms, as illustrated in Figure 3.10. These consist of segmentation, robot detection, and post processing. Segmentation is applied to distinguish objects from the background image. It directly affects the robot detection step, which is able to classify or identify each object, either as a robot or not. The output data from robot detection is no longer image based, but the locations of the robots. Finally, some post-processing algorithms such as for computing the orientations and identities (IDs) of the robots are implemented, so that the system can obtain more accurate and comprehensive information.

In a heterogeneous computing system, it is necessary to select an appropriate algorithm to obtain the benefits of the hardware accelerator (FPGA or GPU). An algorithm that can be effectively mapped into a stream, and parallel data processing approaches are well-matched for FPGA- and GPU-based hardware accelerators. Stream processing is a good fit for pixel-level image processing operations such as point operations and

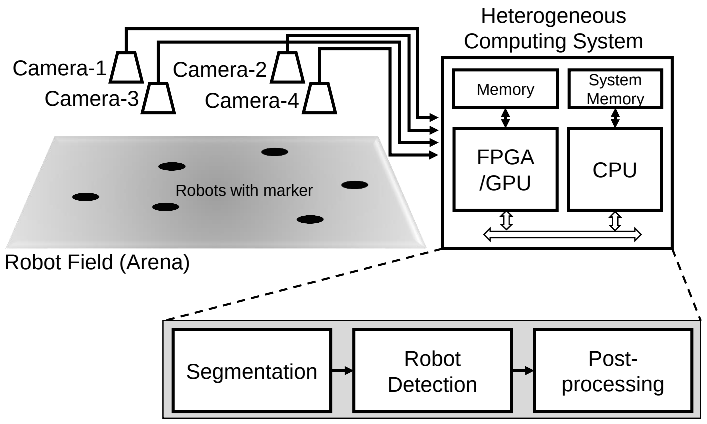


Figure 3.10: Heterogeneous computing system with its algorithm.

local filtering, where processing is executed during a raster scan through the video frame input [8]. Additionally, point operations and local filtering can also be effectively performed in the GPU by employing its many cores and an SIMT approach.

To identify each robot in multi-robot tracking applications, they are individually labeled with a specific marker to be captured and recognized by the system using its camera. The robot marker used in this application is designed to have a high recognition rate. This can be achieved using a specific shape with predefined patterns for the identification of the individual robots. This work adopts the robot marker used in Teleworkbench [103]. Figure 3.11 depicts the marker that is used in our multi-robot tracking application. Each marker consists of a circle, pentagon, and barcode. The circle is used to detect the location of the robot. It is chosen because a circle shape can be effectively recognized in a stream processing fashion, which is performed as a whole circle detection process in one pass during the raster scan of the video frame. This stream processing method is essential for effectively using the FPGA or GPU architecture as a hardware accelerator. Furthermore, the circle detection method is also compatible with detecting two or more colliding circles, which is necessary for multi-robot tracking applications. The pentagon defines the robot's direction, while the bar code represents the robot's identity (ID). The bar code comprises six cells (colored black or white) arranged in three columns and two rows, enabling the identification of up to 64 robots. Additionally, the use of a circle with color in the robot marker makes the maximum number of IDs scalable. If needed, the maximum number can be doubled using two distinct colors (e.g., red and blue) for the circle in the robot marker.

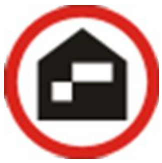


Figure 3.11: Robot marker used in this multi-robot tracking application.

Figure 3.12 shows a detailed block diagram of the vision-based multi-robot tracking algorithm and how its tasks are partitioned between the hardware accelerator and CPU. The hardware accelerators (FPGA/GPU) are used to execute computationally intensive tasks with huge data rates, while the CPU is utilized to perform complex and control operations with low data rates. Therefore, this work divides the algorithm implementations as follows:

- The segmentation and robot detection algorithms that are performed on a video frame of up to 2048 × 2048 pixels (approximately 120 MByte/s for 30 fps) using the hardware accelerator (FPGA/GPU).
- The post-processing algorithm, which is executed on a small image, requires frequent access to the memory and complex control operations, is implemented in the CPU of the host PC.

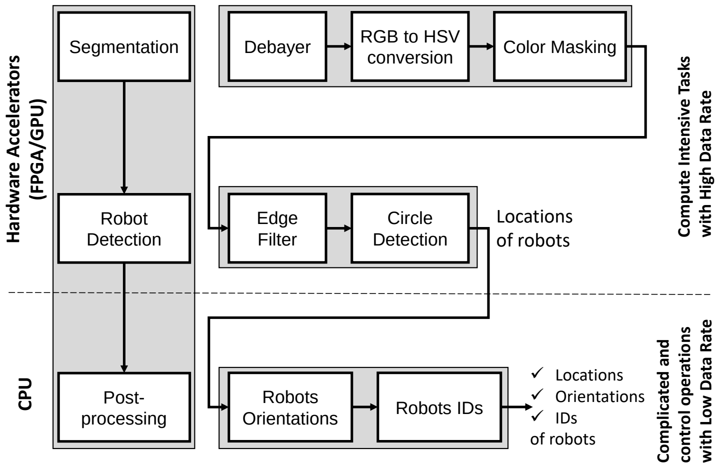


Figure 3.12: Details of vision-based multi-robot tracking algorithm and its task partitions in hardware accelerators and CPU.

The object segmentation algorithm includes the debayer, RGB to HSV color conversion, and color mask operations to extract the circles from an image with a resolution of 2048 × 2048 pixels. The operations are based on the marker's color. These three operations are similar to point operations in image processing. Thus, they are well mapped to an FPGA or a GPU hardware accelerator. For robot detection, the algorithm involves edge filter and circle detection operations. Edge filtering is a gradient-based method that relies on a convolution operation for processing the filter kernel and input image. It can be effectively mapped into the FPGA and GPU architectures. Afterward, the circle detection algorithm receives the edge filtered image to identify the locations of the robots and sends the results to the post-processing step.

In the post processing phase, this work applies the same algorithm as in Teleworkbench [103] to calculate the orientations and IDs of the robots because the same pentagon and barcode are used. Teleworkbench is a vision-based multi-robot tracking system infrastructure that was developed by Congnitronics and the Sensoric Research Group, CITEC, Bielefeld University. The post-processing algorithm is performed based on the cropped images, 40 × 40 pixels each. To obtain these images, cropping operations are conducted on the input image, where the region of interest (ROI) is determined by the coordinates or locations from the previous image processing phase (segmentation and robot detection). The computations for the robot's orientation and ID are implemented based on a 40 × 40 pixels image for each robot marker, utilizing some functions in the OpenCV library (e.g., findContours, minAreaRect, getRotationMatrix2D). In other words, the post-processing algorithm performs the computation operations on a relatively small image, but it also requires frequent access to the memory and complex control operations. As a result, the post-processing algorithm is more suitable for implementation in the CPU rather than in the FPGA/GPU.

3.3.1 Segmentation

In the segmentation phase, the red circle within the robot marker is extracted using a color segmentation algorithm. The goal is to acquire an image with the required information to be used for detecting the robots' locations. The segmentation algorithm includes a Debayer, RGB to HSV color conversion, and color mask operations, as shown in Figure 3.13.

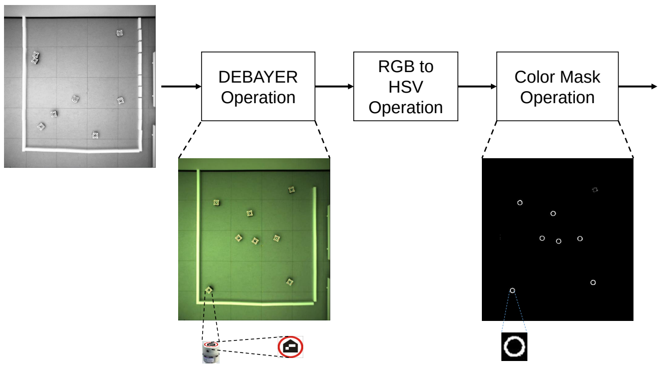


Figure 3.13: Top level block diagram of segmentation module.

3.3.1.1 Debayer

The captured video frame from the camera is raw image data in a Bayer pattern format. As illustrated in Figuree 3.14, each pixel in the Bayer pattern image has only a single color, where one quarter of the pixels are red, half are green, and another quarter are blue. This means that each pixel has missing color components. Therefore, a Debayer operation is needed to create a full RGB color image out of a Bayer-encoded image. To retrieve the complete RGB values for each pixel, this work uses a bilinear interpolation method.

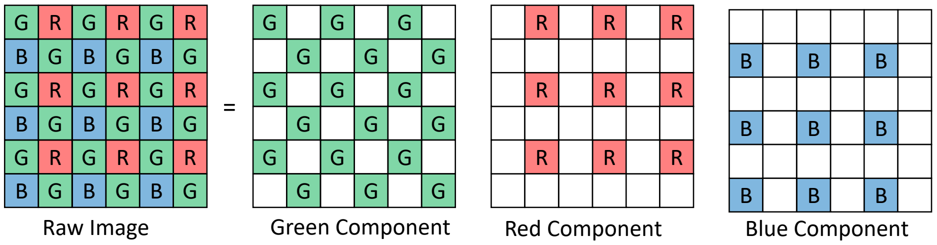


Figure 3.14: Bayer pattern and its color components.

The bilinear interpolation is an eight neighborhood filter. It uses the values of the eight neighbors of a Bayer pattern pixel to estimate the missing color components. Furthermore, the values of the missing color components are determined by calculating the average of the adjacent pixels. Figure 3.15 illustrates the Bayer pattern pixel array, which is used to easily explain the interpolation process. For pixel $G1_{33}$ (green-1, pixel number 33) as the center, the red (R_{33}) component and blue (B_{33}) component need to be estimated. Their interpolation values are calculated based on Equation 3.1. For pixel R_{34} (red, pixel number 34) as the center, the green (G_{34}) component and blue (B_{34}) component are estimated using Equation 3.2, while for pixel B_{43} (blue, pixel number 43) as the center, the green (G_{43}) component and red (R_{43}) component are calculated using Equation 3.3.

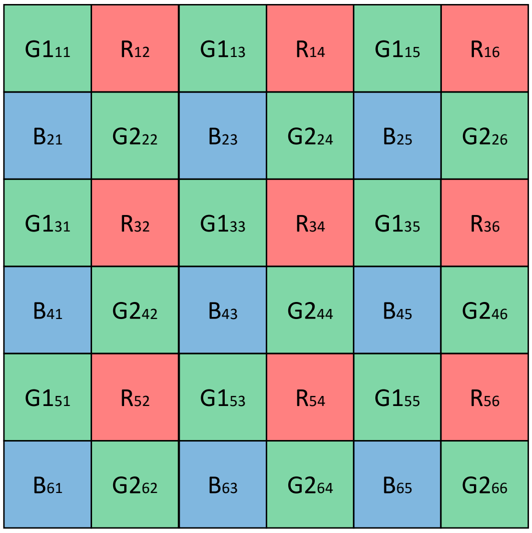


Figure 3.15: Bayer pattern pixel array.

$$R_{33} = \frac{(R_{32} + R_{34})}{2} \quad and \quad B_{33} = \frac{(B_{23} + B_{43})}{2} \tag{3.1}$$

$$G_{34} = \frac{(G1_{33} + G2_{24} + G1_{35} + G2_{44})}{4} \quad and \quad B_{34} = \frac{(B_{23} + B_{25} + B_{43} + B_{45})}{4} \tag{3.2}$$

$$G_{43} = \frac{(G2_{42} + G1_{33} + G2_{44} + G1_{53})}{4} \quad and \quad R_{43} = \frac{(R_{32} + R_{34} + R_{52} + R_{54})}{4} \tag{3.3}$$

3.3.1.2 RGB to HSV color conversion

The hue saturation value (HSV) color space is a widely used color space in image processing for color detection and enhancement. It offers intensity independence for the hue and saturation, which enables more robust segmentation [8]. Previous works [6; 78] proved that an HSV color space provides a more robust performance than the RGB color space with respect to changes in the illumination and lighting. Therefore, an RGB to HSV color conversion operation is integrated into the sequence of the segmentation algorithm. In this work, the RGB to HSV operation is implemented based on the Foley et al. [41] algorithm. Mathematically, the conversion from RGB to HSV is performed as follows:

$$H = \begin{cases} 0 & , \text{if } R = G = B \\ 60 \times \frac{(G-B)}{\Delta} & , \text{if } R = max(R,G,B) \\ 120 + 60 \times \frac{(B-R)}{\Delta} & , \text{if } G = max(R,G,B) \\ 240 + 60 \times \frac{(R-G)}{\Delta} & , \text{if } B = max(R,G,B) \end{cases} \tag{3.4}$$

where $\Delta = max(R,G,B) - min(R,G,B)$ and if $H < 0$, then $H = H + 360$

$$S = \begin{cases} \frac{\Delta}{max(R,G,B)} & , \text{if } max(R,G,B) \neq 0 \\ 0 & , \text{if } max(R,G,B) = 0 \end{cases} \tag{3.5}$$

$$V = max(R,G,B) \tag{3.6}$$

3.3.1.3 Color Mask

A color mask operation is utilized to conceal all the colors except a specific color range on the HSV colored image. In this case, it will be the HSV equivalent values for the red color in the robot marker. The output of this operation is a binary image (as shown in Figure 3.13), where the pixels are set to white (active pixels) if their HSV values fall within the specified threshold parameters in all three channels. Otherwise, the pixels are set to black. The color mask operation is defined as follows:

$$ColorMask = \begin{cases} 255 & \text{, if } H_{mask} = S_{mask} = Vmask = 1 \\ 0 & \text{, otherwise} \end{cases} \tag{3.7}$$

where H_{mask}, S_{mask}, and V_{mask} are obtained using the following formulas:

$$H_{mask} = \begin{cases} 1 & \text{, if } H_{Low} \leqslant H \leqslant H_{High} \\ 0 & \text{, otherwise} \end{cases}$$

$$S_{mask} = \begin{cases} 1 & \text{, if } S_{Low} \leqslant S \leqslant S_{High} \\ 0 & \text{, otherwise} \end{cases}$$

$$V_{mask} = \begin{cases} 1 & \text{, if } V_{Low} \leqslant V \leqslant V_{High} \\ 0 & \text{, otherwise} \end{cases}$$

3.3.2 Robot Detection

The robot localization algorithm includes edge filtering and circle detection, as shown in Figure 3.16. The edge filter operation is used as a preprocessing operation to deliver a decent input image for the circle detection operation. As described in the previous subsection, the output of the segmentation module is a binary image that is obtained from the color segmentation algorithm. In this case, it is the red color that is used in the circle of the robot marker. If there is any additional object (apart from the robot marker) with the same color that is used for the robot marker in the current frame, this object will be not filtered out by the color mask. This could lead to false positive detections of circle candidates. If the size of a homogeneously colored object is greater than or equal to the robot's marker, then this object will be detected as a possible circle. Therefore, an edge detection filter is applied as a preprocessing step to remove large colored areas that do not represent markers.

This design utilizes a Sobel filter for the edge detection filter. Sobel edge detection is a gradient-based method that uses two 3 × 3 kernels which are convolved with the

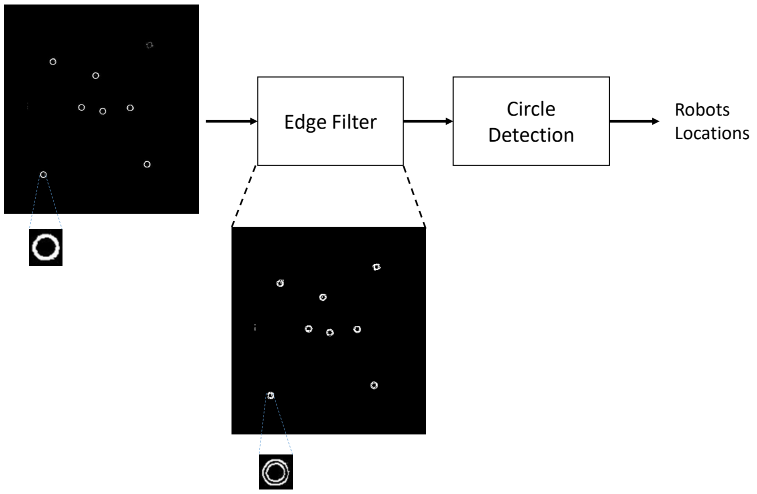


Figure 3.16: Top level block diagram of robot detection algorithm.

input image to compute the approximations of the horizontal and vertical gradients, as shown in Equation 3.8. For the gradient computation, the pixels within the image (I) are multiplied by the corresponding kernel weights (both vertical and horizontal), and then added. The resulting gradients are combined to acquire the total gradient magnitude. The total magnitude of both gradients is ideally given by Equation 3.9. However, the approximation approach shown in Equation 3.10 can also be applied. Finally, the output of this Sobel filter is sent to the next image processing algorithm.

$$G_x = \frac{1}{8}\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I \text{ and } G_y = \frac{1}{8}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I \tag{3.8}$$

where $\frac{1}{8}$ refers to the normalization factor for the Sobel filter.

$$G = \sqrt[2]{G_x^2 + G_y^2} \tag{3.9}$$

$$G = |G_x| + |G_y| \tag{3.10}$$

The circle detection algorithm receives the edge filtered image, which enables it to calculate the locations of the circles that represent the robots' locations. As depicted in Figure 3.17, two circle detection algorithms are presented in this work for the multi-robot tracking application. The algorithm is intended to be implemented in the hardware accelerator (FPGA/GPU). The first method integrates a combination of the circle Hough transform (CHT) and graph cluster algorithms. The second combines the circle scanning window (CSW) technique and graph cluster algorithm. The CHT and CSW are used to generate the circle center candidates. These candidates are then provided to the graph cluster, which analyzes all the candidates and calculates the true centers of the circles.

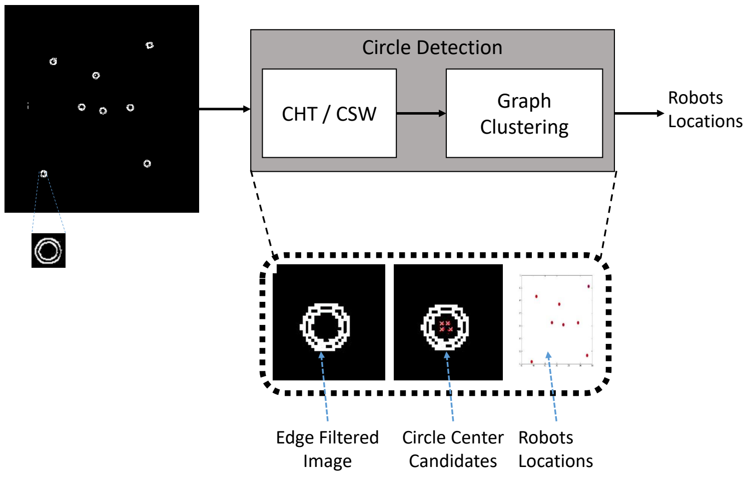


Figure 3.17: CHT/CSW and graph clustering algorithms for the circle detection.

3.3.2.1 Circle Hough transform

One of the most popular methods in circle detection is the CHT algorithm, an extended version of the Hough transform (HT). The generalized HT is a feature extraction technique that is usually used in image analysis, computer vision, and digital image processing [101]. It was invented by Richard Duda and Peter Hart in 1972 [37] based on the related 1962 patent of Paul Hough [54]. Basically, the generalized HT is used to detect geometrical curves such as lines, circles, and ellipses, while the CHT is specifically designed to find circles using a voting procedure.

In a binary image, the HT can be used to determine the parameters of a circle when the number of points that fall on the perimeter are known [31]. A circle with center (a, b) and radius r is specified by the parameters (a, b, r) in Equation 3.11

$$(x - a)^2 + (y - b)^2 = r^2 \tag{3.11}$$

The CHT maps each of the binary image pixels into many points in the Hough (or parameter) space. If the circles in the image are of a known radius r, then the search for a circle is a two-dimensional computation, as illustrated in Figure 3.18. The objective is to find the (a, b) coordinates of the circle's center candidates, as shown in Equation 3.12. Angle α sweeps through the full 360° with distance r for every (x, y). The locus of the points (a, b) in the parameter space fall on a circle of radius r centered at (x, y). The true center point will be common to all parameter circles and can be found using an HT voting procedure [31].

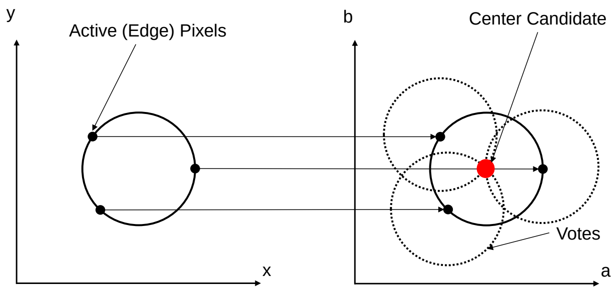


Figure 3.18: CHT from x,y-space (left) to parameter space (right) for a constant radius.

$$a = x - r \cdot \cos\alpha \quad and \quad b = y - r \cdot \sin\alpha \tag{3.12}$$

3.3.2.2 Circle Scanning Window

As an alternative method to find the circle center candidates, we have implemented the circle scanning window technique. Unlike the CHT method, which uses a one-to-many points approach by mapping each of the binary image pixels to many points in the Hough space, the circle scanning window (CSW) technique uses a many-to-one approach. It maps many pixels of the binary image space to one point to find the circle center candidate. While the voting values in the CHT method are generated from the accumulation of the points in the transformed space, in the CSW method, the voting values are obtained directly from the binary image. A point (coordinate) is considered to be a circle center candidate of the CSW when the voting value for this coordinate is higher than the selected threshold value.

As shown in Figure 3.19, a scanning window with its circle pattern pixels is used to find the circle center candidates. In our application, the radius of the circle in the robot marker is defined. Thus, a specific size is used for the scanning window. The $N \times N$ pixels of the scanning window are chosen based on the diameter size of the circles in the edge filtered image (binary image). This window consists of a circle pattern with a predefined radius. The CSW moves in the raster scan mode, scanning the entire image frame to find the circle center candidates. A location is considered to be a circle center candidate if the accumulated voting value in the scanning window block is higher than a selected threshold value. The calculation of the voting value refers to the accumulation of the binary pixels in the scanning window using the predefined circle pattern. The calculation of the circle pattern coordinates is based on Equation 3.13.

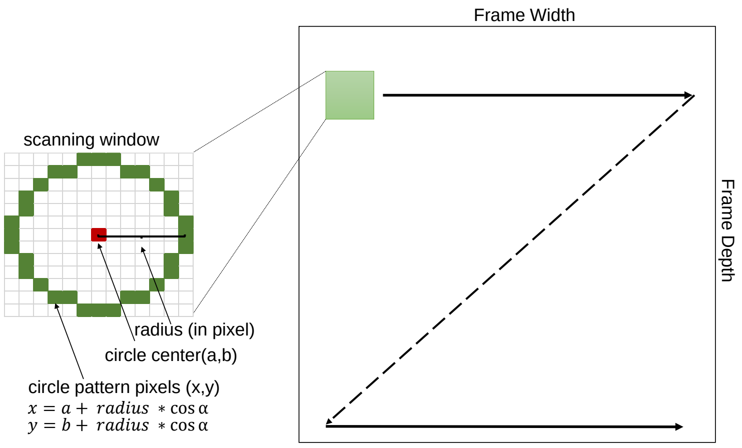

Frame Width
scanning window
Frame Depth
radius (in pixel)
circle center(a,b)
circle pattern pixels (x,y)
$x = a + radius * \cos\alpha$
$y = b + radius * \cos\alpha$

Figure 3.19: Raster scan with circle detection scanning window.

$$x = a + round(radius \cdot \cos\alpha) \quad and \quad y = b + round(radius \cdot \sin\alpha) \tag{3.13}$$

3.3.2.3 Graph clustering

The process of identifying the structure of non-uniform data, in terms of grouping the data elements, is known as clustering or data classification [66; 95]. The goal of clustering is to identify all of the groups in a set of unlabeled data. Graphs are structures that are formed by a set of vertices (also called nodes) and a set of edges, which are connections between pairs of vertices. Graph clustering is the task of grouping the vertices of the graph into clusters, taking into consideration the edge structure of the graph [95]. Figure 3.20 illustrates an example of graph clustering.

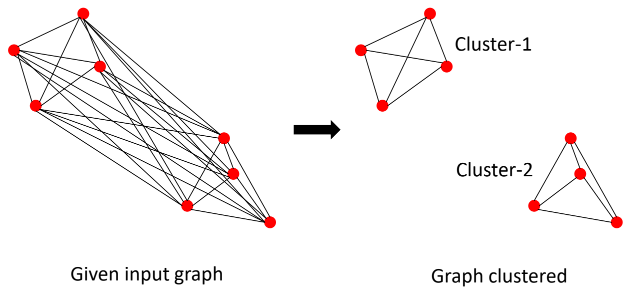


Figure 3.20: Graph clustering operation.

Formally, given a data set, the goal of clustering is to divide the data set into clusters such that the elements assigned to a particular cluster are similar or connected in some predefined sense [95]. In other words, clustering is used to find patterns in data or to group sets of data points together into clusters. Because our design is dedicated to circle detection in multi-robot localization, the design of the clustering method is based on the characteristics of the graph data provided by either the CHT or CSW outputs. In this application, the outputs of the CHT and CSW represent all candidates that have voting values higher than the predefined threshold ($voting_{th}$). As shown in Figure 3.21, the circle center candidates are all located inside the circle of the robot markers, but further processing is needed to obtain the true centers of the circles.

According to the data characteristics, the clustering is built based on the distance between the coordinates of the circle's center candidates determined by the previous module. The coordinates are assigned to the same cluster if the distances between them are lower than the distance threshold. In our application, one cluster represents one robot marker. Finally, the centroid of each cluster is calculated. These centroids become the true centers of the circles, which represent the locations of the robot markers.

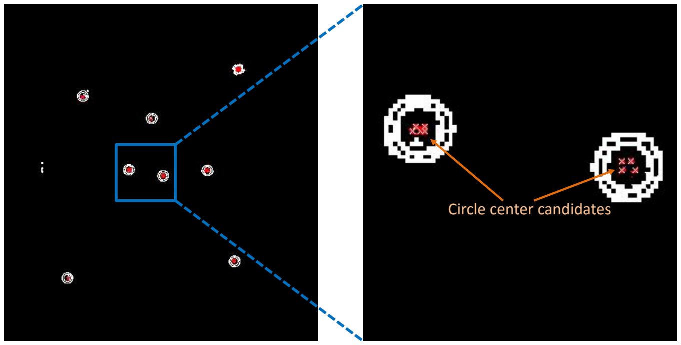


Figure 3.21: Edge detection and its circle center candidates.

The algorithm for the graph clustering operation is illustrated using a flow chart, as depicted in Figure 3.22. This flowchart shows that all of the circle center candidates should initially be ready to be stored in the memory or registers. The first circle center candidate is automatically considered to be the first cluster, and a new cluster ID is subsequently created. When this candidate is not identified as the last one, the next candidate is loaded and compared with previous candidates, in order to define the connection between the new/current candidate and the former candidates. The coordinates between two or more center candidates are considered to be connected or collided when the distance between them is lower than a threshold value. If no collision is detected, a new cluster ID is created. Otherwise, when a collision has occurred and is related to one of the cluster IDs, the current center candidate is set using this cluster ID, and then merged with the collided center candidate. If the collision is connected to two or more cluster IDs, all of the candidates with the same value as those collided cluster IDs are merged and updated with one of those cluster IDs (e.g., the smallest ID). These complete processes are repeated until the last circle center candidate appears. Finally, all of the centroids (circle centers) are calculated based on the average value of all the members in each cluster (group). These centroids represent the locations of the robots, which are sent to the host PC for further processes, including tracking and visualization.

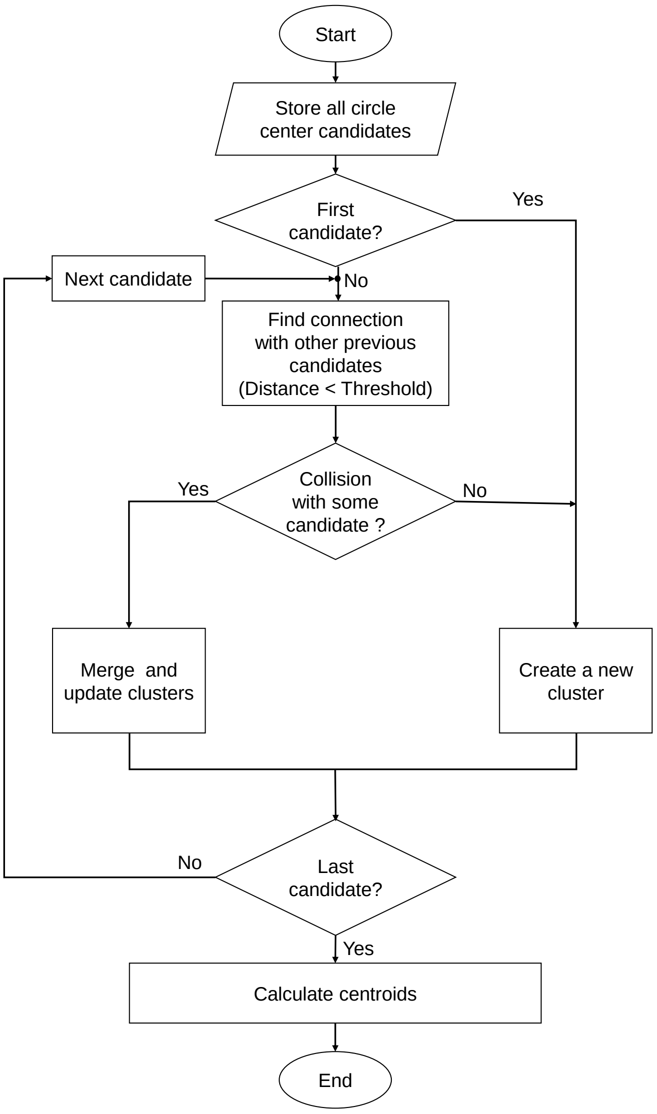

Start
Store all circle center candidates
First candidate?
Yes
No
Next candidate
Find connection with other previous candidates (Distance < Threshold)
Collision with some candidate ?
Yes
No
Merge and update clusters
Create a new cluster
No
Last candidate?
Yes
Calculate centroids
End

Figure 3.22: Graph clustering flowchart.

3.3.3 Post-processing

As previously explained, this work uses the same pentagon and barcode as utilized in the Teleworkbench [103] robot marker. Therefore, the post-processing step applies the same algorithm and library for detecting the orientations and IDs of the robots. This approach allows robot marker identification of up to 64 unique IDs with detection rates of 100% and 99.99% in static and dynamic tests, respectively [103].

Figure 3.23 depicts the flow chart of the post-processing algorithm. As illustrated in this flow chart, the output of the hardware accelerator (robot markers' locations) is used to generate numerous sub-images (cropped images). Then, the complete post-processing algorithm is performed on these images.

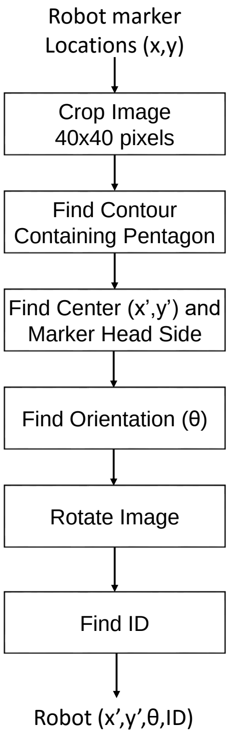


Figure 3.23: Post processing algorithm in our application.

Figure 3.24 shows examples of the cropped images (sub-images) with different orientations of the robots with their markers. The size of each image is 40 × 40 pixels. All of these sub-images are collected and buffered in the host PC's memory, and then they are all processed by the quad-core CPU using a multi-thread approach. More details about the multi-thread processing in the host PC will be discussed later in section 4.5, while this section focuses on the post-processing algorithm.

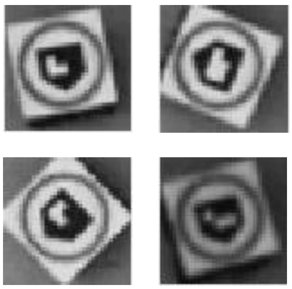


Figure 3.24: Example of image robots with their marker that have been cropped based on locations from hardware accelerators (FPGA/GPU).

To calculate the robot orientation in the 40 × 40 pixels of the cropped image, the algorithm begins by finding the pentagon. The *findContours* and *minAreaRect* functions from OpenCV library are applied to find the center of a rectangle that covers the whole area of the pentagon, as shown in Figure 3.25. Using this approach, the circle center coordinates from the hardware accelerator (FPGA or GPU) can be improved because the pentagon is located in the center of the circle. Afterward, the head of the pentagon can be calculated and detected.

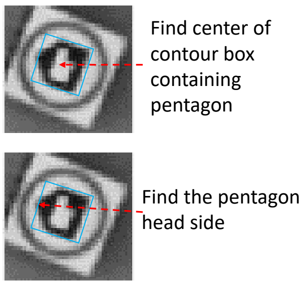

Find center of contour box containing pentagon
Find the pentagon head side

Figure 3.25: Finding contour box of pentagon and its head side for calculating orientation.

The orientation of the robot marker is calculated based on the previous information about the pentagon's center and head side. Finally, using the orientation value, the $getRotationMatrix2D$ function from OpenCV is applied to rotate the image back into a standard position, as depicted in Figure 3.26, so that the robot's ID can be decoded from the barcode.

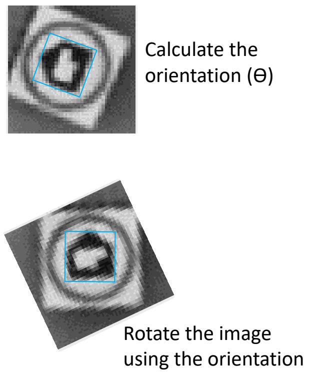

Calculate the orientation (Θ)
Rotate the image using the orientation

Figure 3.26: Calculate robot marker orientation and use orientation to rotate image.

3.4 Summary

The architectures of heterogeneous computing systems for vision-based multi-robot tracking and their design flows in both FPGA-and GPU-accelerated platforms have been presented in this chapter. Furthermore, this chapter has presented algorithms to detect the locations, orientations, and IDs of the robots. These algorithms consist of segmentation, robot detection, and post processing. For the robot marker detection, two unique algorithms have been introduced. The first one integrates a combination of the CHT and graph cluster algorithms. The second one combines the CSW technique and graph cluster algorithm. Additionally, this chapter has shown how the task partitions are divided between the hardware accelerators (FPGA/GPU) and CPU. The segmentation and robot detection algorithms are designed to be implemented in hardware accelerators because of their computationally intensive tasks and high data rate. Meanwhile, the CPU is used for executing some complex and control operations with low data rates in the post-processing algorithm.

More details about the implementation of the FPGA-accelerated computing system for vision-based multi-robot tracking can be found in chapter 4, while the implementation with the same approach using the GPU-accelerated computing system is presented in chapter 5.

4 Implementation in FPGA-accelerated Heterogeneous Computing System

This chapter emphasizes the implementation of FPGA-accelerated heterogeneous computing systems for vision-based multi-robot tracking. The FPGA architecture possesses the benefits of a parallel structure and customizable design, which could be used to increase the computing performances. In particular, this chapter delineates the system architecture of the proposed design and the implementation of several video processing modules on the FPGA, with the goal of constructing the complete proposed system. In addition, it presents three basic configurations with different numbers of streaming hardware accelerators and thus different levels of parallelism in the implementation. Additionally, this chapter presents two unique architectures for FPGA-based circle detection for multi-robot tracking, using the CHT-graph cluster algorithm and CSW technique-graph cluster algorithm combinations. Finally, it analyzes the logic resource requirements of all the video processing modules.

Figure 4.1 shows the architecture of the FPGA-CPU heterogeneous computing system for vision-based multi-robot tracking applications. It is a vision-based multi-robot tracking platform that utilizes FPGAs for video processing and a host PC for post-processing and visualization. The architecture supports a scalable number of GigE Vision cameras with a maximum frame size of 1024 × 1024 pixels for each camera, allowing them to cover a robot arena with a size of $6\ m \times 6\ m$. Each robot is individually labeled with a specific marker for distinct identification. Thus, the system can easily support different types of robots. Four cameras are directly connected to the FPGA, and most parts of the video processing chain for detecting the locations of robot markers are implemented in hardware to achieve the maximum advantage of using FPGA technology.

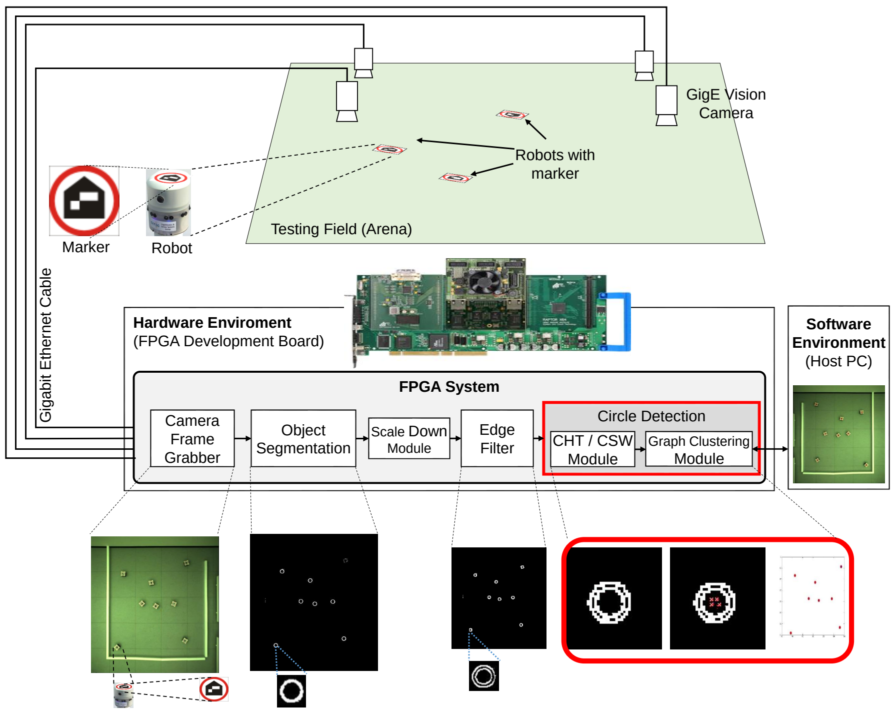


Figure 4.1: Top-level block diagram of system architecture with circle detection IP core.

4.1 FPGA-CPU Hardware Environment Description

The proposed design is implemented on the CPU and modular FPGA-based rapid prototyping system RAPTOR [91], as shown in Figures 4.2 and 4.3. It uses an Intel i7 quad-core CPU as the processor in the host PC and Virtex-4 FPGA on the RAPTOR development board as the hardware accelerator. For communication and data transfer between the CPU and FPGA, the system uses the peripheral component interconnect (PCI) bus, while the camera can be directly connected to the FPGA daughter board module (DBM). Up to six DBMs can be attached in the RAPTOR system; it offers system scalability in terms of the number of FPGAs and cameras.

RAPTOR is a modular base rapid prototyping system, which was designed by Cognitronics and Sensoric Research Group, Bielefeld University. It applies a modular approach, which consists of a base system and variety of extension daughter board modules. The base system provides communication and management functionalities,

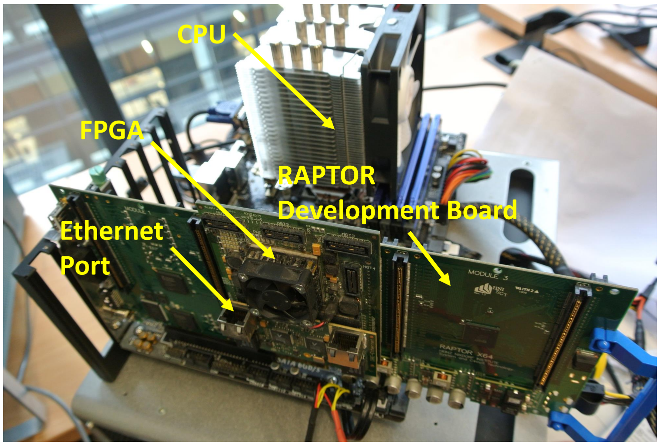


Figure 4.2: FPGA-CPU hardware environment picture.

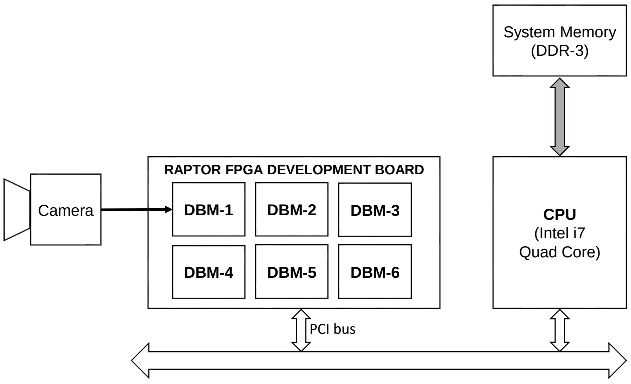


Figure 4.3: FPGA-CPU hardware environment block diagram.

which are used by the extension daughter boards. Figure 4.4 illustrates the architecture of the RAPTOR development board. The RAPTOR system supports up to six FPGA daughter board modules. In our implementation, each daughter board consists of one FPGA and one Ethernet board, providing two Gigabit Ethernet ports. Our design was designed to use only one FPGA to handle four cameras. However, it is possible to support more cameras and Gigabit Ethernet interfaces by utilizing additional FPGA daughter boards, without the need for extra host PCs.

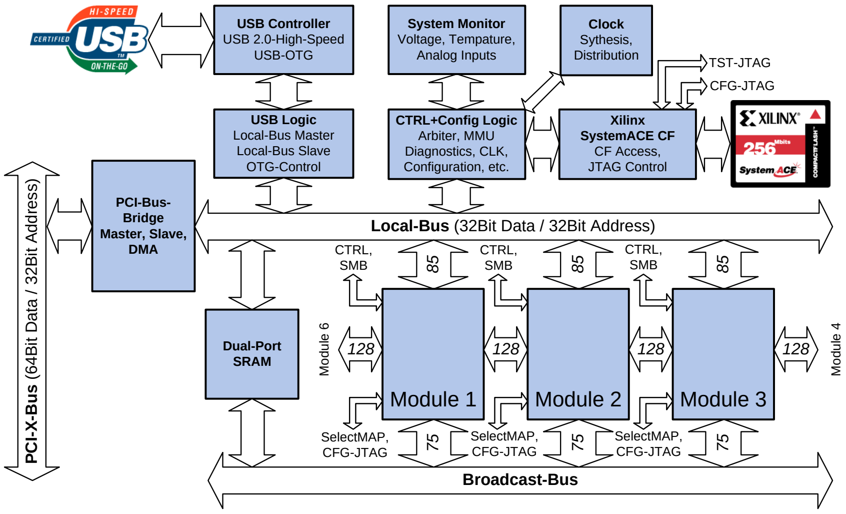


Figure 4.4: RAPTOR development board architecture [91].

Figure 4.5 shows details of the complete platform of the proposed design, the so-called FPGA hardware environment. A multi-port memory controller (MPMC) [116] is used as the memory controller. It supports LL SDMA, VFBC, NPI, and PLB interfaces. The MPMC provides access to the external DDR2-SDRAM memory through one to eight ports. This memory is necessarily used as a frame buffer. In this regard, the AXI4-S to video frame buffer controller (VFBC) becomes a bridge that connects the VFBC port of the MPMC and the user IP core. The VFBC allows a user IP core to read and write data in two-dimensional (2D) formats regardless of the size or organization of the external memory transactions [116]. Furthermore, a local bus slave to NPI controller is designed for a control signal and data transfer between the FPGA and the host PC through the PCI interface [63]. All of the processes (from grabbing the video frame from the camera to detecting the location of the robot through its marker) are implemented in the FPGA as individual modules (IP cores).

Finally, the proposed design with its base system and all of the video processing modules are implemented using the RAPTOR prototyping system populated with a Xilinx Virtex-4 XC4VFX100-11 FPGA. The following sections provide more details on the design of each video processing module and its implementation, including the multi-camera GigE Vision frame grabber module, preprocessing module, edge filter module, and circle detection module.

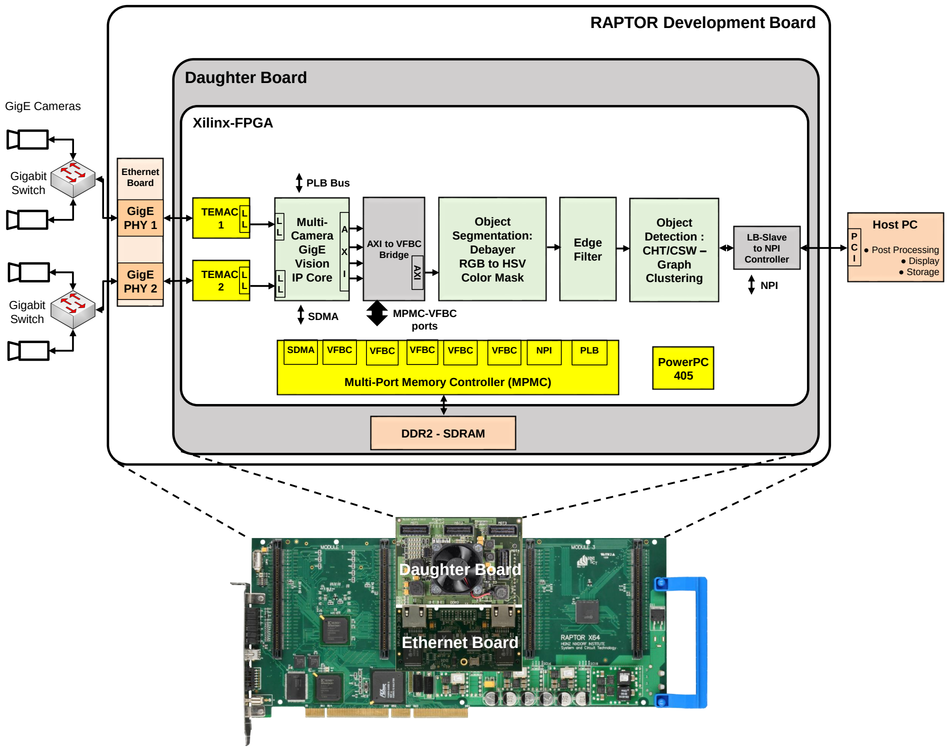


Figure 4.5: Complete platform of FPGA hardware environtment.

4.2 Algorithm Implementation

Improving the computational performance is the main aim of the proposed FPGA-CPU computing system. Therefore, the algorithm implementation for vision-based multi-robot tracking is distributed between the FPGA and CPU, as depicted in Figure 4.6. The FPGA implementation comprises four main modules, which are used to identify the locations of all the robots. Meanwhile, the image post-processing algorithm in the host PC plays a role in detecting the robot orientations and IDs. The computations are implemented based on a 40 × 40 pixel image for each robot marker, using some functions in the openCV library. The post processing algorithm in the CPU is the same algorithm used for Teleworkbench [103].

The FPGA hardware implementation includes the GigE Vision camera frame grabber, object segmentation, edge detection, and circle detection modules, as shown in Figures 4.1 and 4.6. The GigE Vision camera grabber module captures the video frames directly from the multiple cameras. The object segmentation module includes the debayer, RGB to HSV color conversion, and color mask operations to extract the circles from the image based on the marker's color using color segmentation. This segmented image is a binary image containing the extracted circles. This image is downscaled to reduce the number of required logic resources. It is scaled down by a factor of two in the x and y directions by the integrated downscaling module. A factor of two is chosen because this reduction size can still maintain a good recognition of the circle. The image is scaled down by skipping all the odd columns and rows. Afterward, the resized image is delivered to the edge detection module, which applies a Sobel filter. This filter is applied as a preprocessing step for removing large colored areas, not representing markers. Finally, the circle detection module acquires the centroids of the circles, which represent the locations of the robots. This thesis presents two unique FPGA-based circle detection architectures for multi-robot tracking applications. The first one integrates a CHT and graph cluster algorithm combination. The second architecture combines the CSW technique and a graph cluster algorithm. After the circle has been detected, the robots' locations are sent to the host PC for further processing, including tracking and visualization.

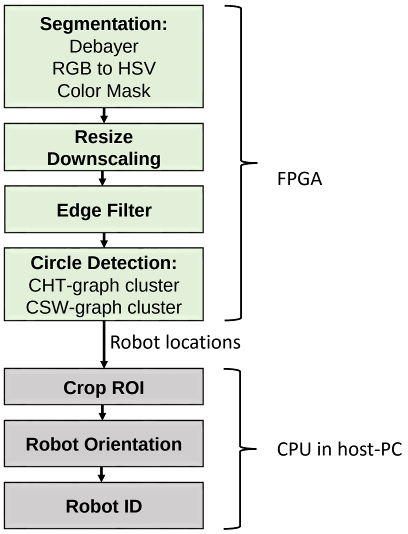
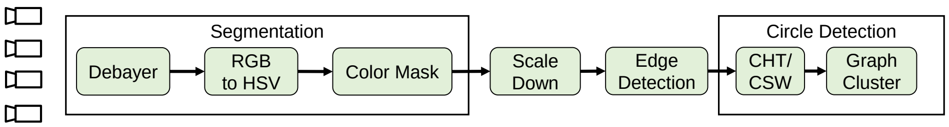
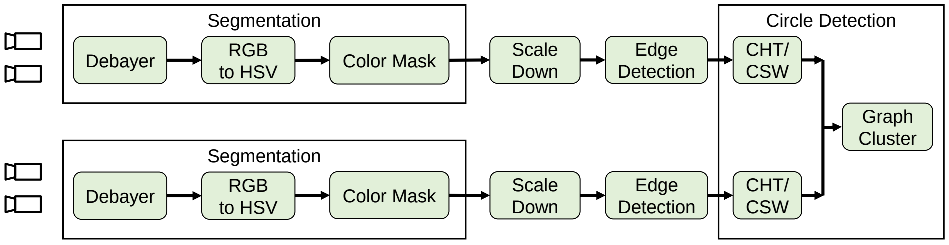


Figure 4.6: FPGA-CPU algorithm distribution.

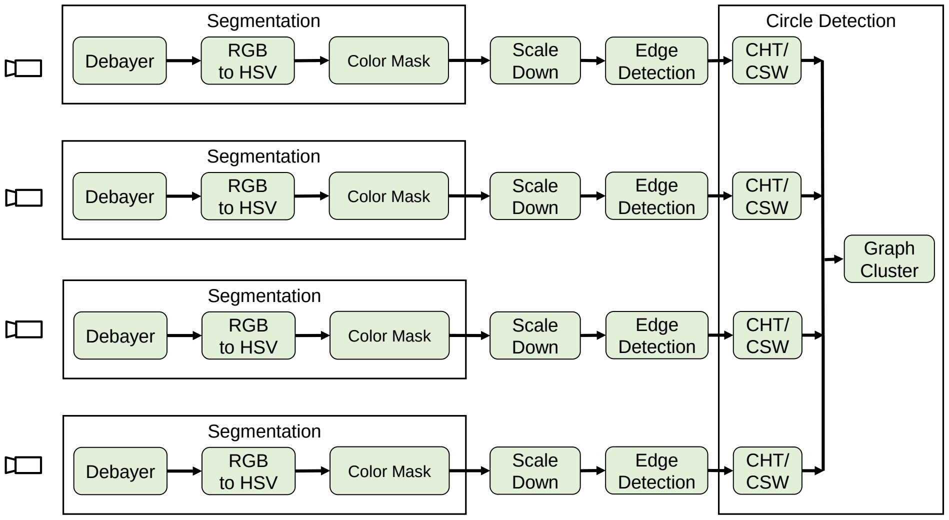
This design supports multiple cameras with different configurations of video processing algorithms as hardware accelerators. Figure 4.7 shows three configurations (A, B,and C) of video processing hardware accelerators. It presents three basic configurations that differ in the number of streaming hardware accelerators and thus in the parallelism of the implementation.



(a) Configuration A



(b) Configuration B



(c) Configuration C

Figure 4.7: Examples of different configurations of video processing hardware accelerators.

Configuration A supports four cameras using only one video stream hardware accelerator, as shown in Figure 4.7-a. Figure 4.7-b illustrates configuration B, which utilizes two parallel video stream hardware accelerators, i.e., each video stream hardware accelerator handles two cameras. This approach can significantly increase the performance of the system. Finally, for the maximum performance, one video stream hardware accelerator can be applied for every single camera, as shown in configuration C (Figure 4.7-c). The number of parallel video stream hardware accelerators is flexibly scalable, because it is limited only by the number of logic resources inside the FPGA.

Figure 4.8 depicts an example of a configuration that supports an application that uses two different colors (e.g., red and blue) for the circle in the robot marker. Each color uses stream hardware accelerators to obtain the locations of the robots. Using this configuration, the system performance will be comparable to the system that uses configuration A, but the supported maximum number of robot IDs is doubled from 64 to 128. The configuration offers scalability in terms of the numbers of robots and IDs supported without sacrificing the computing performance.

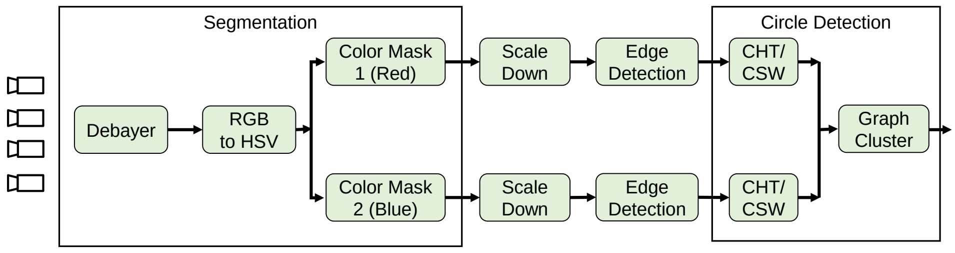


Figure 4.8: Example of configuration that uses two different colors for circle in robot marker, which can increase maximum numbers of used robots and IDs.

4.3 Vision Processing Module Implementation in FPGAs

This section provides more details on the FPGA implementation of the hardware accelerator module, as shown in Figures 4.1 and 4.6. It explores the design of the multi-camera frame grabber, object segmentation, edge filter, and circle detection modules.

4.3.1 Multi-Camera GigE Vision Frame Grabber Module

Figure 4.1, in the beginning of this chapter, shows that the design uses a GigE Vision frame grabber module for directly interfacing with all the cameras. This module is a scalable and lightweight FPGA implementation of the GigE Vision standard designed by Congnitronic and Sensoric Research Group, CITEC, Bielefeld University [58]. It has the ability to reconstruct the video frames at wire-speed by extracting the raw video data from the GigE Vision packets of each camera. Multiple GigE Vision cameras (not utilizing the complete bandwidth of 1 Gbit/s) can be connected to this module through a single Gigabit Ethernet interface and Gigabit Ethernet switch.

GigE Vision is a global and widely adopted camera interface standard, which was developed using the Gigabit Ethernet (IEEE 802.3) communication standard. It allows for fast image transfer using the available bandwidth of 1 Gbps (faster data transfer is supported using link aggregation and 10 Gigabit Ethernet). Additionally, GigE Vision supports long-distance data transmission of up to 100 m using low-cost copper cables (CAT5e, CAT6a, or CAT7). This transmission distance can be increased using switches or optical fiber cables. With GigE Vision, hardware and software from different vendors can interoperate seamlessly over GigE connections [7]. GigE Vision is based on UDP/IP, and the raw video frames are packetized and transmitted as GigE Vision packets from the GigE Vision camera.

In our vision-based multi-robot tracking platform, each pair of the four GigE Vision cameras are connected to one Gigabit Ethernet PHY via a Gigabit Ethernet switch, as shown in Figure 4.9. In this approach, the resources required for interfacing the cameras are reduced because only two gigabit PHYs, two TEMACs, and two multi-camera GigE Vision IP cores are used (instead of four gigabit PHYs, four TEMACs, and four single-camera GigE Vision IP cores).

The multi-camera GigE Vision video frame grabber module consists of the tri-mode Ethernet media access controller (TEMAC), the multi-camera GigE Vision (MC_GigEV), and the camera configuration IP cores as shown in Figure 4.9. The video stream from each camera consists of GigE Vision packets that encapsulate the raw video data.

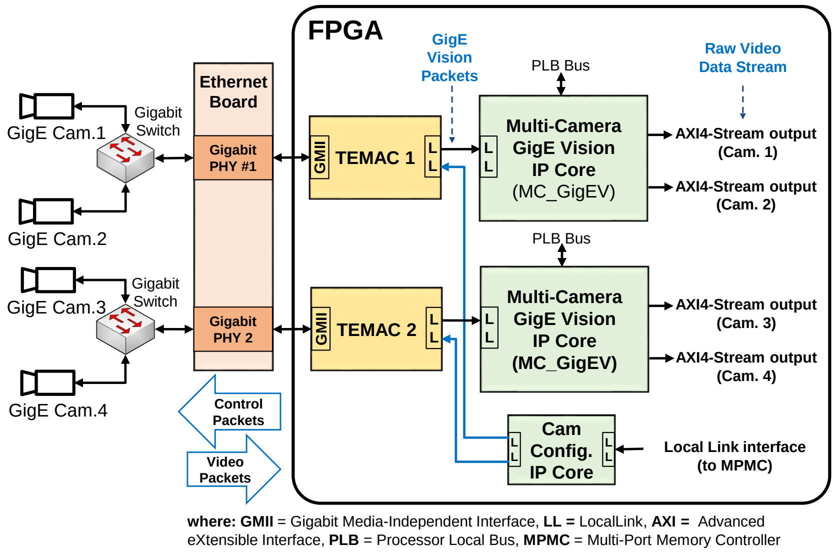

FPGA
GigE Vision Packets
PLB Bus
Raw Video Data Stream
Ethernet Board
Gigabit PHY #1
Gigabit PHY 2
GigE Cam.1
GigE Cam.2
GigE Cam.3
GigE Cam.4
Gigabit Switch
Gigabit Switch
GMII
TEMAC 1
TEMAC 2
LL
Multi-Camera GigE Vision IP Core (MC_GigEV)
AXI4-Stream output (Cam. 1)
AXI4-Stream output (Cam. 2)
AXI4-Stream output (Cam. 3)
AXI4-Stream output (Cam. 4)
Cam Config. IP Core
Local Link interface (to MPMC)
Control Packets
Video Packets
where: GMII = Gigabit Media-Independent Interface, LL = LocalLink, AXI = Advanced eXtensible Interface, PLB = Processor Local Bus, MPMC = Multi-Port Memory Controller

Figure 4.9: Multi-camera GigE Vision module used in TWB [62].

Afterward, the TEMAC controller that is responsible for the implementation of the Ethernet link and physical layers [117] receives the GigE Vision packets, using its Gigabit Media Independent Interface (GMII). It continuously passes the packets from different camera sources to the MC_GigEV IP core. The MC_GigEV IP core extracts the raw video data and reconstructs the video frames from each video stream. Finally, the core provides the extracted video data as an AXI4-Stream in a separate channel for each video stream, which allows the video data to be easily processed further. To configure the cameras with the desired frame rates and resolutions, GigE Vision control packets are sent to the desired camera through the camera configuration IP core (Cam_Config).

4.3.2 Object Segmentation

The output of the multi-camera video frame grabber module is the raw video data in a Bayer pattern format [14]. Further processing is needed to distinguish the robot markers from the background image. In this multi-robot tracking application, the red circle in the robot marker is extracted using a color segmentation algorithm. This object segmentation module includes a debayer, RGB to HSV color conversion, and color mask units, as shown in Figure 4.10. The debayer unit creates a full RGB color image out of Bayer encoded image. Afterward, the RGB to HSV unit converts the color space from

RGB to HSV. It is implemented to provide a more robust segmentation performance than the RGB color space with respect to changes in the illumination and lighting [6; 78]. Finally, the color mask unit thresholds the HSV image to extract the red circles in the robot marker. All of these units are fully pipelined. Each unit performs its operation and passes the result to the next stage (unit).

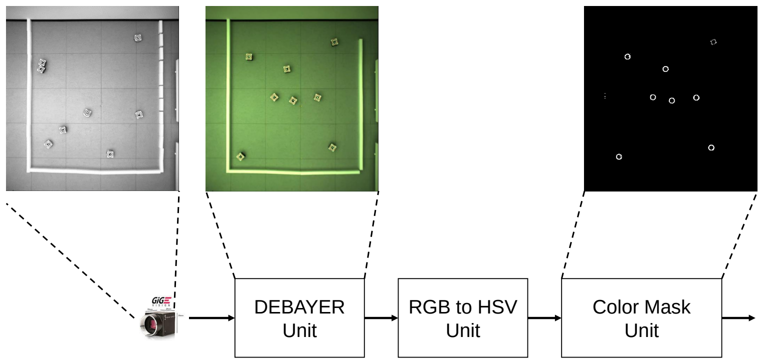


Figure 4.10: Top-level block diagram of segmentation module [62].

The implementation of the debayer unit in VHDL is based on the bilinear interpolation algorithm, which was described in section 3.3.1.1 using Equation 3.1, 3.2, and 3.3. Figure 4.11 shows the hardware implementation block diagram of the bi-linear interpolation. It utilizes two-row buffers to form a 3×3 window for an eight neighborhood filter operation. The multiplexer outputs depend on the pixel that is currently being processed. The operation of the debayer unit requires a latency of $2 \times image\ width$, which is equal to the total length of the row buffer. Latency refers to the difference in the times (clock cycles) that the data is first input to an operation and the corresponding output is available [8].

In this work, the RGB to HSV unit is designed based on the algorithm of Foley et al. [41], which has been previously described in section 3.3.1.2. The implementation is based on Equations 3.4, 3.5, and 3.6. The hue and saturation color conversions in Equation 3.4 and Equation 3.5 require a divider operator, which is inefficient in relation to the logic resource requirement in an FPGA design. Therefore, these equations are modified to obtain a more efficient design in terms of the logic resource requirement by removing the divider operator. The modifications of the hue and saturation formulas are comprehensively presented in Equations 4.1 and 4.2. Finally, these two modified equations, together with Equation 3.6, are implemented as a hardware accelerator in the FPGA. Using these equations, the divider is not needed, as depicted in Figure 4.12.

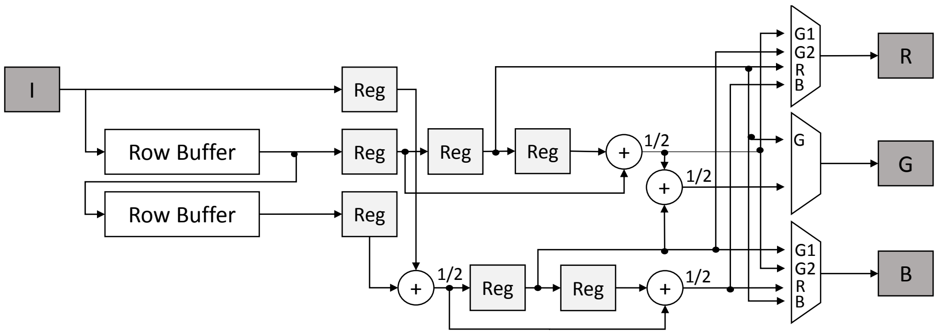


Figure 4.11: Bilinear interpolation block diagram [8].

$$\frac{H}{2} \times \Delta = \begin{cases} 0 & \text{, if } R = G = B \\ 30 \times (G - B) & \text{, if } R = max(R,G,B) \\ 30 \times ((2 \times \Delta) + (B - R)) & \text{, if } G = max(R,G,B) \\ 30 \times ((4 \times \Delta) + (R - G)) & \text{, if } B = max(R,G,B) \end{cases} \tag{4.1}$$

where $\Delta = max(R,G,B) - min(R,G,B)$ and for a condition where the output is negative: if $\frac{H}{2} \times \Delta < 0$, then $\frac{H}{2} \times \Delta = \frac{H}{2} \times \Delta + (180 \times \Delta)$

$$S \times max = \begin{cases} \Delta & \text{, if } max(R,G,B) \neq 0 \\ 0 & \text{, if } max(R,G,B) = 0 \end{cases} \tag{4.2}$$

The FPGA implementation of color mask units is based on Equation 4.3, which is a modification of Equation 3.7. As shown in Figure 4.12, the color mask unit directly thresholds the output from the RGB to HSV unit. It consists of comparators with threshold parameters (the highest and lowest values for each of the HSV channels). The values of these threshold parameters must use the applied range values for the HSV channels. They are between 0 and 180 for the H-channel, 0 and 128 for the S-channel, and 0 and 255 for the V-channel. A comparison operation occurs between the HSV image and the set threshold to obtain the output. The output of this unit is a binary image, where the pixels are set to white (active pixels) if their HSV values fall within

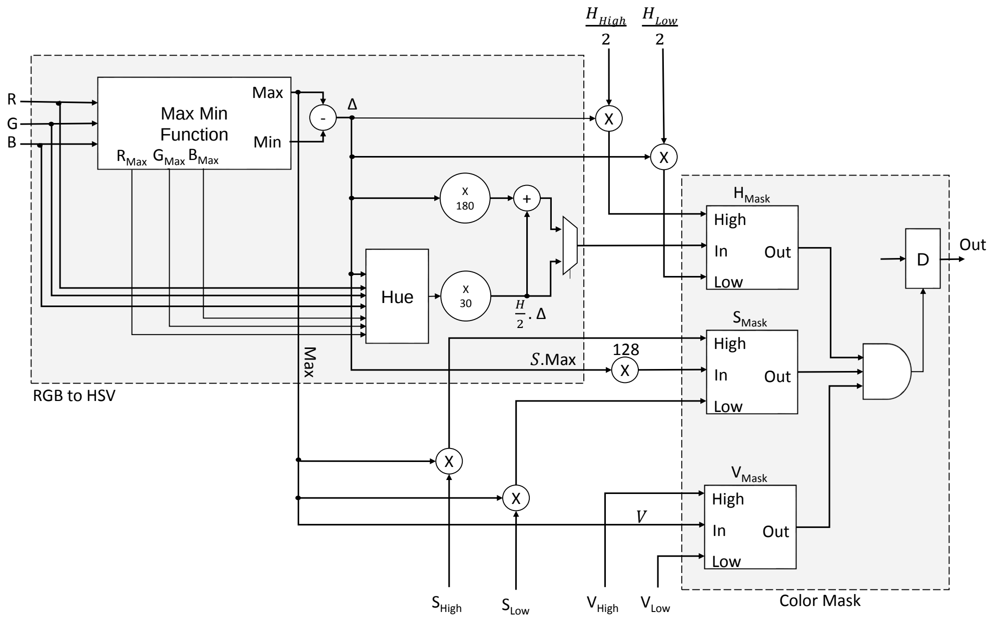


Figure 4.12: RGB to HSV and color mask units.

the specified threshold parameters in all three channels. Otherwise, the pixels are set to black.

$$ColorMask = \begin{cases} 255 & \text{, if } H_{mask} = S_{mask} = Vmask = 1 \\ \\ 0 & \text{, otherwise} \end{cases} \tag{4.3}$$

where H_{mask}, S_{mask}, and V_{mask} can be obtained using the following formulas:

$$H_{mask} = \begin{cases} 1 & \text{, if } \frac{H_{Low}}{2} \times \Delta \leqslant \frac{H}{2} \times \Delta \leqslant \frac{H_{High}}{2} \times \Delta \\ \\ 0 & \text{, otherwise} \end{cases}$$

$$S_{mask} = \begin{cases} 1 & \text{, if } S_{Low} \times max \leqslant S \times max \leqslant S_{High} \times max \\ 0 & \text{, otherwise} \end{cases}$$

$$V_{mask} = \begin{cases} 1 & \text{, if } V_{Low} \leqslant V \leqslant V_{High} \\ 0 & \text{, otherwise} \end{cases}$$

4.3.3 Edge Filter Module

This design uses a Sobel filter for the edge detection filter. As previously described in section 3.3.2, a Sobel filter is a gradient-based method that applies two 3×3 kernels that are convolved with the input image to approximate the horizontal and vertical gradients. Figure 4.13 shows a block diagram of the Sobel filter module. It utilizes one pair of line buffers to execute two 3×3 kernel windows simultaneously. During the gradient computation, each pixel within the image (I) is multiplied in parallel by the corresponding kernel weight and then added. The resulting gradients are then combined to obtain the total gradient magnitude.

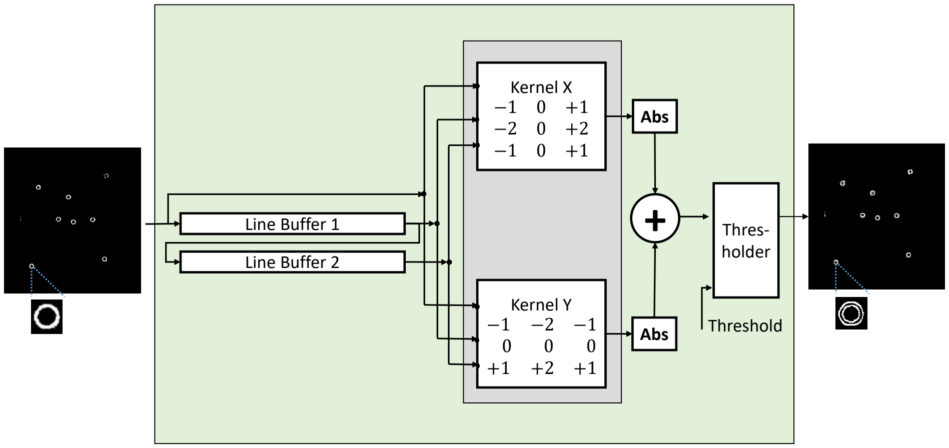


Figure 4.13: Block diagram of Sobel fiter module.

The total magnitude of the gradients is ideally given by Equation 3.9, which is $G = \sqrt[2]{G_x^2 + G_y^2}$. However, this equation is too costly in terms of the logic resource requirements for the FPGA design. As an alternative, the approximation approach shown in Equation 3.10 ($G = |G_x| + |G_y|$) can be applied. Finally, the total gradient magnitude value is compared with the selected threshold value. A pixel is set to white if the gradient value is higher than the specified threshold parameter. Otherwise, the pixel is set to black.

The logic resource requirement of this Sobel filter module relies on the image size. In particular, the length of the line buffers is equal to the image width size. Resizing (downscaling) the segmented image is similar to reducing the logic resources. This resizing also causes a reduction in the required logic resources in the circle detection module.

4.3.4 Object Localization

The circle detection module has the main task of defining the locations of the circles, which represent the locations of the robots. In the literature, several FPGA-based circle detection designs have been proposed, e.g., [5; 39; 64; 99; 122]. However, this work targets a resource-efficient approach that supports the detection of multiple circles (objects) in real-time based on high-resolution video frames, which is not supported by any of the above-mentioned implementations.

This work presents two unique architectures for FPGA-based circle detection for a multi-robot tracking application, as depicted in Figure 4.14. The first architecture integrates a combination of the CHT and graph cluster algorithms. The second one combines the CSW technique and graph cluster algorithm. The CHT and CSW are used for the generation of circle center candidates. These candidates are then provided to the graph cluster, which analyzes all the candidates and calculates the true centers of the circles. To reduce the number of required logic resources, the input video (segmented image) is first scaled down by a factor of two in the x and y directions using the integrated scaling module. This resizing also causes a reduction in the binary image for the CHT module or CSW module, as shown in Figure 4.1. The diameter of the circle markers in the real world is 10 cm (line thickness: 8 mm), corresponding to a diameter of 26 pixels and an average line thickness of 3 pixels in the video frame. More details about the CHT, CSW, and graph clustering modules are presented in the following subsections.

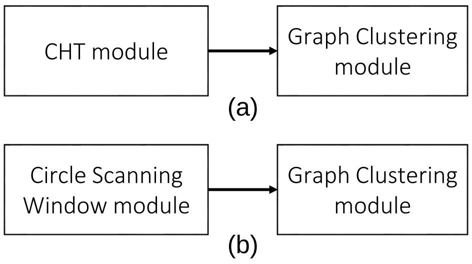


Figure 4.14: Proposed circle detection module: (a) CHT-graph cluster and (b) CSW-graph cluster.

4.3.4.1 Circular Hough transform module

The FPGA implementation of the CHT algorithm is based on Equation 3.12, which was previously described in section 3.3.2.1. In this implementation, 16 and 32 values of α are used to sweep the full 360°. As a result, we will have 16 or 32 votes for each active pixel. These votes are considered to be a good trade-off between the circle detection performance and the number of required logic resources for the FPGA, as listed in Table 4.1. The 16 votes approach requires fewer logic resources but achieves a slightly lower performance than the 32 votes approach for detecting the circle. Here, the precision and recall are used as the standard detection metrics, as described in [44]. The precision is the ratio between the number of correctly detected circles (true positives (TP)) and all the detections (true positives (TP) and false positives (FP)), as shown in Equation 4.4. The recall is the number of circles that are correctly detected out of the total number of circles that should have been detected (true positives and false negatives (FN), i.e., the ground truth).

$$Precision = \frac{TP}{TP+FP}, \; Recall = \frac{TP}{TP+FN} \tag{4.4}$$

The values in Table 4.1 are based on Matlab simulations using 16 generated test circles and 12 different scenarios (images) with respect to the circle positions, in particular for collision conditions between two or more circles. These results are obtained using Equation 4.4 by calculating the average value over all the images. In this step, the circle coordinates are predefined, and they are generated using a Matlab simulation. Therefore, we verified the detected circles from the CHT using these generated circles to obtain the true positives. The maximum difference between the centroids of the detected circles and the generated ones is 3 pixels. Under certain conditions, circle collisions can create a virtual circle center candidate for the CHT algorithm that potentially increases the number of false positives and false negatives. The logic resources are based on synthesis results using a Xilinx Virtex-4 FPGA.

Table 4.1: Number of votes, detection metrics (precision and recall) and required logic resources.

Votes per Pixel (N)	Prec.	Recall	Registers	LUTs	BRAM
8	95.47%	91.66%	396	509	57
16	98.98%	99.47%	409	522	66
32	100%	100%	419	567	66

The top-level block diagram of the CHT module is shown in Figure 4.15. The architecture is composed of three main units: the edge pixel buffer FIFO, calculation unit for generating the votes, and dual-port memory (DP-RAM) unit for the voting process. The edge pixel buffer FIFO unit stores the locations of active edge pixels. It consists of a counter and an FIFO, as illustrated in Figure 4.16-a. The counter is used to generate the locations of the edge filtered image. Only the coordinates of active pixels are written into the FIFO, which is composed of eighteen 18-kbit BRAMs. Based on simulations under worst case conditions, these FIFOs can buffer up to 16384 pixel locations. This results in location information for up to 125 robots, considering that only the active pixels are extracted from the robots. If there are more than 16384 unprocessed pixel locations, an FIFO overflow will occur. Additionally, the processing in the CHT module must be completed within a time frame of 512 × 512 pixel clocks (equivalent to the image size after downscaling in configuration C for each camera). This means, during this time, that all of the locations that are stored in the FIFO should already be read and converted to votes. Otherwise, the output from the CHT module will be corrupted. This situation is very unlikely to occur in a real scenario. However, if detected, a flag register is set, allowing further debugging. By decreasing the number of votes, this issue will be solved. Another solution is increasing the threshold in the edge filter, which reduces the number of active edge pixels. In this case, the detection accuracy could be decreased. Chapter 6 elaborates further on the accuracy when using different numbers of votes and robots.

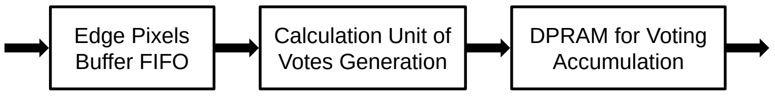

Edge Pixels Buffer FIFO
Calculation Unit of Votes Generation
DPRAM for Voting Accumulation

Figure 4.15: Top-level block diagram of FPGA-based CHT module.

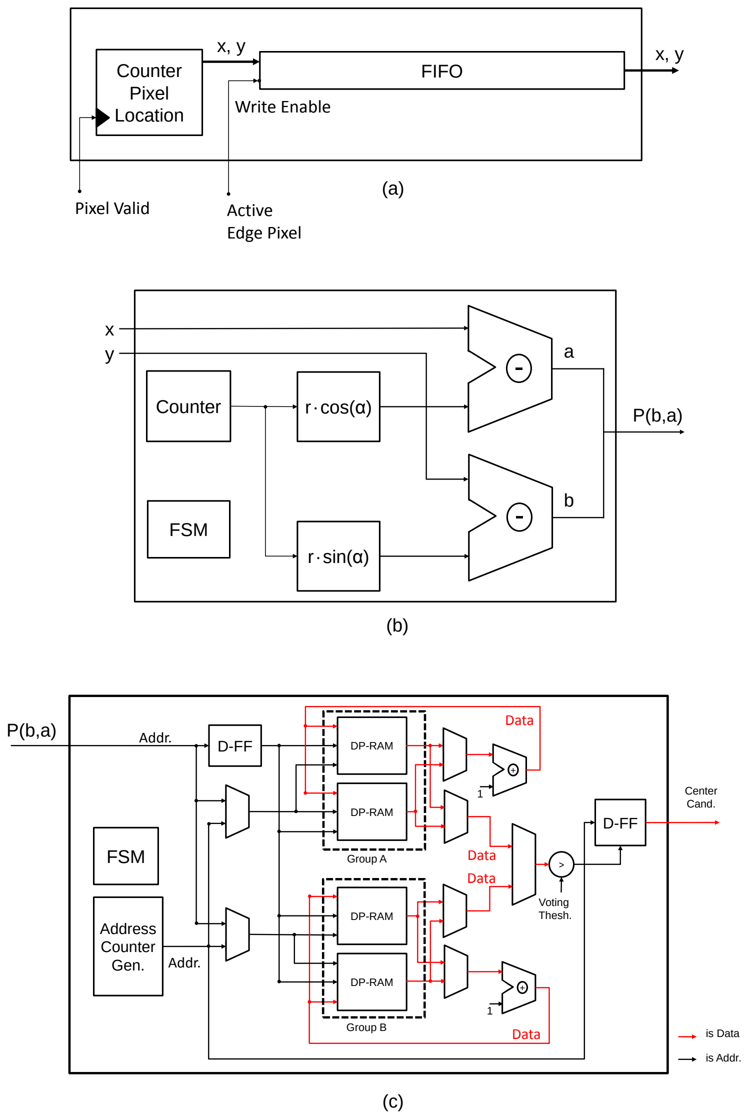


Figure 4.16: (a) Calculation unit of votes generator block diagram, (b) votes buffer FIFO, and (c) dual-port RAM unit for Hough voting process [62].

The structure of the calculation unit of the vote generator is depicted in Figure 4.16-b. It consists of two LUTs for realizing the sine and cosine operations multiplied by the predefined radius, a counter for selecting the LUT outputs, and two subtractors for calculating the $P(b,a)$ coordinates, which are used as offset address in the next step. N votes are generated sequentially for every location of active edge pixels. This means that N iterations are needed for N votes (e.g., 16 or 32). All of the processes are controlled by the finite-state-machine (FSM).

Finally, the output from the votes generator is sent to the DP-RAM unit to be accumulated based on the repetition of their values. The data at the generated offset address in the DP-RAM is incremented by 1. Utilizing a D-FF, this address is delayed for writing-back the incremented value to the DP-RAM, as shown in Figure 4.16-c. In this unit, for every pixel in the image after edge detection, there is a respective location in the DP-RAM that is used for the accumulated value of the votes. The votes with a final accumulation value higher than the predefined threshold ($voting_{th}$) are provided by the CHT module as the circle's center candidates.

Regarding the DP-RAM implementation, there are two issues to be considered for storing the votes. First, the size of the DP-RAM depends on the processed image size. Because the DP-RAM size is limited in FPGAs, there is a need to reduce the use of the internal memory in order to process high-resolution images. The second issue is that the design should be able to process the video stream in real time. This means that using an external memory is not an option because it significantly limits the throughput and increases the latency. Therefore, our objective is to find an architecture that minimizes the internal memory requirements.

The main reduction of the internal memory requirements in our approach is achieved by dividing the image into blocks of 32 rows each. This block size covers markers with a maximum diameter of 32 pixels, which is used in our application. All of the votes are handled by a double buffering method, which allows accumulations to be performed in a streaming approach. The votes of odd blocks are located in group A, while the votes from even blocks are located in group B. When accumulating one block of data in group A, the data in group B is erased and vice versa. For the accumulation, each vote is stored in two DP-RAMs of the same group. When erasing the data, only one of the two DP-RAMs in the same group is deleted, while the other one retains the data. This means that for both groups (A and B), at least one of their DP-RAMs stores updated values. This mechanism is useful for handling an overlap condition, where one or more circles are located between odd and even blocks. It is realized using four 16-kbyte DP-RAMs, which are divided into two groups, A and B, as depicted in Figure 4.16-c.

The two DP-RAM groups are controlled by a finite-state-machine, deciding which DP-RAM will be deleted, written, or read. The respective addresses for the DP-RAM are generated by the address counter generator. Only the accumulated votes with values greater than the predefined threshold ($voting_{th}$) will be considered for output as the circle's center candidates, as shown in Figure 4.16-c. Finally, the circle center candidate output is delivered to the graph clustering module to obtain the true circle center, which represents the location of the robot. Using the proposed approach reduces the total amount of required memory by a factor of four compared to storing the vote's accumulation values for the whole image. The total number of clock cycles required for the CHT module is ($width \times depth$) cycles.

4.3.4.2 Circle scanning window module

As an alternative method to find the circle center candidates, an FPGA-based CSW has been implemented. Basically, this method is similar to a convolution approach [11; 53]. As shown in Figure 4.17, a scanning window with its circle pattern pixels is used to find the circle center candidates. It maps many pixels of the binary image space to one point to find the circle center candidate. In our application, the circle radius in the robot marker is fixed. Thus, a specific size is used for the scanning window, with 13 × 13 pixels selected based on the size of the circles in the edge filtered image (after downscaling). This window consists of a circle pattern with a predefined radius. The calculation of the circle pattern coordinates is based on Equation 3.13. The CSW moves in the raster scan mode, scanning the entire image frame to find the circle center candidates. A location is considered to be a circle center candidate if the accumulated voting value in the scanning window block is higher than a selected threshold value. The voting value is calculated based on the accumulation of the binary pixels in the scanning window using the predefined circle pattern.

As shown in Figure 4.18, line buffers are needed to perform the raster scan window operation on the edge filtered image. RAM-based shift registers are utilized to build the line buffers. Twelve line buffers are needed for a 13 × 13 block in the scanning window operation. The scanning operation begins after 12 lines of the input frame are buffered. To detect the circle center candidates, 13 × 13 registers are used for temporarily storing the values of the buffered pixels. In our application, 32 of these registers are used as voting registers, which are arranged in a circle shape, as shown in Figure 4.18. The values of these voting registers are accumulated using the adder unit. The total voting value of the adder unit is compared with a certain threshold. If it is higher than the threshold, this means that a circle is detected, and its center is obtained from the address counter as a new circle center candidate, as shown in Figure 4.18. In our application, one circle usually has several circle center candidates. Finally, these circle center candidates are sent to the graph clustering module to obtain the true circle center, which represents the robot's location

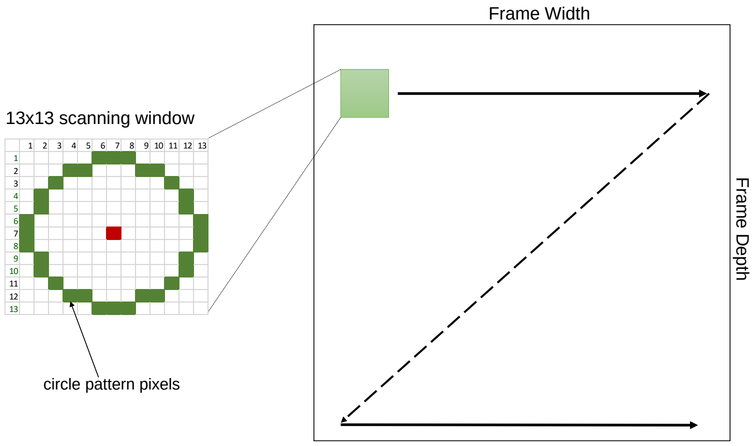


Figure 4.17: Raster scan with circle detection scanning window [62].

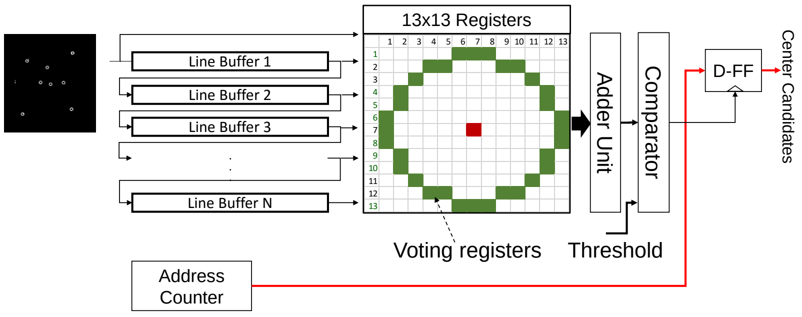


Figure 4.18: Top-level block diagram of the scanning window module [62].

4.3.4.3 Graph clustering module

In the proposed design, one of the main components used to implement the graph clustering in hardware is the distance calculator unit [46], which computes the distance between the previously determined candidates of circle centers. As illustrated in Figure 4.19, the output is a binary value. Here, 1 means that the candidates are connected and belong to the same cluster because the calculated distance d is smaller than the threshold value d_{th}. Binary distance 0 indicates that the corresponding candidates belong to different clusters.

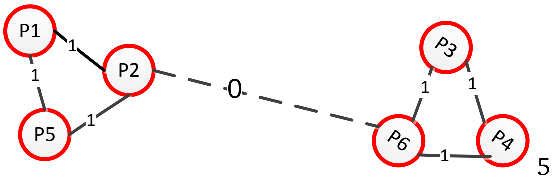


Figure 4.19: Binary distance in graph clustering.

A possible method for the distance calculation is using the Euclidean distance (Equation 4.5). Unfortunately, this requires additional resources (logic and DSP blocks) on the FPGA as a result of the square operations.

$$d = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2} \tag{4.5}$$

A resource-efficient solution is to use the Manhattan distance (Equation 4.6), which just requires the implementation of subtraction and addition in hardware.

$$d = |X_2 - X_1| + |Y_2 - Y_1| \tag{4.6}$$

Another alternative that requires even fewer logic resources is a multiplier-less binary distance calculation unit based on Chebyshev. It calculates the absolute magnitude of the differences between coordinates [46]. This binary distance calculation unit uses only subtraction, comparator, and logical *AND* operations, as shown in Equation 4.7.

$$if \;\; |X_2 - X_1| < d_{th} \;\; and \;\; |Y_2 - Y_1| < d_{th} \;\; bindist = 1 \; else \; bindist = 0 \tag{4.7}$$

Here, d_{th} is the threshold value for the maximum distance, and $bindist$ is the binary distance result.

For our application, each distance calculation method is simulated in software, and the results are listed in Table 4.2. In our simulation, we used our dataset, which is based on multi-circle detection in a multi-robot application. All of the distance calculations are in pixel units. As shown in Table 4.2, all of the methods provide sufficiently high performances (precision and recall). Under some conditions, when two or more robots collide, a virtual circle center candidate could be created that is either detected as a circle or shifts the location of a real circle, thus reducing the precision and recall values. The detection metrics are measured based on the calculated clusters for over 2000 images using Equation 4.4. All of the detected clusters are verified by detecting the pentagon shape inside the marker. If the pentagon is detected based on the calculated centroid of the clusters, this centroid is considered to be a correctly detected circle (true positive). Otherwise, it is not counted as a circle (false positive). As stated earlier, TP + FN in Equation 4.4 represents the total number of circles, which is predefined in our experiments. For the final implementation, we selected the modified Chebyshev method because it achieves the required accuracy with the lowest amount of logic resources. The threshold values d_{th} were selected based on our empirical simulation results.

Table 4.2: Simulation results of graph clustering using Euclidean, Manhattan and Chebyshev distance for 8 robots.

Dist.Thresh.	**Euclidean**		**Manhattan** ($d_{th}.\sqrt{2}$)		**Chebyshev**	
d_{th} (in pixels)	Pr.	Rec.	Pr.	Rec.	Pr.	Rec.
10	99.82	99.79	99.57	99.43	99.81	99.79
12	99.70	99.70	99.67	99.30	99.56	99.57
14	99.55	99.47	99.72	99.14	99.57	99.40
16	99.68	99.32	99.75	99.04	99.74	99.30
18	99.73	99.21	99.74	98.98	99.74	99.14

The initial version of the FPGA-based graph clustering module is designed based on the flow chart in Figure 3.22, which has previously been described in section 3.3.2.3. Its architecture block diagram is shown in Figure 4.20. The system consists of three main units: the circle center candidate registers, clustering unit, and centroid calculation unit. More details about this graph cluster module can be found in [61]. There are two main drawbacks in this design. First, based on the requirement of the algorithm, all of the circle center candidates must be stored in registers before the clustering can begin. A higher number of circle center candidates will take a larger amount of logic resources. As a consequence, the number of circle center candidates is limited to 256. Second, the design requires too many logic resources. This graph cluster module utilizes about 10600 slice registers and 12300 LUTs. Therefore, the architecture has been improved in the current updated version.

Unlike the initial version, which processed the clustering after all of the center candidates were completely collected, in the new implementation, the clustering is processed immediately when the graph clustering module receives a new circle candidate. Using this new approach, the number of candidates is not limited to 256, but can exceed 4096 circle center candidates in one frame. A block diagram of the FPGA-based graph clustering module and its flow chart are shown in Figures 4.21 and Figure 4.22, respectively. It is an updated version of our previous implementation, as discussed in [61].

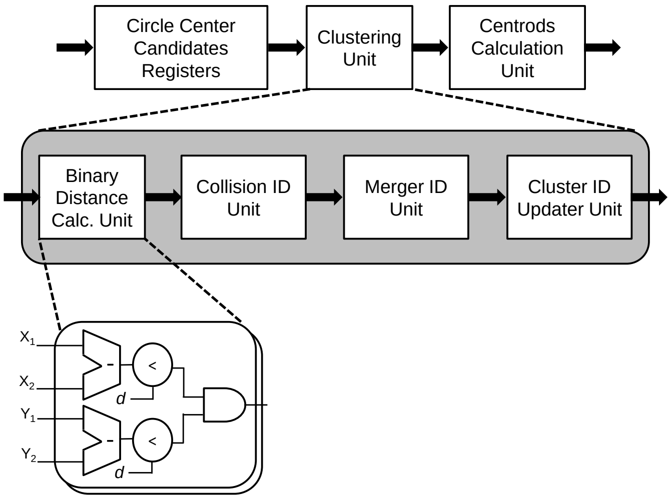

Circle Center Candidates Registers
Clustering Unit
Centrods Calculation Unit
Binary Distance Calc. Unit
Collision ID Unit
Merger ID Unit
Cluster ID Updater Unit
X1
X2
Y1
Y2
-
<
d

Figure 4.20: Block diagram of graph clustering module (limited number of circle center candidates) [61].

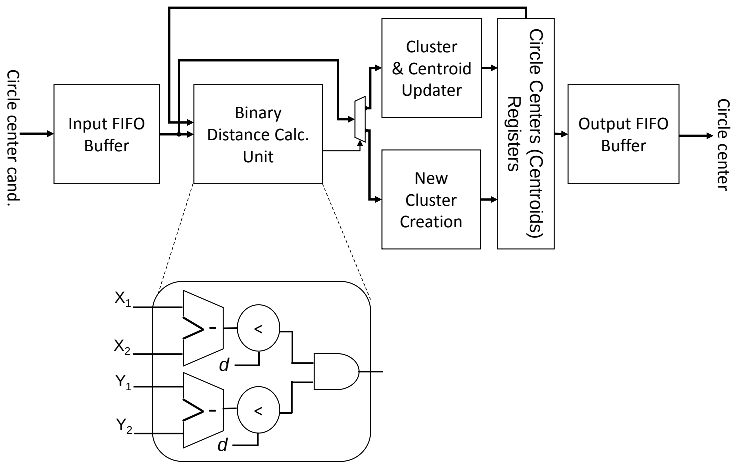


Figure 4.21: Block diagram of graph cluster module [62].

The graph clustering module consists of five main units: the input FIFO buffer, binary distance calculation (BDC) unit, cluster & centroid updater, new cluster creation, circle center (centroid) registers, and output FIFO buffer. The input FIFO is used to buffer the circle center candidates. This buffering is needed because a minimum of three clock cycles is required to process every candidate during the clustering operation.

The BDC, cluster & centroid updater, new cluster creation, and circle center (centroid) register units work together in the clustering operation. These units are fully pipelined so that incoming data can be processed every clock cycle. The BDC unit uses a multiplier-less distance calculation, based on Equation 4.7. Therefore, it simply requires a combination of a logical AND operator, subtractors, and comparators, as shown in Figure 4.21.

The functions of the other three units (cluster & centroid updater, new cluster creation, and circle center (centroid) registers) are explained using the flowchart depicted in Figure 4.22. As shown in this flow chart, the first circle center candidate will automatically be considered to be the first cluster and centroid. The next candidate from the input FIFO is processed in the BDC unit to find the connection between this new candidate and the existing clusters, as illustrated in Figure 4.23. A value of "1" at the BDC unit output means that the candidate is connected and belongs to one of the existing clusters. In this case, the centroid of the connected cluster is updated using the mean value of this circle center candidate and the centroid of the current cluster. Next, the module will read the new circle center candidate from the input FIFO for the subsequent clustering operation. Otherwise, if the output of the BDC unit is "0", the BDC calculations are repeated with the next existing clusters until the BDC output is "1", i.e., a connection with a cluster is found. If the last existing cluster is reached and no connection is found (BDC output is "0"), a new cluster is created. This process is repeated until the end of the frame. Finally, all of the centroids (circle centers) are transferred to the output FIFO. These circle centers represent the locations of the robots, which are sent to the host PC for further processing, including tracking and visualization.

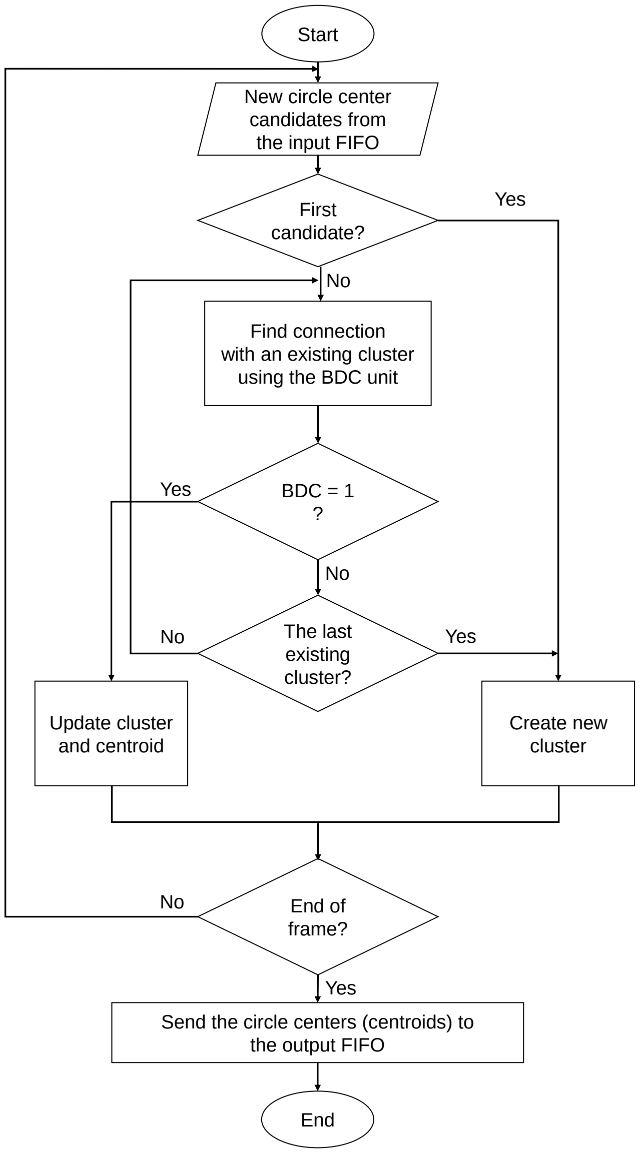

Start
New circle center candidates from the input FIFO
First candidate?
Yes
No
Find connection with an existing cluster using the BDC unit
BDC = 1 ?
Yes
No
The last existing cluster?
No
Yes
Update cluster and centroid
Create new cluster
End of frame?
No
Yes
Send the circle centers (centroids) to the output FIFO
End

Figure 4.22: Graph clustering flowchart in our application [62].

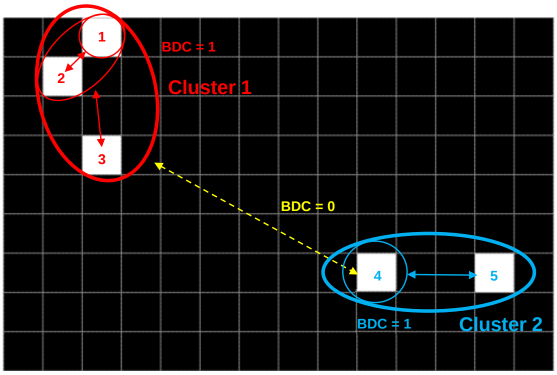


Figure 4.23: BDC unit and clustering operation [62].

4.4 Resource Utilization

This section evaluates the amount of logic resource utilization for the video processing modules. The resource utilization report is related to the number of logic units used in the FPGA implementation. As described in section 2.3.3, an FPGA consists of different logic units, including the FF, LUTs, RAM, and DSP. Using the resource utilization information, the required logic in the FPGA for each video processing module can be determined and analyzed. In particular, video processing modules that require a large or small number of resources can be identified. Based on this information, the scalability of the proposed design in an FPGA can be predicted.

Table 4.3, Table 4.4, and Table 4.5 list the device utilization values on a Virtex-4 FX100-11 for configurations A, B, and C, respectively. These configurations differ in the number of streaming hardware accelerators and thus in the parallelism of the implementation, as previously presented in section 4.2 and Figure 4.7. As shown in Table 4.3, the required logic resources of the video processing modules in configuration A, including the debayer, RGB to HSV color conversion, color mask, image scale down, CHT, and CSW, are relatively small compared to the graph cluster module. In configurations B and C, the designs use only one graph cluster and duplicate the other modules to increase the parallel processing. Accordingly, the required logic resources of the graph clustering module remain the same, while those of the other modules, which require relatively few logic resources, are doubled. Therefore, the total number of used logic resources, especially the numbers of FFs and LUTs, are not significantly different between configurations A and B. However, the total numbers of BRAMs and DSPs are doubled.

As shown in the tables, most of the BRAMs are used for the implementation of the CHT or CSW module. The CSW module requires significantly fewer logic resources than the CHT module. Therefore, the design that uses the CSW module is more scalable than the design with the CHT module. Table 4.5 shows that configuration C with the CHT module requires about 76% of total BRAMs in the Virtex-4 FPGA, while the same configuration with the CSW module uses only 19% of the total BRAMs. Because the base system accounts for 38% of the total BRAMs in the Virtex-4 FPGA, the design that uses the CHT module cannot be used to implement configuration C on a single Virtex-4 device. However, this is not an issue for the design with the CSW module. Therefore, using the CSW module is preferable.

Table 4.3: Device utilization: Virtex-4 FX100-11 (configuration A).

Module	**FFs**	**LUTs**	**BRAM**	**DSP48**
	1-bit	4-bit	18-Kbit	
(Max.)	(84352)	(84352)	(376)	(160)
Debayer	333	272	2	0
RGB-HSV-Col.Mask	180	352	0	4
Sobel Filter	389	303	2	6
Scale Down	25	53	0	0
CHT / CSW	419 / 430	567 / 511	66 / 12	0 / 0
Graph Cluster	3040	3163	4	0
Total	4386 / 4397	4710 / 4654	74 / 20	10 / 10
(%)	5.20 / 5.21	5.58 / 5.52	19.68 / 5.32	6.25 / 6.25

Table 4.4: Device utilization: Virtex-4 FX100-11 (configuration B).

Module	**FFs**	**LUTs**	**BRAM**	**DSP48**
	1-bit	4-bit	18-Kbit	
(Max.)	(84352)	(84352)	(376)	(160)
Debayer	666	546	4	0
RGB-HSV-Col.Mask	360	732	0	8
Sobel Filter	776	604	4	12
Scale Down	50	106	0	0
CHT / CSW	803 / 776	1129 / 912	132 / 24	0 / 0
BridgeToGraph	213	126	2	0
Graph Cluster	3040	3163	4	0
Total	5908 / 5881	6406 / 6189	146 / 38	20 / 20
(%)	7.00 / 6.97	7.59 / 7.34	38.83 / 10.11	12.5 / 12.5

Table 4.5: Device utilization: Virtex-4 FX100-11 (configuration C).

Module	**FFs**	**LUTs**	**BRAM**	**DSP48**
	1-bit	4-bit	18-Kbit	
(Max.)	(84352)	(84352)	(376)	(160)
Debayer	1324	1084	8	0
RGB-HSV-Col.Mask	720	1464	0	16
Sobel Filter	1552	1208	8	24
Scale Down	100	212	0	0
CHT / CSW	1605 / 1336	2237 / 1509	264 / 48	0 / 0
BridgeToGraph	405	272	4	0
Graph Cluster	3040	3163	4	0
Total	8746 / 8477	9640 / 8912	288 / 72	40 / 40
(%)	9.89 / 9.57	11.11 / 10.24	76.59 / 19.15	25.0 / 25.0

Regarding the scalability of the proposed design related to the number of cameras and the video resolution, the number of BRAMs increase significantly following the number of cameras and the video resolution. This is because the BRAMs are utilized as line buffers in the video processing modules. Meanwhile, increasing the number of robots will enlarge the numbers of FFs and LUTs, because they are used as registers and control units in the graph cluster module.

The remaining modules that are used in our implementation are the multi-camera frame grabber and base system. The multi-camera frame grabber occupies 3% of the total registers and 3% of the total LUTs of the Virtex-4 FPGA. The base system consists of the VFBC bridges, multi-port memory controller (MPMC), TEMAC, local bus (LB)-slave to native port interface (NPI) controller, PPC subsystem, and clock management. The base system accounts for 22% of the total registers, 25% of the total LUTs , and 38% of the total BRAMs of the Virtex-4 FPGA.

The IP cores can seamlessly be used in more recent FPGA technologies. As an example, they have been implemented on the Xilinx Virtex-6 XC6SX475T-2 and Xilinx Virtex-7 VX690T-2, and the respective results are presented in Table 4.6 and Table 4.7. In contrast to the Virtex-4 FPGA which is fabricated using 90 nm process technology, the Virtex-6 and Virtex-7 FPGA are built with newer process technologies. The Virtex-6 FPGA is built based on 40 nm, while the Virtex-7 is manufactured based on 28 nm process technology. All of the reported logic resources are post-place and route data. A maximum clock frequency of 160 MHz is achieved on the Virtex-4, whereas 185 MHz and 230 MHz are achieved on the Virtex-6 and Virtex-7, respectively. Because the newer FPGAs use 6-bit LUTs, while the Virtex-4 uses 4-bit LUTs, the tables show both values in order to provide a comparison. For conversion, it is assumed that a 6-bit LUT is equivalent to one and a half 4-bit LUTs.

Table 4.6: Device utilization: Virtex-6 XC6SX475T-2 (configuration A).

Module	**FFs**	**LUTs**	**LUTs**	**BRAM**	**DSP48**
	1-bit	6-bit	4-bit Eq	18/36-Kbit	
(Max.)	(595200)	(297600)		(2128/1064	(2016)
Debayer	327	246	369	2/0	0
RGB-HSV-Col.Mask	180	226	339	0/0	4
Sobel Filter	375	269	403	2/0	6
Scale Down	25	29	43	0/0	0
CHT	393	367	550	0/33	0
CSW	230	206	309	12/0	0
Graph Cluster	3027	1260	1890	4/0	0
Total with	4327	2397	3594	8/33	10
CHT	0.73%	0.81%	0.81%	0.38/3.10%	0.50%
Total with	4164	2236	3353	20/0	10
CSW	0.70%	0.75%	0.75%	0.94/0%	0.50%

Table 4.7: Device utilization: Virtex-7 VX690T-2 (configuration A).

Module	**FFs**	**LUTs**	**LUTs**	**BRAM**	**DSP48**
	1-bit	6-bit	4-bit Eq	18/36-Kbit	
(Max.)	(866400)	(433200)		(2940/1470	(3600)
Debayer	335	232	348	2/0	0
RGB-HSV-Col.Mask	180	236	354	0/0	4
Sobel Filter	375	259	388	2/0	6
Scale Down	25	29	43	0/0	0
CHT	392	397	595	0/33	0/0
CSW	230	209	313	12/0	0
Graph Cluster	3027	1286	1929	4/0	0
Total with	4334	2439	3657	20/33	10
CHT	0.50%	0.56%	0.56%	0.27/2.24%	0.28%
Total with	4172	2251	3375	20/0	10
CSW	0.48%	0.52%	0.52%	0.68/0%	0.28%

In addition to obtaining a faster maximum clock frequency, using the newer FPGA technology also means that more logic resources can be utilized. As shown in Table 4.6

and Table 4.7, the Xilinx Virtex-6 XC6SX475T-2 and Xilinx Virtex-7 VX690T-2 devices provide much higher numbers of logic resources compared to the Virtex-4 FX100. Therefore, in these tables, the numbers of used logic resources (e.g., FFs, LUTs, BRAMs, and DSP) are relatively very small compared to the maximum numbers of logic resources in the FPGA devices. Accordingly, greater parallelism and scalability can be supported using these newer FPGA devices.

4.5 Post Processing in Host PC

While previous sections presented the FPGA design for the implemented algorithms to detect the robots' locations, this section shows how the post-processing algorithm is implemented in the CPU. As previously described in section 3.3.3, the post-processing algorithm is performed on sub-images (cropped images). The output of the FPGA (robot markers' locations) is used to obtain all of the sub-images. Cropping operations are performed on a robot's marker coordinates, 40 × 40 pixels each, using an input image that is downloaded from the FPGA's external memory. Sequentially, all of the cropped images are stored in the CPU's memory. Finally, all of the sub-images are processed by the quad-core CPU in a multi-thread manner. The algorithm is used to process images at the frame rate of the cameras. Therefore, to obtain a frame rate of 30 fps, the processing time should be no more than 33 ms.

Figure 4.24 depicts the CPU's approach for handling the post-processing algorithm. A multi-thread method is applied to maximize the advantage of the multi-core architecture in the CPU. Four threads perform similar tasks to define the robots' orientations and IDs, executing the post-processing algorithms. The post-processing algorithm in the CPU utilizes the same method that is used in Teleworkbench [103], a vision-based multi-robot tracking system infrastructure that was developed by Congnitronic and Sensoric Research Group, CITEC, Bielefeld University.

Because it uses four threads for the computation of the robots' orientations and IDs, the CPU can simultaneously process four sub-images. Based on our experiment, a processing time of approximately 12 ms can be reached for 64 robots, while for one robot it takes only about 0.72 ms. This means that when working on a small image (40 × 40), the algorithm can be effectively performed in the CPU. Furthermore, this performance can potentially be increased when more threads are utilized for the post processing.

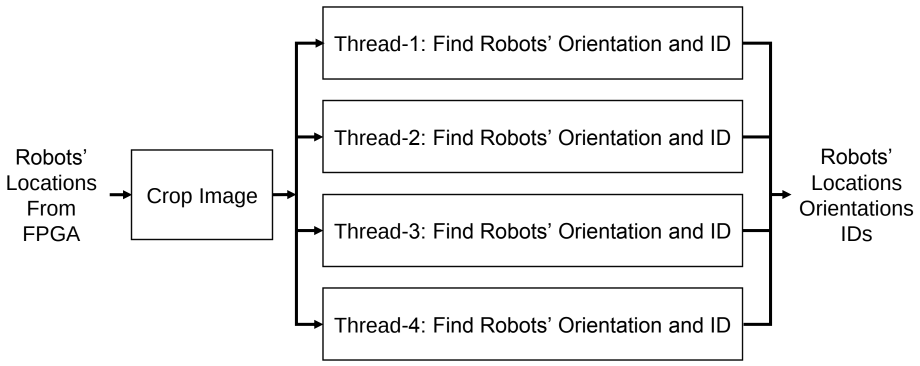

Robots' Locations From FPGA
Crop Image
Thread-1: Find Robots' Orientation and ID
Thread-2: Find Robots' Orientation and ID
Thread-3: Find Robots' Orientation and ID
Thread-4: Find Robots' Orientation and ID
Robots' Locations Orientations IDs

Figure 4.24: Top-level block diagram of multi-threads operation in CPU.

4.6 Summary

This chapter described the implementation of the FPGA-accelerated computing system for vision-based multi-robot tracking using multiple cameras. The system architecture and implementation of the video processing modules in the FPGA were presented. The customized design and high parallelism of the FPGA were used to provide various configurations that differed in the number of streaming hardware accelerators and thus in the parallelism of the implementation. This approach potentially increases the parallel processing when utilizing the video inputs from multiple cameras. Additionally, this chapter also presented two unique architectures for FPGA-based circle detection for multi-robot tracking, using the CHT-graph cluster algorithm and CSW technique-graph cluster algorithm combinations. Both designs support streaming operations, which are essential for multi-camera video processing in an FPGA. According to the resource utilization evaluation, the proposed design requires fewer logic resources and can be implemented in a single Virtex-4 FPGA. The design with the CSW module is preferable compared to the same design with the CHT because the CSW design requires fewer logic resources.

More details on the accuracy, performance, and efficiency of the proposed design can be found in chapter 6. Meanwhile, the same approach using a GPU as the hardware accelerator is presented in chapter 5.

5 Implementation in GPU-accelerated Heterogeneous Computing System

This chapter discusses the implementation of GPU-accelerated heterogeneous computing systems for vision-based multi-robot tracking. The proposed system is designed to improve the computation performance by utilizing the benefits of the GPU, particularly its massively parallel architecture and thousands of lightweight programmable cores. The discussion begins with descriptions of the proposed GPU-CPU hardware architectures, followed by a presentation of the algorithm and its implementation on a GPU using CUDA kernels. The implementation includes object segmentation (debayer, RGB to HSV color conversion, and color masking operations), edge filtering, and circle detection.

Figure 5.1 illustrates the architecture of the proposed vision-based multi-robot tracking platform using the GPU-CPU heterogeneous computing system. Improving the computational performance becomes the main aim of the proposed architecture. This design is the second alternative for a heterogeneous computing system for a multi-robot tracking platform, following the FPGA-CPU system that was presented in chapter 4. The GPU-based system basically uses a configuration similar to the FPGA-CPU design. For example, four GigE Vision cameras are used to cover the entire robot arena, robot markers are used to identify individual mobile robots, and a CPU is used for post-processing operations. The main difference compared to the FPGA-based system is the GPU-based system's use of an additional multi-gigabit Ethernet port PCIe card to access the cameras. In this design, most of the vision processing algorithm chains used to detect the locations of the robot markers are performed in the GPU. Three main video processing algorithms are implemented as GPU kernels, including the object segmentation, edge filtering, and object localization algorithms, as shown in Figure 5.1.

The following sections in this chapter further delineate the GPU implementation for vision-based robot tracking. It includes the GPU-CPU hardware environment description and algorithm implementation (GPU kernel).

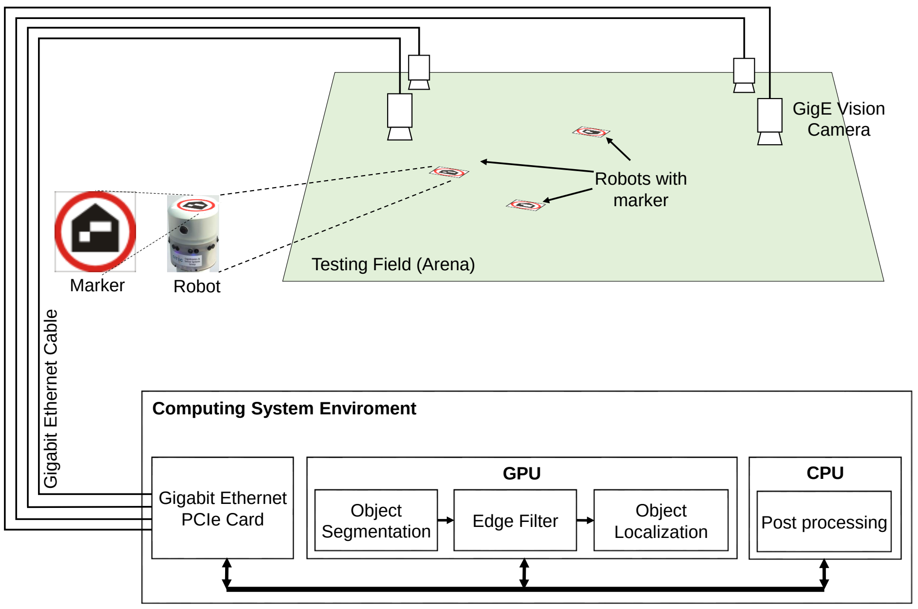


Figure 5.1: Top-level block diagram of GPU-CPU computing system for vision-based multi-robot tracking.

5.1 GPU-CPU Hardware Environment Description

Figure 5.2 depicts the complete scheme of the hardware environment of the proposed design. The peripheral component interconnect express (PCIe) works as the system bus and serves as the interconnection between the GPU and CPU, as well as the connection to the multi-gigabit Ethernet ports. Using the PCIe technology, the system has a high-speed serial computer expansion bus standard. This is essential because the data transfer speeds between the GPU, CPU, and multi-gigabit Ethernet ports rely on the expansion bus throughput. Additionally, an extra GPU device can easily be added to the system utilizing the PCIe-expansion slot.

The design utilizes an Intel i7 quad-core CPU as the processor in the host PC and two different GPUs as the hardware accelerators. Two GPUs with distinct architectures and fabrication process technologies are selected: the NVIDIA GTX 580 and GTX 780 GPUs. The GTX 780 GPU has a more up-to-date hardware architecture and fabrication process technology compared to the GTX 580. The GTX 780 GPU architecture is based on the

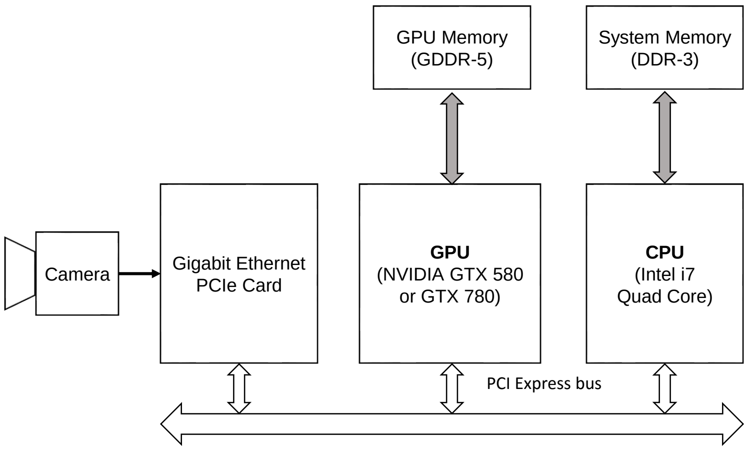


Figure 5.2: GPU-CPU hardware environtment block diagram.

NVIDIA's Kepler architecture and manufactured using the 28 nm fabrication process, while the GTX 580 GPU is designed based on NVIDIA's Fermi architecture and fabricated using the 40 nm fabrication process. Therefore, they are employed and evaluated individually to examine the characteristics of GPUs with different architectures and fabrication technologies.

Table 5.1 illustrates further differences between the GTX 580 GPU and GTX 780 GPU. The former comprises up to 512 CUDA cores, whereas the later consists of 2304 CUDA cores. Furthermore, the GTX 780 GPU has better specifications than the GTX 580 GPU. For example, the GTX-780 supports a higher clock rate, larger number of registers, larger memory bandwidth, and lower total power dissipation per CUDA core, as well as being furnished with a newer bus interface. The performance comparison between the two GPUs for vision-based robot tracking is examined in section 6.2.

Table 5.1: Comparison of GTX 580 and GTX 780 GPU.

GPU	GTX 580	GTX 780
Transistors	3 B	7.1 B
CUDA cores	512	2304
Streaming multiprocessor (SM)	16	12
Max warps per SM	48	64
Max threads per SM	1536	2048
Max register per Threads	63	255
Clock rate	772 MHz	863 MHz
Memory bandwidth	192 GB/s	288 GB/s
Bus interface	PCIe 2.0 × 16	PCIe 3.0 × 16
TDP	244 Watt	250 Watt

Figure 5.3 shows that the CPU and GPU have individual external memories, and both use their embedded on-chip memory controllers to access these memories. Accordingly, the system does not support the direct sharing of data between the GPU and CPU. The data must be copied from the GPU's external memory to the CPU's external memory or vice versa to enable data sharing, which means additional time is required to complete this procedure. Therefore, the data communication between the GPU and CPU should be minimized.

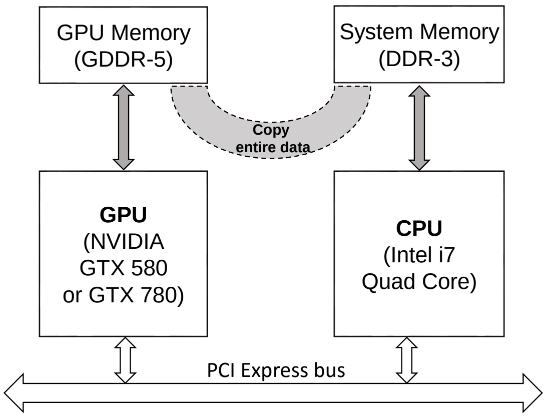


Figure 5.3: Data sharing between CPU and GPU.

5.2 Algorithm Implementation

Figure 5.4 shows the top-level block diagram of the algorithm for the vision-based multi-robot tracking application using the GPU-CPU heterogeneous computing systems. The GPU is used to execute computationally intensive tasks and huge data rates, while the CPU is utilized to perform complex and control operations with low data rates. It comprises four main steps, which are distributed between the GPU and CPU:

- Video frame grabber and image merging operations are performed in the CPU.
- Object localization, which consists of segmentation, Sobel edge filtering, and circle detection algorithms, are conducted on the GPU. These algorithms are performed on a video with a resolution of 2048×2048 pixels.
- A graph clustering algorithm is executed on the CPU.
- Post-processing operations, which consist of robot orientation and ID decoding algorithms, are completed on the CPU. These are performed on images with a resolution of 40×40.

In the first step, the CPU uses the multiple Gigabit Ethernet PCIe card to handle all the cameras. All of the captured video frames from the four cameras are buffered and merged into a single large frame. This is done using some OpenCV library functions. Afterward, this merged frame is copied from the host PC memory to the GPU memory for the next step.

In the GPU, the object segmentation, edge filtering, and circle detection algorithms are applied to the merged video frame. Two different algorithms are used to obtain the circle center candidates from the robot markers. These are the CHT algorithm and CSW technique. Furthermore, as shown in Figure 5.4, two different vision processing chains exist in the GPU. The first processes the original size of the input video frame (segmented image), and the other uses the Resize function to downscale the frame size. This downscaling approach aims to reduce the execution time. A factor of two in the x and y directions is chosen because this reduction size can still maintain a good recognition of the circle. An examination of the trade-off between the accuracy and the computation performance (execution time) will be performed for the above different configurations (CHT, CSW, downscaling).

The output data from the object localization algorithm are no longer image based, but the location candidates of the robots. To obtain the true circle centers, the CPU uses the following processes. In this regard, the host PC copies the GPU output to complete the object localization process. A graph clustering algorithm is applied to define the

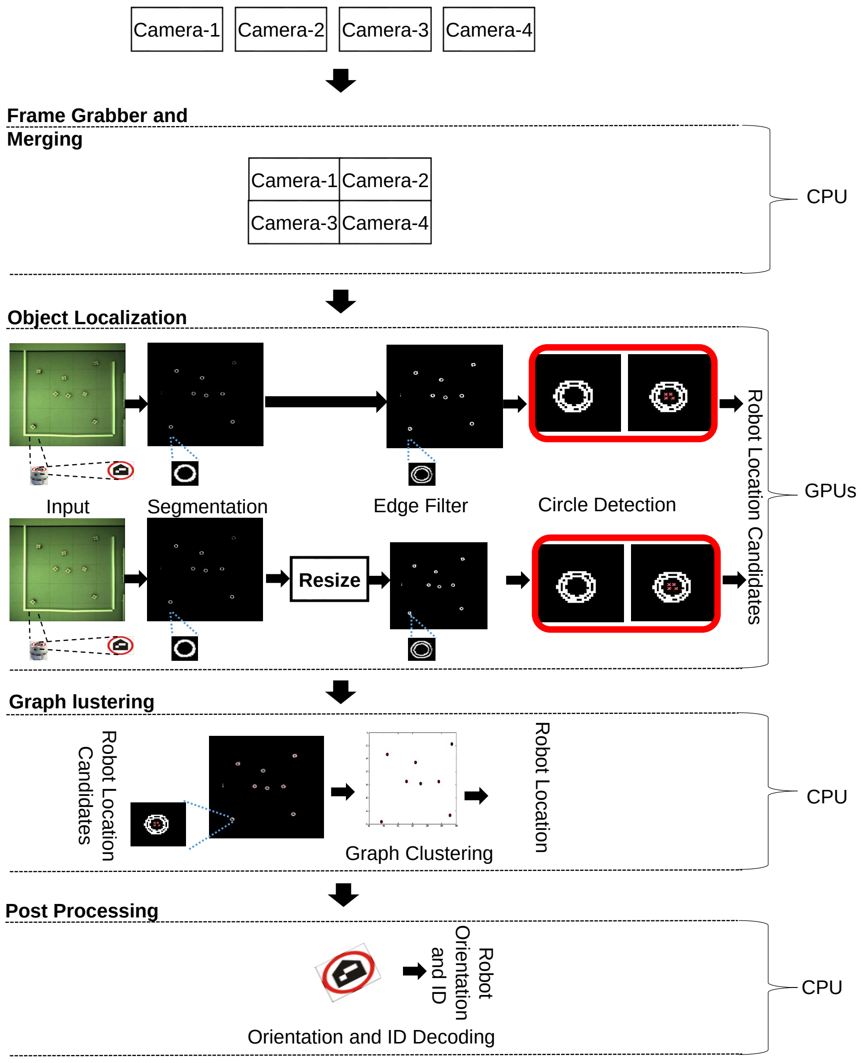


Figure 5.4: Algorithm of vision-based robot tracking in GPU-CPU.

true circle centers, which represent the locations of the robots. In this implementation, the graph clustering algorithm is performed outside the GPU because of its limited data parallelism. Yet, it requires more branch and control operations. Therefore, this algorithm is implemented in the CPU. Finally, after obtaining the coordinates of the robots, additional information related to each robot, including its orientation and ID, is calculated in the post-processing step. To minimize the computational requirements in the CPU, the calculations of the robot orientations and IDs are performed based on a 40×40 pixel image of each robot marker.

5.3 CUDA Kernels Implementation

This section describes the detailed implementations of the proposed algorithms in the GPU step. As illustrated in Figure 5.5, the algorithm implementation consists of three main processes: object segmentation, which includes the debayer, RGB to HSV color conversion, and color mask algorithms; edge detection filtering with the Sobel filter algorithm; and object localization using the circle detection algorithm. Because some of the algorithms are not available in the OpenCV GPU (e.g., the debayer, color mask, and CSW), all of the algorithm implementations are fully written as specific GPU kernel codes. This approach will also make the data transfer among kernels straightforward and easily defined. The following subsections explore the algorithm implementations in the GPU kernels in more detail.

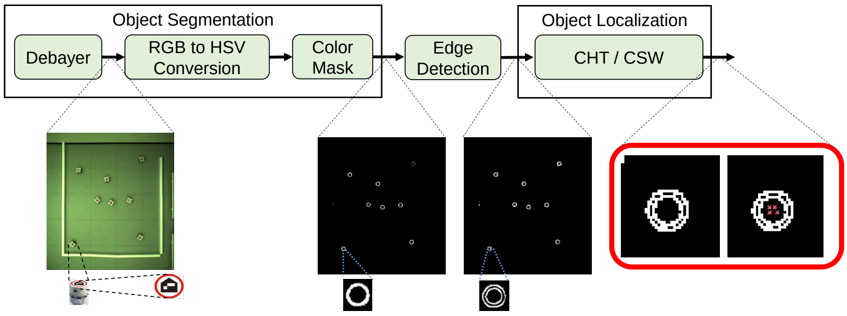


Figure 5.5: Top level block diagram of GPU algorithm implementation.

5.3.1 Object Segmentation

In the CUDA-supported GPU, an SIMT execution model is used to handle and execute many threads. From the hardware perspective, each thread is executed in a CUDA core. Meanwhile, a thread block is performed in a stream multiprocessor (SM). When a kernel is launched, a grid consisting of numerous thread blocks is scheduled to implement an algorithm. Figure 5.6 illustrates the implementation of the object segmentation algorithm from a hardware perspective. A kernel grid consisting of numerous thread blocks is used to implement the object segmentation algorithm, which extracts the circles of all the robot markers using a color segmentation method. The algorithm includes the debayer, RGB to HSV color conversion, and color mask.

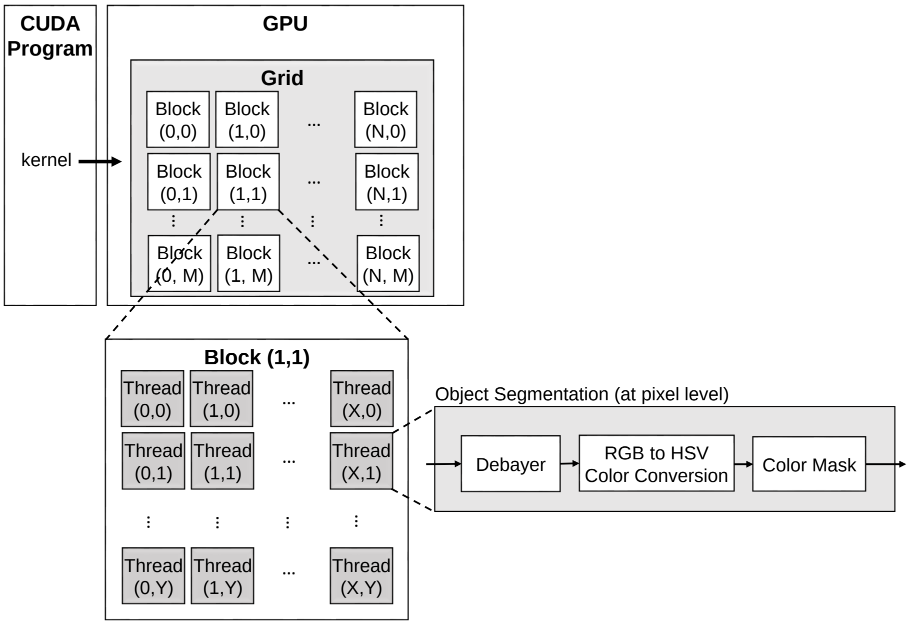


Figure 5.6: Top-level block diagram of GPU object segmentation implementation.

Furthermore, Figure 5.6 shows that the object segmentation algorithm is executed at the pixel level. The pixel processing operations run sequentially from the debayer algorithm to the RGB to HSV color conversion algorithm, and finally to the color mask algorithm. In the GPU, these operations can be efficiently processed by employing its hundreds or even thousands of cores. This is thanks to the SIMT approach, which simultaneously executes the operations using its many cores. Additionally, the CUDA platform manages the scheduling of all the threads to be executed in a concurrent processing manner.

Listing 5.1 shows the pseudocode of the CUDA kernel for the object segmentation algorithm. The debayer algorithm produces a full RGB color image out of a Bayer-encoded image. In this work, a bilinear interpolation is implemented to interpolate the missing color components to retrieve an RGB value for each pixel. It generates the values of the missing color's components by calculating the average of eight neighborhood pixels. The RGB to HSV algorithm is designed based on the algorithm of Foley et al. [41], which has been previously described in section 3.3.1.2. This color conversion attempts to utilize the robustness of the HSV color space in terms of illumination or lighting changes, as has been proven in some previous experiments [6; 78]. Finally, a color mask algorithm is used to mask off all the colors except a specific color range based on the values of the HSV color image. In this case, it will be the HSV equivalent values for the red color in the robot marker.

Listing 5.1: Pseudocode for object segmentation CUDA kernel

```
__global__ void bayerGR_to_rgb_to_hsv_to_colourmask (
unsigned char  BayerIn, gpu::PtrStepb MaskOut,
int Width, int Height)
{
// G1 R
// B  G2

int xIndex = 2   ((blockIdx.x blockDim.x) + threadIdx.x);
int yIndex = 2   ((blockIdx.y blockDim.y) + threadIdx.y);

if ((yIndex < Height) && (xIndex < Width))
   {
        Calculate_Debayer_G1(BayerIn,rgb_G1);
        Calculate_RGB_to_HSV(rgb_G1,hsv_G1);
        Calculate_HSV_to_ColorMask(hsv_G1,MaskOut);

        Calculate_Debayer_R(BayerIn,rgb_R);
        Calculate_RGB_to_HSV(rgb_R,hsv_R);
        Calculate_HSV_to_ColorMask(hsv_R,MaskOut);

        Calculate_Debayer_B(BayerIn,rgb_B);
        Calculate_RGB_to_HSV(rgb_B,hsv_B);
        Calculate_HSV_to_ColorMask(hsv_B,MaskOut);

        Calculate_Debayer_G2(BayerIn,rgb_G2);
        Calculate_RGB_to_HSV(rgb_G2,hsv_G2);
        Calculate_HSV_to_ColorMask(hsv_G2,MaskOut);
   }
}
```

5.3.2 Edge Filter

Figure 5.7 illustrates the implementation of the Sobel filter in the GPU, and its pseudocode is depicted in Listing 5.2. An edge detection filter is implemented to pass only the edges of HSV color masked objects. It is based on the Sobel filter algorithm. The Sobel filter operation is running sequentially at each thread level, while the CUDA platform schedules the operation for all the threads at the GPU device level. The shared memory is utilized as a buffer for the color masked input image, which can reduce the latency between the threads and global memory, as well as to increase the data transfer speed of the computing operations. The edge filtered image is taken from the gradient magnitude at each pixel in the image in both the horizontal (Gx) and vertical (Gy) directions. This involves a 3 × 3 kernel matrix convolution operation with the input image to compute the approximations of the gradients. The Gx and Gy gradients are combined to acquire the total gradient (G). Finally, the gradient G is compared to a threshold value to define the output image.

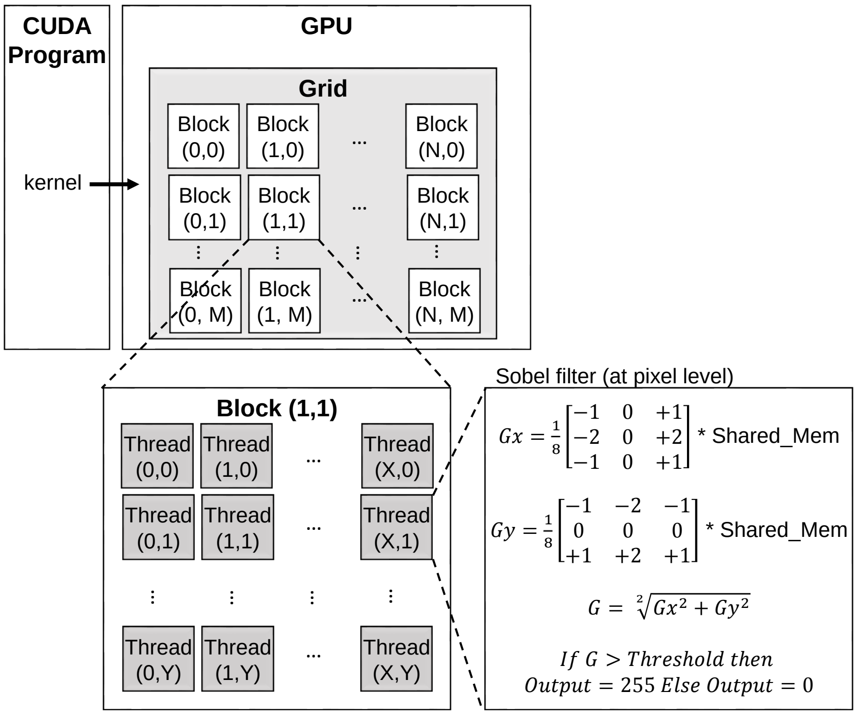

CUDA Program
kernel
GPU
Grid
Block (0,0)
Block (1,0)
Block (N,0)
Block (0,1)
Block (1,1)
Block (N,1)
Block (0, M)
Block (1, M)
Block (N, M)
Block (1,1)
Thread (0,0)
Thread (1,0)
Thread (X,0)
Thread (0,1)
Thread (1,1)
Thread (X,1)
Thread (0,Y)
Thread (1,Y)
Thread (X,Y)
Sobel filter (at pixel level)
$Gx = \frac{1}{8}\begin{bmatrix}-1 & 0 & +1\\ -2 & 0 & +2\\ -1 & 0 & +1\end{bmatrix}$ * Shared_Mem
$Gy = \frac{1}{8}\begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ +1 & +2 & +1\end{bmatrix}$ * Shared_Mem
$G = \sqrt[2]{Gx^2 + Gy^2}$
$If\ G > Threshold\ then$
$Output = 255\ Else\ Output = 0$

Figure 5.7: Top-level block diagram of GPU Sobel filter implementation.

In this GPU kernel, two different approaches are applied to obtain the total gradient (G). They are a standard approach that uses Pythagoras' theorem $G = \sqrt[2]{Gx^2 + Gy^2}$ and an approximate approach, which utilizes an absolute add operation $G = |Gx| + |Gy|$.

However, only the latter approach is used in the FPGA implementation, as previously presented in chapter 4.

Listing 5.2: Pseudocode for Sobel edge filter CUDA kernel.

```
__global__  void Sobel_kernel(
gpu::PtrStepSz<unsigned char> MaskImgInput, cv::gpu::PtrStepb Output,
const int Width, const int Height, const size_t Threshold)

__shared__ unsigned char shared_mem[Size_Y][Size_X];

int xIndex = blockIdx.x blockDim.x + threadIdx.x;
int yIndex = blockIdx.y blockDim.y + threadIdx.y;

if ((yIndex < Height) && (xIndex < Width))
{
// copy input to shared memory
shared_mem = MaskImgInput.ptr(yIndex)[xIndex]

//wait for all threads to finish read
__syncthreads();

// Calculate gradien using matrix convolution
Calculate_Gx(shared_mem,Gx);
Calculate_Gy(shared_mem,Gy);

// Combine Gradient
Calculate_G(Gx,Gy);

// Threshold operation
If G > Threshold Output = 255 else Output = 0;

}
}
```

5.3.3 Object Localization

In the GPU-accelerated computing system, similar to the FPGA implementation in chapter 4, the circular HT-graph cluster algorithm and scanning window-graph cluster algorithm are implemented to detect the circles of the robot markers, which represent the locations of the robots. The GPU uses the circular HT and scanning window algorithms to generate the circle center candidates, while CPU processes the graph clustering algorithm to define the true circle centers. In this implementation, the graph clustering algorithm is performed outside the GPU because of its limited data parallelism, yet its requirement of more branch and control operations. Therefore, this algorithm is implemented in the CPU.

5.3.3.1 Circular Hough transform in GPU

Figures 5.8 and 5.9 show the block diagram of the CHT algorithm implementation in the GPU, whereas Listing 5.3 represents the algorithm's pseudocode. The algorithm is based on Equation 3.12, which is also used in the FPGA implementation. The CHT algorithm implementation consists of two steps. First, the algorithm generates votes from every active pixel and accumulates the voting values. Second, the process continuous by thresholding the accumulated result and storing the circle center candidates. These two steps are illustrated in Figures 5.8 and 5.9.

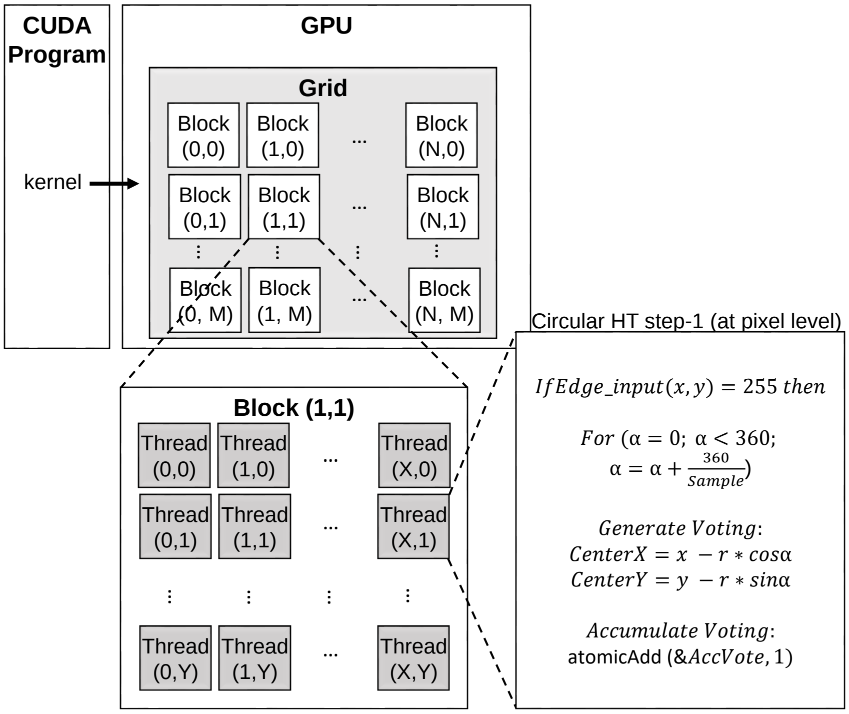


Figure 5.8: Top level block diagram of GPU circular HT implementation.

In the first step of this GPU implementation, two different sampling value numbers (16 and 32) for α-degree (in Equation 3.12) are used to sweep the full 360°. This sampling represents the CHT votes that are generated by the GPU's core from every active pixel (value = 255). Because these pixels are mostly extracted from the circles of the robot markers, they indicate that the number of robot markers likely influences the execution time in the GPU. Sequentially, all of the generated votes are accumulated.

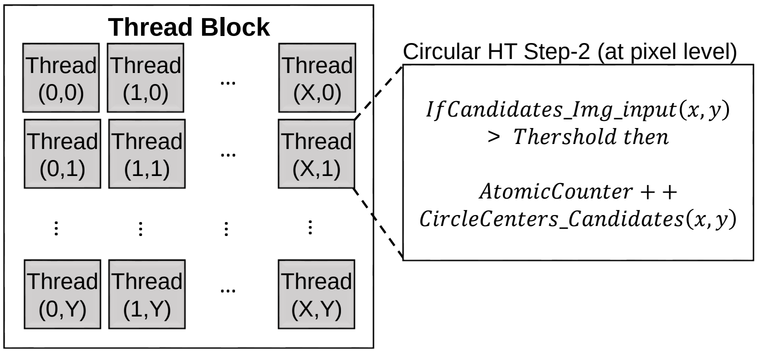


Figure 5.9: GPU circular HT implementation (step-2).

The voting accumulations emerge based on the repetition of the generated vote values (in coordinate $(CenterX, centerY)$ format). In the second step, the accumulation results are being processed. If an accumulation value $AccVote(CenterX, centerY)$ is higher than the selected threshold value, the coordinates $(CenterX, centerY)$ are defined and counted as a circle center candidate. Based on empirical experiments, the threshold value for generating the circle center candidates is within at least 62.5% of the vote-sampling value (e.g., if the CHT vote sampling = 16, the threshold = 10). Finally, all of the circle center candidates and their number are sent to the CPU.

In contrast with the FPGA implementation, where increasing the sampling number implies an increase in the required logic resources, the GPU implementation is relatively flexible, which means that it is free to determine the sampling number in the GPU. However, the sampling number will affect the execution time in the GPU. In this regard, a GPU core needs more iterations to produce a higher sampling number, which results in a longer execution time. Section 6.2.2 further delineates the computation performance evaluation.

Listing 5.3: Pseudocode for circular HT CUDA kernel.

```
__global__ void CHT_kernel_step1 (
gpu::PtrStepb Edge_input, unsigned int  AccVote,
int Width, int Height, int VoteTheshold)
{
   int xIndex = blockIdx.x blockDim.x + threadIdx.x;
   int yIndex = blockIdx.y blockDim.y + threadIdx.y;

   if ((yIndex < Height) && (xIndex < Width))
      {
         if ((Edge_input(xIndex,yIndex) == 255)
            {
               for (float alpha = 0; alpha < 360; alpha += 360/sampling)
                  {
                  // Generate voting
                     CenterX = xIndex - r cos(alpha);
                     CenterY = yIndex - r sin(alpha);

                  // Accumulate voting
                     atomicAdd(&AccVote[CenterX+CenterY width], 1);
                   }
              }
      }
}


__global__ void CHT_kernel_step2(
unsigned int  acc_input, gpu::PtrStepb CircleCenters_Candidates,
unsigned int  counter, int Width, int Height)
{
int xIndex = blockIdx.x blockDim.x + threadIdx.x;
int yIndex = blockIdx.y blockDim.y + threadIdx.y;
long int xy_index = yIndex width + xIndex;

if ((yIndex < Height) && (xIndex < Width))
   {

      // reset atomic counter
      if ((xIndex == 0) && (yIndex == 0) ) atomicAnd(counter, 0);;

      if ((acc_input(xIndex,yIndex) == 255)
         {
         // increment counter
            AtomicCounter =     atomicAdd(counter, 1);

         // Save Circle Center Candidates
            CircleCenters_Candidates[AtomicCounter].centerX = xIndex;
            CircleCenters_Candidates[AtomicCounter].centerY = yIndex;
         }
   }
}
```

5.3.3.2 Circle scanning window in GPU

The main concept of implementing the scanning window technique in the GPU is similar to the approach utilized in the FPGA implementation, as shown in Figure 4.17 (in the previous chapter). The CSW technique follows the many-to-one approach. To find the circle center candidates, it maps many pixels of the binary image space into one point. Compared to the CHT method, which takes the voting values from the points in the transformed space, the CSW method directly obtains the voting values from the binary image (edge filtered image).

In the context of the relationship between the execution time, which is analogous to the number of iterations, and the number of robot markers, the CHT and CSW kernels represent different characteristics. Unlike in the former, where the execution time depends on the number of robot markers and samplings, the latter shows that the number of robots markers (active pixels) does not affect the execution time. Yet, the size of the input image and samplings determine the duration of the process. More detailed explanations of the performance evaluations of the CHT and CSW kernels are presented in section 6.2.2.

The implementation of the CSW technique in the FPGA uses numerous line buffers, controllers, and logic resources, whereas the GPU relies on the program execution, employing hundreds or even thousands of threads, which run in a concurrent manner. The CSW implementation uses a circle pattern with a predefined radius. It moves in the raster scan mode, scanning the entire image frame to find the circle center candidates.

Figure 5.10 illustrates the implementation of the scanning window technique in the GPU, and Listing 5.4 represents its pseudo code. In the GPU implementation, the circle pattern emerges by storing two arrays of coordinate numbers in the GPU's registers (shown as Xc and Yc arrays in Listing 5.4). These are calculated based on Equation 3.13. The voting accumulation is performed in the cores of many GPUs and executed in the SIMT style for all the pixels of the binary image (active and inactive pixels). Each thread requires N sampling number (e.g., 16, or 32) iterations to point out pixels. The voting accumulation is incremented by one (1) if the pointed pixel's value is an active edge pixel. A coordinate is considered to be a circle center candidate when the voting value for this coordinate is higher than the selected threshold value. The threshold value has been selected based on empirical experiments, and at least represents 62.5% of the vote-sampling value.

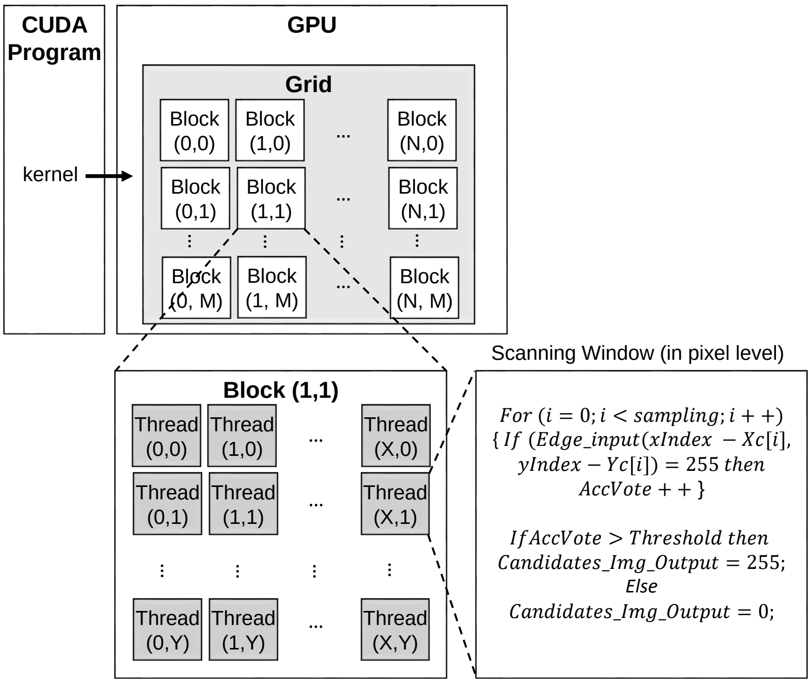


Figure 5.10: Top-level block diagram of scanning window implementation in GPU.

Listing 5.4: Pseudocode for scanning window CUDA kernel.

```
__global__ void ScanningWindow_kernel (
gpu::PtrStepb Edge_input, gpu::PtrStepb Candidates_Img_Output,
int Width, int Height, int VoteTheshold)
{

//// 16 Sampling Circle Pattern
int Xc[16] = { 12, 9, 5, 0, −5, −9, −12, −13, −12, −9, −5, 0, 5, 9, 12, 13 };
int Yc[16] = { 5, 9, 12, 13, 12, 9, 5, 0, −5, −9, −12, −13, −12, −9, −5, 0 };

int xIndex = blockIdx.x blockDim.x + threadIdx.x;
int yIndex = blockIdx.y blockDim.y + threadIdx.y;

if ((yIndex < Height) && (xIndex < Width))
   {

   // Accumulate voting
   for (int i = 0; i < sampling; i++)
      {
         if ((Edge_input(xIndex − Xc, yIndex − Yc) == 255) then
            {
            // Accumulate voting
               AccVote(CenterX, CenterY) ++;
            }
      }


// Mark Cicrcle Center Candidates
if (VotingValue > VoteTheshold)
   {
      Candidates_Img_Output(CenterX, CenterY)=255;
   }
   else
   {
      Candidates_Img_Output(CenterX, CenterY)=0;
   }

}
```

Finally, the circle center candidates are sent to the graph clustering module to obtain the true circle centers, which represent the robots' locations. To obtain the true circle centers, the host PC copies the GPU output to complete the object localization process by applying the graph cluster algorithm.

5.4 Achieved Occupancy

As previously described in section 2.3.2.1 and section 2.3.2.2, a kernel program is executed using the SIMT method and processed in a group fashion. The SM of the GPU determines the threads in groups of 32 parallel threads called warps. The SM resources such as registers and shared memories are limited and shared among warps and blocks. Therefore, not all of the warps are active on an SM. The ratio of active warps to the total number of available warps is called the occupancy. Theoretical occupancy reveals the upper bound active warps on an SM while achieved occupancy shows the true number of active warps varies over the duration of the kernel, as warps begin and end [85]. A higher warp occupancy means a better utilization of a GPU's computation resources [28]. Of course, this occupancy metric is not the only method to measure the effectiveness of a kernel that can be implemented in a GPU. However, it is the one that is provided by the NVIDIA CUDA tool for examining a kernel's resource occupancy.

This section provides an evaluation of the occupancy profiling report for the video processing kernels, which have been explained in the previous sections. Table 5.2 and Table 5.3 detail the achieved occupancy for each kernel running on the GTX-580 and

GTX-780 GPUs, respectively. Table 5.2 shows that the achieved occupancies on the GTX-580, particularly for the object segmentation, Sobel filter, and CHT kernels, are limited by the number of registers in its SMs. However, for the GTX-780 GPU, most of the kernels (e.g., object segmentation, Sobel filter, resize (downscaling), and CSW) have a high occupancy. Only the CHT kernel has a low occupancy, which is limited by the number of registers. The GTX-780 GPU has better occupancy results than the GTX-580 because it has a larger number of registers, as previously shown in Table 5.1.

Table 5.2: Occupancy of kernels on GPU GTX-580.

Kernel	**Achieved**	**Theoretical**	**Limiter**
Object Segmentation	49.8%	50%	Registers
Resize (Downscaling)	82.6%	83.3%	-
Edge Filter (Sobel)	49.7%	50%	Registers
CHT	31.7%	33.3%	Registers
CSW	83.1%	83.3%	-

Table 5.3: Occupancy of kernels on GPU GTX-780.

Kernel	**Achieved**	**Theoretical**	**Limiter**
Object Segmentation	74.3%	75%	-
Resize (Downscaling)	98.6%	100%	-
Edge Filter (Sobel)	74.1%	75%	-
CHT	47.9%	50%	Registers
CSW	99.1%	100%	-

5.5 Post Processing in Host PC

In the context of the heterogeneous computing system, the implementations of video processing algorithms in the GPU have been discussed in detail in the previous sections. Therefore, this section presents the implementation of the post-processing algorithm in the CPU. While the GPU performs the computationally intensive tasks on the full-resolution image, the CPU executes the post-processing algorithm on sub-images (cropped images). The output of the GPU (robot markers' location candidates) is used to obtain all of these sub-images.

Figure 5.11 depicts the block diagram of the CPU's implementation for the post-processing algorithm. As previously described in section 5.3.3, the graph clustering operation used to obtain the robots' locations is performed in the CPU. The host PC copies the robot markers' location candidates in the GPU's memory and sequentially processes these data using the graph cluster algorithm to obtain the robots' locations. These locations are used to obtain all of the sub-images. This is done by applying a cropping operation at every location of the robot. Sequentially, all of the cropped images (40 × 40 pixels) are buffered in the CPU's memory for further processing.

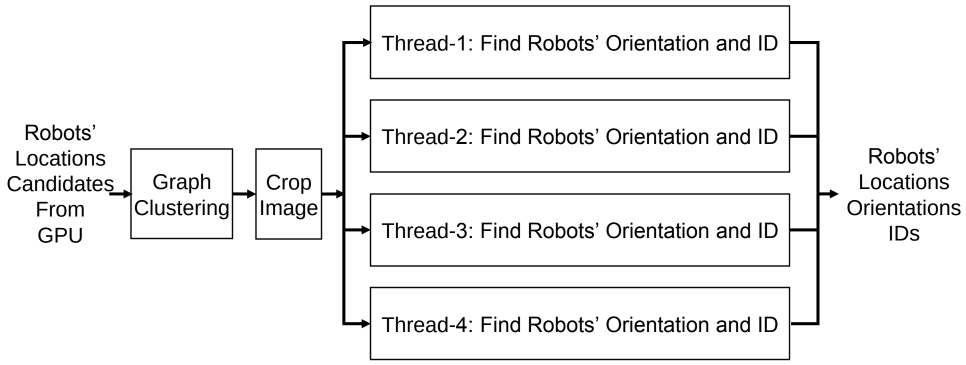

Robots' Locations Candidates From GPU
Graph Clustering
Crop Image
Thread-1: Find Robots' Orientation and ID
Thread-2: Find Robots' Orientation and ID
Thread-3: Find Robots' Orientation and ID
Thread-4: Find Robots' Orientation and ID
Robots' Locations Orientations IDs

Figure 5.11: Top-level block diagram of multi-threads operation in CPU.

To obtain the robots' orientations and IDs, the same approach used in the FPGA-CPU heterogeneous system is applied in the GPU-based system. The advantage of the multi-core architecture in the CPU is employed for processing all the sub-images in a multi-thread approach. All of the images are distributed into four threads, where every thread executes the same algorithm. Finally, the CPU simultaneously processes the Find Robot's Orientation and ID algorithm. This algorithm is used to process images at the frame rate of the cameras. Therefore, to obtain a frame rate of 30 fps, the processing time should be not more than 33 ms. Based on our experiment, including the graph clustering operation, a processing time of approximately 14 ms can be reached for 64 robots, while for one robot it takes only about 0.74 ms.

5.6 Summary

The implementation of a GPU-accelerated computing system for vision-based multi-robot tracking has been presented in this chapter. The GPU is used to process computationally intensive tasks in the algorithms used for detecting the robot marker locations, while the CPU is used for the post processing. Using the SIMT approach, three main video processing algorithms are implemented as GPU kernels, including object segmentation (debayer, RGB to HSV color conversion, and color masking operations), edge filtering, and object localization.

More details on the accuracy, performance, and efficiency of the proposed design, as well as a comparison between the FPGA-and GPU-based systems, are presented in chapter 6.

6 Results and Analysis

The previous chapters comprehensively presented the architecture and implementation of the FPGA- and GPU-accelerated heterogeneous computing systems for vision-based multi-robot tracking. Our method aims to off-load the computation-intensive tasks from the host computer utilizing an FPGA-based or a GPU-based hardware accelerator. Meanwhile, tasks with less data parallelism and more branch-control operations are completed in the CPU. The main objectives of these proposed designs are improving the computation performance.

Therefore, this chapter discusses the experimental results and analysis of the implemented FPGA- and GPU-accelerated computing systems. It aims to find the most advantageous design because the experiments and examinations focus on the detection performance, computing performance, and power consumption. Based on the examinations and analysis, this thesis proposes a preferable design for a vision-based multi-robot tracking computing system.

6.1 Detection Performance

In the context of a vision-based multi-robot tracking computing system, the detection performance refers to the ability of the system to accurately detect the locations of all the robots. In this regard, a detection test is essential in the implementation of FPGA-CPU- and GPU-CPU-based computing systems, in order to verify the functionality of the proposed algorithm and the architecture implementation.

In this work, precision and recall metrics [44] are used to investigate the detection performance. They are calculated according to the correctly detected circles (true positive) of the robot markers in all of the processed frames of our recorded video data sets. The precision and recall results are obtained by calculating the average value over all the images in the data sets (N_{Frame}). Each data set contains about 5000 frames, and the total number of mobile robots with their markers varies from 4 to 64.

Basically, the same metrics presented in Equation 4.4 are used to obtain the detection performance. Equations 6.1 and 6.2 show the formulas for calculating the precision and recall, respectively. The precision metric is obtained based on the ratio between the number of correctly detected robots (TP) and the total number of detected robots (correctly detected (TP) and incorrectly detected (FP)), as shown in Equation 6.1. In other words, the precision represents the portion of the detected robots that are correct. A high precision value means that almost all of the robots detected by the hardware accelerator (e.g., FPGA or GPU) are correct.

In contrast to the precision, the recall value, also known as the detection rate, is calculated based on the ratio between correctly detected robots (TP) and the total number of robots that should have been detected (TP and FN), i.e., the total number of real robots (N_R). A high recall value means a high number of true robot markers can be detected from the total number of real robots.

In this evaluation, the number of correctly detected robots (TP) is acquired by verifying all of the detected robots. The verification is applied to all of coordinates of the detected robots by discovering the pentagon shape in the robot marker. If the pentagon is detected in the coordinates of the detected robot, these coordinates are considered to refer to a correctly detected robot (TP). Otherwise, it is not counted as a robot's location (FP). The algorithm in Teleworkbench [103] is applied to verify the robots' coordinates. It has a detection rate of approximately 99.99%.

$$Precision = \frac{1}{N_{Frame}} \times \sum_{i=1}^{N_{Frame}} \frac{TP_i}{TP_i + FP_i} \times 100\ \% \tag{6.1}$$

$$\begin{aligned} Recall &= \frac{1}{N_{Frame}} \times \sum_{i=1}^{N_{Frame}} \frac{TP_i}{TP_i + FN_i} \times 100\ \% \\ &= \frac{1}{N_{Frame}} \times \sum_{i=1}^{N_{Frame}} \frac{TP_i}{N_R} \times 100\ \% \end{aligned} \tag{6.2}$$

Where:

- N_{Frame} = Total number of frames.
- N_R = Total number of robot markers. $N_R = 4, 8, 16, 32, 64$
- TP_i = True positives in the current frame, which represents the number of robot markers that are correctly detected.

- FP_i = False positives in the current frame, which represents the number of robot markers that are incorrectly detected.
- FN_i = False negatives in the current frame, which represents the number of robot markers that are not detected, also called the detection failure.

The proposed design is targeted to have 99% of the detection performance for both the precision and recall metrics. Additionally, the post-processing algorithm in the host PC is also used to improve the output from the hardware accelerators (FPGA or GPU), by detecting the center of the pentagon in the robot marker, as previously described in section 3.3.3. With this detection performance, the design will be able to correctly detect almost all of the robots in the video frames. In this work, the detection performance does not include a scenario where some robots are crossing between two cameras. This is because the current detection test targets the ability of the algorithm and proposed hardware accelerator architectures to detect the locations of robots.

According to the experiments, both the FPGA-based and GPU-based designs show high detection performances for multiple robot localizations. The following sub-sections provide more details on the detection metric evaluations for both the FPGA- and GPU-accelerated computing systems.

6.1.1 FPGA implementation

Another basic requirement for verifying the functionality of the proposed design is implementing and integrating all of the developed video processing modules into the test system using the Virtex-4 FPGA, as depicted in Figure 6.1. This system supports the debugging and verification of the implementation using recorded videos from our Teleworkbench as the input datasets. The video is sent to the FPGA to detect the locations of the robot markers. Finally, these locations are delivered to the host PC for further analysis and display.

A verification is performed based on cropping an image of the region of interest, which is directed by the coordinates output from the FPGA hardware environment. The software environment system (host PC) verifies each cropped image (in 40 × 40 pixels) by detecting the pentagon shape and extracting the ID of the robot marker. If the pentagon and ID are detected, the coordinates are counted as valid coordinates. Otherwise, they are rejected as valid coordinates. This hardware-in-the-loop approach makes debugging and verification easier because the output of the FPGA design can be directly analyzed.

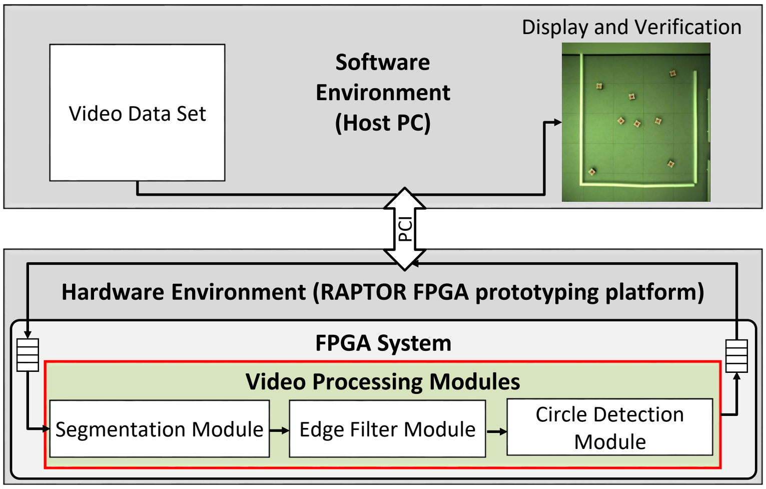


Figure 6.1: Developed test system for debugging and detection evaluation FPGA accelerated vision-based multi-robot tracking.

Precision and recall evaluations have been performed to investigate the detection performance of the proposed design. The system completes the precision and recall tests by continuously processing the robots' locations (the output of the FPGA) in all of the processed frames of our recorded video data sets. The radius (r) parameter is set to $r = 6$. Based on our empirical experiments, the threshold value for generating the circle center candidates in CHT or CSW is set to at least 62.5% of the vote-sampling value (e.g., if the CHT or CSW vote sampling = 16, the threshold = 10).

According to our experiments, the proposed design can handle multi-robot localization with a typical precision and recall of 99 % under well-defined lighting conditions. In other words, the design and its algorithm can provide a high performance for detecting the robot locations. The precision and recall for different circle detection methods (CHT-graph cluster and CSW-graph cluster) and a various number of robots markers are shown in Table 6.1 and Table 6.2. It is shown that the detection performance of our system using the CHT-graph cluster method with 32-votes is slightly higher than the same system with 16-votes for up to 8 robots. The CHT is more robust when two or more robots are collided as reported previously in Table 4.1. However, the 32-votes implementation is not capable of handling more than 8 robots in real-time due to the requirements discussed in Section 4.3.4.1. For the 16-votes implementation as well as for the CSW implementation, 64 robots can be detected with high precision and recall, as detailed in in Table 6.1 and Table 6.2. Commonly, the precision and recall values are increased when using a higher number of robots. It is because their values are related to the ratio between the true positive (TP) and the total number of detected robots. However, in some conditions, the collision between robots could slightly reduce the precision and recall performances.

Table 6.1: Precision and recall results of proposed system on FPGA, which were developed based on CHT-graph clustering algorithm.

N_R	**Precision (%)**		**Recall (%)**	
	S_{16}	S_{32}	S_{16}	S_{32}
4	99.48	99.57	99.40	99.44
8	99.69	99.69	99.19	99.25
16	99.56	N/A	99.47	N/A
32	99.86	N/A	99.71	N/A
64	99.81	N/A	99.72	N/A

Table 6.2: Precision and recall results of proposed system on FPGA, which were developed based on CSW-graph clustering algorithm.

N_R	**Precision (%)**		**Recall (%)**	
	S_{16}	S_{32}	S_{16}	S_{32}
4	99.66	99.67	99.43	99.47
8	99.64	99.68	99.28	99.42
16	99.57	99.71	99.59	99.73
32	99.86	99.93	99.72	99.84
64	99.80	99.81	99.75	99.78

Using the CSW-graph cluster method, our system has a higher precision and recall value than the CHT-graph cluster method for most of the test cases. This higher performance is achieved because the CSW-graph cluster method is more robust with respect to robot collisions. Additionally, the CSW-based method does not have the FIFO overflow and clock cycle limitation issues discussed in section 4.3.4.1. Therefore, and because it also requires less hardware resources as shown in Figure 6.2, the CSW-graph cluster method is the preferred solution for our implementation.

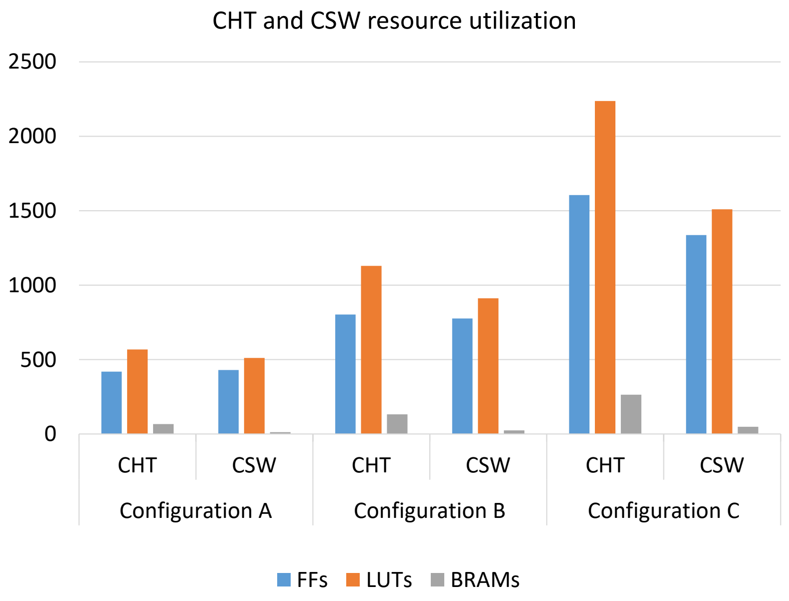


Figure 6.2: Comparison of CHT and CSW logic resources utilization in FPGA Virtex-4.

6.1.2 GPU implementation

Figure 6.3 illustrates a system that was built for debugging and testing the functionality of the proposed GPU-accelerated computing system.

To evaluate the functionality of the proposed GPU-accelerated computing system, all of the CUDA kernel and host PC codes are integrated into the test system. The testing system utilizes the same video datasets used for the debugging and verification of the FPGA-based system implementation. The GPU receives the video and continuously executes all of the CUDA kernels (segmentation, edge filter, and circle detection algorithms) to obtain the circle center candidates, which represent the coordinate candidates of the robots' locations. Sequentially, all of the coordinate candidates are loaded into the CPU's memory. Then, the host PC uses the graph clustering algorithm to obtain the true robot marker coordinates. Finally, the system verifies the detection performance of these results (coordinates) and displays the output in the host PC.

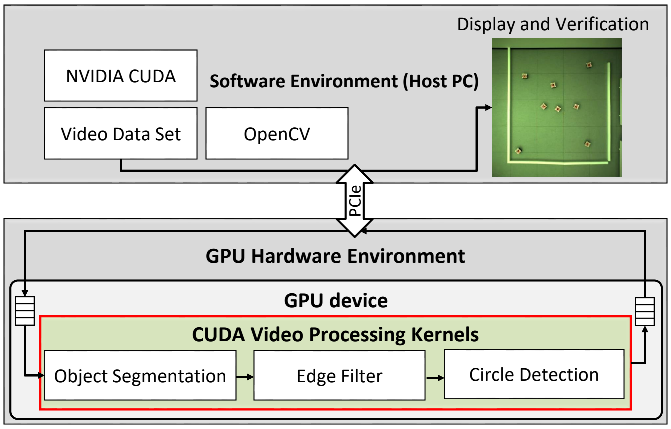


Figure 6.3: Testing system for debugging and detection evaluation of GPU accelerated vision-based multi-robot tracking.

In line with the algorithms and implemented GPU kernels that were previously described in section 5.2, the experiments for evaluating the detection performance of the proposed design considered three main aspects: the circle detection method, which used either the circular Hough transform (CHT) or circle scanning window (CSW); the utilization of a downscaling method; and a method for combining the gradient magnitude in the Sobel kernel, which applied either Pythagoras' theorem or an approximation technique. Accordingly, this work examined eight configurations to study the detection performance.

Here are the four configurations that were built based on the CHT:

- **CHT-Full-Pyth** configuration: the proposed design uses the CHT with the full video frame size and Pythagoras' theorem.
- **CHT-Full-Approx** configuration: the proposed design uses the CHT with the full video frame size and an approximation technique.
- **CHT-Resize-Pyth** configuration: the proposed design uses the CHT with a resized video frame and Pythagoras' theorem.
- **CHT-Resize-Approx** configuration: the proposed design uses the CHT with the full video frame size and an approximation technique.

In addition, four configurations were developed based on the CSW:

- **CSW-Full-Pyth** configuration: the proposed design uses the CSW with the full video frame size and Pythagoras' theorem.
- **CSW-Full-Approx** configuration: the proposed design uses the CSW with the full video frame size and an approximation technique.
- **CSW-Resize-Pyth** configuration: the proposed design uses the CSW with a resized video frame and Pythagoras' theorem.
- **CSW-Resize-Approx** configuration: the proposed design uses the CSW with the full video frame size and an approximation technique.

A detection performance evaluation was performed to investigate the precision and recall of the proposed design. The radius (r) parameter values for the full and resized (downscaled) methods were set to $r = 13$ and $r = 6$, respectively. The threshold value parameter for generating the circle center candidates in the CHT or CSW was set to at least 62.5% of the vote-sampling value.

Table 6.3 lists the detection rates and accuracies of the proposed system based on the CHT algorithm with different numbers of robots and CHT vote samples S (16 and 32). Based on the experiments, the proposed design can handle multi-robot localization with a typical precision and recall of 99%. This means that the design and its algorithm can provide an excellent performance for detecting the robots' locations.

When using a higher number of vote samples (S), the system produces a higher precision and recall. A configuration that uses the full video frame size has a slightly higher detection performance than one that works on resized video frames. This is

Table 6.3: Precision and recall values of proposed system developed based on CHT algorithm.

N_R	Precision (%)		Recall (%)	
	S_{16}	S_{32}	S_{16}	S_{32}
CHT-Full-Pyth configuration				
4	99.70	99.78	99.57	99.76
8	99.70	99.85	99.57	99.83
16	99.60	99.85	99.37	99.82
32	99.72	99.93	99.63	99.92
64	99.61	99.62	99.24	99.61
CHT-Full-Approx configuration				
4	99.66	99.79	99.56	99.77
8	99.72	99.85	99.59	99.84
16	99.61	99.84	99.36	99.81
32	99.72	99.92	99.59	99.91
64	99.61	99.62	99.25	99.62
CHT-Resize-Pyth configuration				
4	98.88	99.39	99.40	99.53
8	99.44	99.77	99.48	99.65
16	99.45	99.39	99.58	99.72
32	99.51	99.75	99.38	99.50
64	99.05	99.59	99.19	99.52
CHT-Resize-Approx configuration				
4	98.83	99.40	99.33	99.55
8	99.43	99.80	99.49	99.66
16	99.44	99.40	99.56	99.73
32	99.51	99.74	99.38	99.49
64	99.05	99.59	99.18	99.53

probably because the downscaling process for the image frame affects the circle shape. However, the detection performance difference between them (full and resized image configurations) is relatively small. Configurations that use Pythagoras' theorem or an approximation technique to calculate the gradient magnitude in the Sobel kernel have an almost similar detection result. In other words, the proposed design can simply select the method based on the processing time. The processing time evaluations are presented in section 6.2.2.

Table 6.4 lists the precision and recall values of the proposed system using the CSW technique. Typically, the proposed design using the CSW algorithm provides a detection performance similar to that obtained by a system with the CHT algorithm. When using a higher number of vote samples (S), the system produces a higher precision and recall. A configuration that utilizes the full video frame size has a slightly higher detection performance than one that works on resized video frames. The design and its algorithm can provide an excellent precision and recall of about 99% for detecting the robots' locations.

The CSW technique with the full frame size configuration provides almost the same detection performance as the CHT algorithm with a similar configuration. For the resized (downscaling) image configuration, the CSW technique is able to obtain a slightly higher detection than the CHT approach with the same configuration. This is probably because the CSW algorithm is more robust than the CHT, when considering robot collisions.

Both the CHT and CSW configurations utilizing 16 and 32 vote samples (S16 and S32) provide high detection performances for different numbers of robots (4, 8, 16, 32, and 64). This means that the proposed design and its algorithm are sufficiently robust for multiple robot tracking. As a result, the computing performances (or processing times) for executing the CHT and CSW algorithms in the GPU are the main factors when deciding on the best method. Section 6.2.2 presents a detail evaluation of the computing performance of the proposed design.

Table 6.4: Precision and recall values of proposed system developed based on CSW algorithm.

N_R	Precision (%)		Recall (%)	
	S_{16}	S_{32}	S_{16}	S_{32}
CSW-Full-Pyth configuration				
4	99.76	99.79	99.67	99.42
8	99.83	99.87	99.77	99.82
16	99.79	99.88	99.73	99.81
32	99.91	99.94	99.89	99.94
64	99.57	99.63	99.55	99.62
CSW-Full-Approx configuration				
4	99.73	99.79	99.66	99.43
8	99.83	99.86	99.80	99.82
16	99.78	99.88	99.76	99.81
32	99.91	99.95	99.90	99.93
64	99.56	99.64	99.55	99.63
CSW-Resize-Pyth configuration				
4	99.23	99.57	99.28	99.68
8	99.66	99.75	99.69	99.75
16	99.16	99.44	99.71	99.77
32	99.86	99.87	99.37	99.85
64	99.55	99.63	99.60	99.63
CSW-Resize-Approx configuration				
4	99.22	99.57	99.30	99.68
8	99.65	99.77	99.67	99.77
16	99.09	99.47	99.72	99.78
32	99.83	99.88	99.35	99.86
64	99.53	99.63	99.60	99.63

6.1.3 Comparisons

According to the detection test results for the FPGA- and GPU-accelerated computing systems discussed in the previous sections, both designs were able to provide high performances of about 99% for the multi-robot localization. This means that both the hardware accelerators and implemented algorithms are capable of providing high detection performances. Additionally, the post-processing in the host PC is also used to improved the coordinates from the hardware accelerator (FPGA or GPU), by detecting the pentagon located in the center of the circle in the robot marker.

The FPGA and GPU implementations have slightly different results. These differences resulted because some image processing operations in the FPGA design are based on a fixed point operation, while the GPU kernel uses floating point operations. For example, in the object segmentation module, the debayer, RGB to HSV, and color mask algorithms are designed to be efficiently mapped in the FPGA, as presented in section 4.3.2 while the RGB to HSV algorithm in the GPU is implemented using floating point operation, as shown in 3.4 and 3.5.

The implementation of the CHT algorithm in the FPGA-based design has to consider clock cycle limitation issues, as discussed in section 4.3.4.1, especially for a high number of CHT vote samples such as 32 (S32). However, there is a solution to deal with this issue using a lower CHT vote sample. For the 16-vote implementation, 64 robots can be detected with high precision and recall, as detailed in Table 6.1. Meanwhile, the GPU-based implementation does not have any issue regarding the number of CHT vote samples. The CHT can obtain a high detection performance with either 16 or 32 samples.

In contrast to the CHT algorithm, the CSW technique is more favorable for both FPGA and GPU implementations because of its performance and robustness for robot collision situations. Additionally, for the FPGA-based design, the CSW technique does not have any FIFO overflow and clock cycle limitation issues. It also utilizes fewer hardware resources than the CHT implementation.

Because both the FPGA- and GPU-based implementations can provide high detection rates, the computing performance and power efficiencies will be the main factors in deciding which approach is more favorable. The following sections present more details regarding the computing performance and resource efficiency of the proposed design.

6.2 Computing Performance

To find the most beneficial design, particularly in terms of the performance rate, computing performance evaluations using various configurations were completed for both the FPGA- and GPU-based hardware accelerators.

The design targeted multi-robot tracking for mini-robots such as AMiRo and Khepera. According to their specifications, AMiro has a speed of 0–800 mm/s, while Khepera has a speed of 0–1000 mm/s. Using a camera with a frame rate of 30 fps, the AMiRo robot can move a maximum of 26.67 mm between two consecutive frames. This is equal to 7 pixels or approximately 25% of the robot marker's size. Meanwhile, the Khepera robot can move up to 9 pixels between two consecutive frames, which is approximately 33% of the robot marker's size. This means that using a camera with a frame rate of 30 fps can provide sufficient speed to track the robots.

To obtain a frame rate of 30 fps, the processing time should not be more than 33 ms. The following sections provide further details on the computing performance evaluations of both the FPGA and GPU hardware accelerator implementations.

6.2.1 FPGA implementation

The maximum performance of the proposed design on the FPGA is calculated using Equation 6.3. It is shown that the performance mainly depends on the frame size (fr_{size}) and number of video stream hardware accelerators (N_{par}) (e.g., one for configuration A, two for configuration B, and four for configuration C). The maximum number of hardware accelerators is equal to the number of cameras, whereas the frame size is equal to the total frame size, merging the frames from all cameras. The number 64 in Equation 6.3 refers to the maximum number of detected robots. Our design is implemented on a Xilinx Virtex4-XC4VFX100 FPGA and a maximum clock frequency (f_{max}) of approximately 160 MHz is achieved for the CSW-based design, while about 150 MHz is reached for the CHT-based design.

$$fr_{rate} = f_{max}/((fr_{size}/N_{par}) + 64) \qquad (6.3)$$

The maximum clock frequency f_{max} is slightly different for the configurations (A, B, and C). For instance, the implementation of CSW based design on a Xilinx Virtex-4 XC4VFX100-11 FPGA can achieve a maximum clock frequency of 161 MHz for configuration A. While f_{max} is 172 MHz for configuration B and 162 MHz for configuration C.

Figure 6.4 and Table 6.5 show the computing performances for different frame sizes and numbers of hardware accelerators. For this evaluation, four cameras and three different hardware accelerator configurations (as illustrated in Figure 4.7 of chapter 4) were analyzed. As shown in Figure 6.4 and Table 6.5, the frame size and number accelerators are the main factors that influence the system's performance. This means that a higher frame rate can be achieved when using a lower resolution. Additionally, increasing the number of hardware accelerators significantly enhances the system's performance.

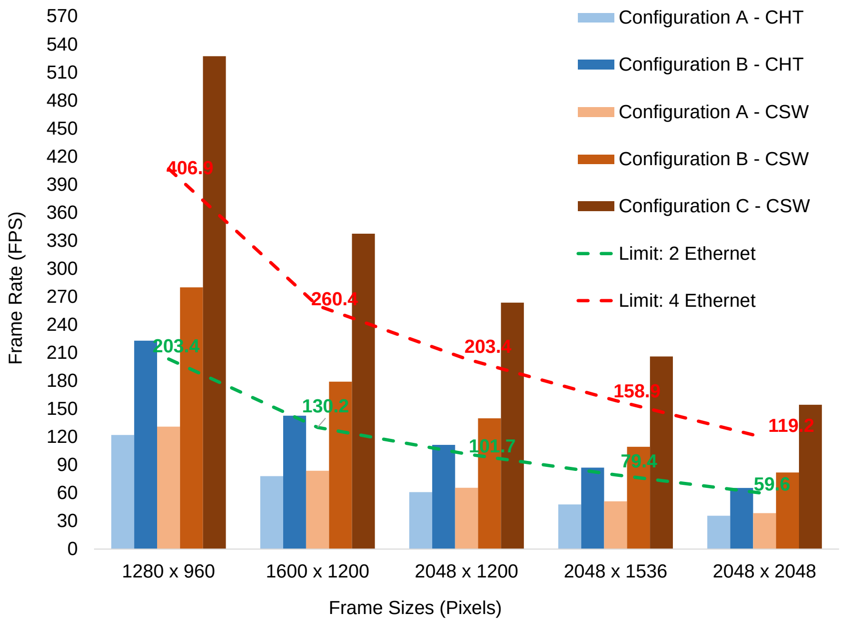


Figure 6.4: Computing performance of proposed design on Virtex-4 FPGA, measured in frames per second (fps).

Using configuration A, the maximum frame rate reaches 38 fps for a total image size of 2048 × 2048 pixels, whereas the maximum frame rate reaches 82 fps using configuration B. There is also an alternative to obtain the maximum performance using configuration C. This is implementing one hardware accelerator for every video stream from each camera, as illustrated in Figure 4.7-c. In this configuration, a maximum frame rate of 154 fps can be achieved. The system's performances with configurations B and C exceed the Gigabit Ethernet bandwidth. As shown in Figure 6.4, the performance of a design using four cameras and two Gigabit Ethernet interfaces is limited to 59 fps for a video frame with a total resolution of 2048 × 2048 pixels; meanwhile, a similar system with four Gigabit Ethernet interfaces achieves up to 119 fps.

Table 6.5: Computing performance of proposed design on Virtex-4 FPGA, measured in frames per second (fps).

Frame Size (pixels)	**Configuration A**		**Configuration B**		**Configuration C**	
	CHT	CSW	CHT	CSW	CHT	CSW
1280 × 960	122	131	223	279	N/A	527
1600 × 1200	78	83	142	179	N/A	337
2048 × 1200	61	65	111	140	N/A	263
2048 × 1536	47	51	87	109	N/A	206
2048 × 2048	35	38	65	82	N/A	154

Because of the VHDL-based design, the IP cores implemented in configurations A, B, and C are also applicable to more recent FPGA technologies. For example, they have been implemented on the Xilinx Virtex-6 SX475T-2 and Virtex-7 VX690T-2 to estimate the computing performance of the proposed design in newer FPGA technology. In contrast to the Virtex-4 FPGA, which is fabricated using 90 nm process technology, the Virtex-6 and Virtex-7 FPGAs are built with newer process technology. The Virtex-6 FPGA is built based on 40 nm, while Virtex-7 is manufactured based on 28 nm process technology.

For the implementation on Xilinx Virtex-6 SX475T-2 FPGA, a maximum clock frequency (f_{max}) of 222 MHz is achieved using configuration A, 217 MHz for configuration B and 217 MHz for configuration C. With respect to implementation on Xilinx Virtex-7 VX690T-2 FPGA, a maximum clock frequency of 250 MHz for configuration A is achieved. While f_{max} is 237 MHz for configuration B and 232 MHz for configuration C. These results are significantly higher than the maximum frequency achieved on the Virtex-4 FPGA. Accordingly, the Virtex-6 and Virtex-7 FPGAs provide higher frame rates compared to the Virtex-4 FPGA. Figure 6.5 shows the performances with different frame sizes and different numbers of hardware accelerators on the Virtex-6 and 7. Using configuration A, implementation on the Virtex-6 and Virtex-7 is able to reach maximum frame rates of 52 fps and 59 fps, respectively. This performance is increased significantly than that when using configuration B, which achieves frame rates of 103 fps on the Virtex-6 and 113 fps on the Virtex-7. Finally, configuration C produces the maximum performance, with maximum frame rates of 206 fps on the Virtex-6 and 221 fps on the Virtex-7. The performance of the design is limited by the bandwidth of the Gigabit Ethernet interface, as shown in Figure 6.5.

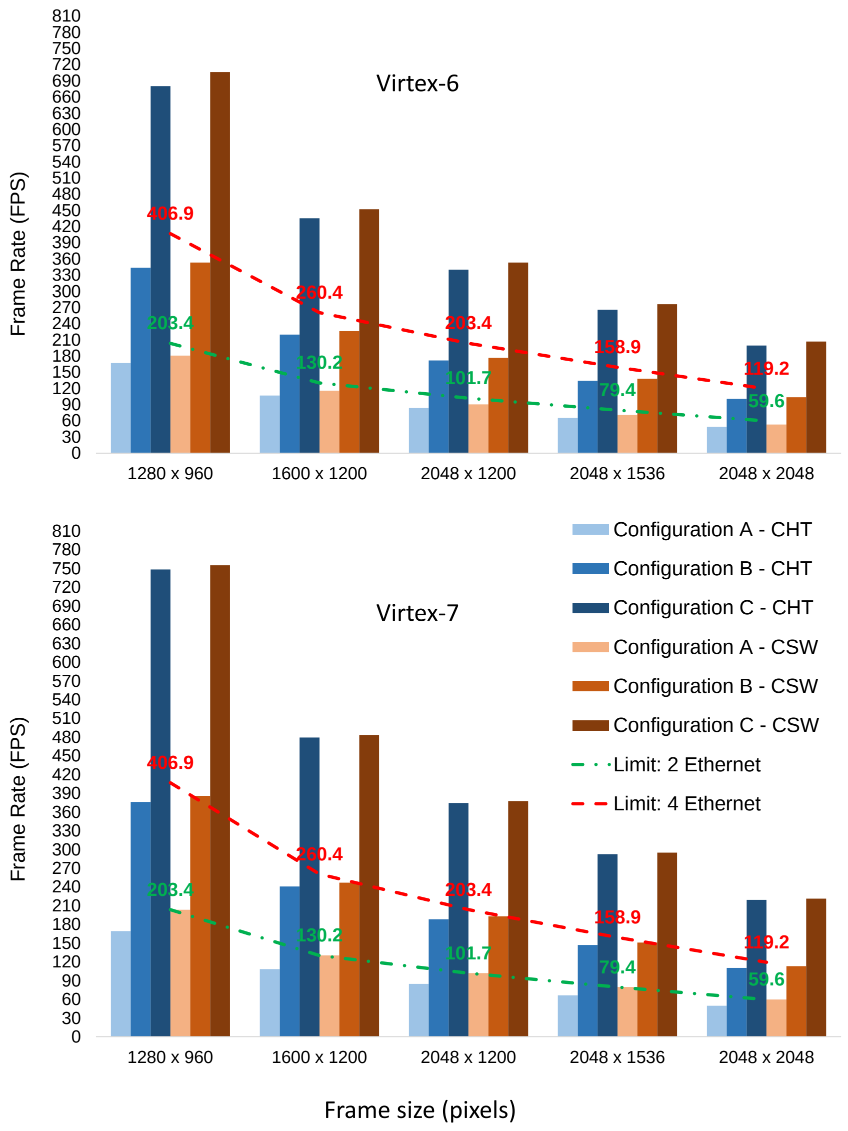


Figure 6.5: Performances of proposed design on Virtex-6 and Virtex-7 FPGA, measured in frames per second (fps).

Table 6.6: Computing performances of proposed design on Virtex-6 and -7 FPGA, measured in frames per second (fps).

Virtex-6						
Frame Size (pixels)	**Configuration A**		**Configuration B**		**Configuration C**	
	CHT	CSW	CHT	CSW	CHT	CSW
1280 × 960	166	180	343	353	680	706
1600 × 1200	106	115	219	226	435	452
2048 × 1200	83	90	171	176	340	353
2048 × 1536	65	70	134	138	265	275
2048 × 2048	48	52	100	103	199	206
Virtex-7						
Frame Size (pixels)	**Configuration A**		**Configuration B**		**Configuration C**	
	CHT	CSW	CHT	CSW	CHT	CSW
1280 × 960	169	203	375	385	748	755
1600 × 1200	108	130	240	246	479	483
2048 × 1200	84	101	188	192	374	377
2048 × 1536	66	79	146	150	292	295
2048 × 2048	49	59	110	113	219	221

To estimate the speed gain of the proposed design compared to the implementation on a state-of-the-art workstation, we implemented multi-robot detection using the OpenCV library on a 3.5 GHz Intel i7 quad core CPU (4770K, Haswell). The performance comparison between the FPGA and CPU implementations for different numbers of robots (1, 2, 4, 8, 16, 32, and 64) is depicted in Figure 6.6. The CPU implementation with the CHT-based algorithm for detecting the robot marker has a higher performance than the one with the CSW-based algorithm. The CSW-based algorithm is dependent on the video frame size, and the operation is performed on every pixel. In contrast, the CHT-based algorithm operation is performed only on active pixels. Therefore, the CHT-based algorithm is more favorable for the CPU implementation.

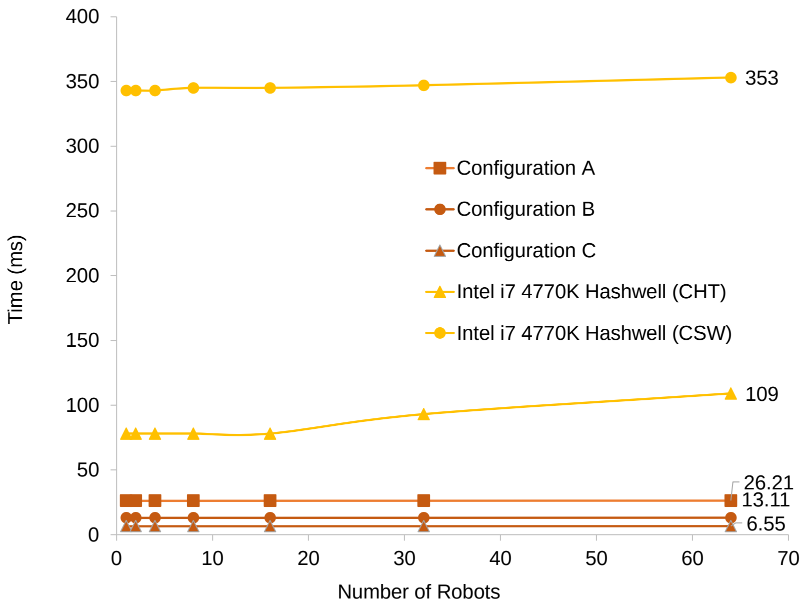


Figure 6.6: Performance comparison between FPGA and CPU implementations for different numbers of robots running on video frames with total resolution of 2048 × 2048 pixels, measured in timing operation (ms).

As shown in Figure 6.6, the Virtex-4 design with a clock frequency of 160 MHz achieves a speed-up of about 16–53 compared to the multi-threaded processor- based implementation, while the use of a newer FPGA such as the Virtex-6 or Virtex-7 FPGA causes a higher speed-up factor, as shown in Figure 6.7. For the CPU implementation, the execution time increases with the number of robots; while the execution time does not depend to the number of robots for the FPGA implementation.

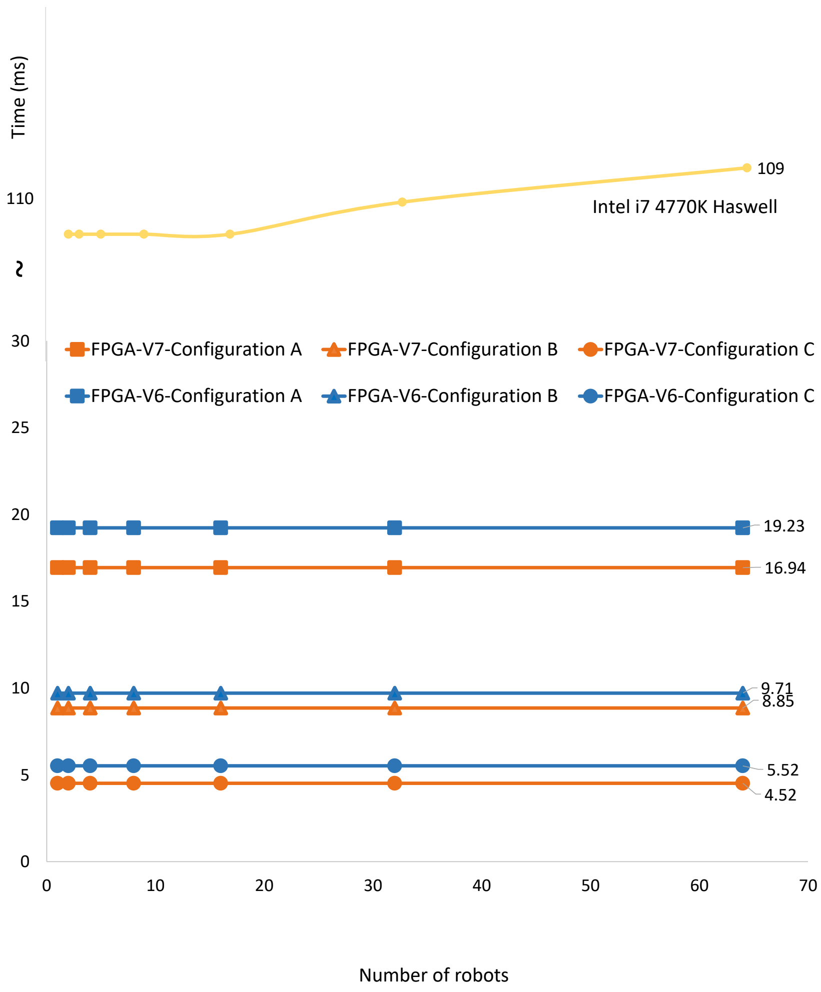


Figure 6.7: Performance comparison between FPGA- (V6 and V7) and CPU-based implementations for different numbers of robots running on video frames with total resolution of 2048 × 2048 pixels, measured in timing operation (ms).

6.2.2 GPU implementation

To obtain the computational performance of the proposed design on the GPU, the NVIDIA Visual Profiler software was used. This software is available as part of the CUDA Toolkit, which was installed on the testing platform, as previously shown in Figure 6.3.

Configuration with CHT-based algorithm

Figure 6.8 illustrates the execution times of the implemented kernels on the GTX-580 GPU for different numbers of robots (NRx) and CHT vote-samples (Sx), as well as different configurations of the CHT-based method. As shown in the chart, all of the configurations require the same execution time for the object segmentation kernel, but they have different execution times for the Sobel and CHT kernels. An increase in the vote-samples from 16 up to 32 significantly increases the execution time of the CHT kernel. This is because more iterations are required to execute a higher number of vote-samples. Because the voting procedure of the CHT algorithm is performed on every edge pixel, the execution time also gradually rises when the number of robots becomes higher.

The configurations that use the approximation method in the Sobel kernel have a slightly faster computing performance than similar configurations with the Pythagoras theorem. Both methods provide nearly the same detection performance, as previously reported in section 6.1.2. Therefore, the approximation method for the Sobel kernel is preferable because it has a faster processing time.

Downscaling the frame size of the segmented image successfully reduced the execution time in the Sobel and CHT kernels by up to 75%. Thus, the configurations that apply the downscaling technique to reduce the frame size of the segmented image have significantly faster execution times than the configurations with the full image. Indeed, the downscaling approach provides a speed-up factor of approximately two times. This produces only a slight reduction in the detection performance compared to the full frame size approach, as shown in the Table 6.3. Therefore, the configuration with downscaling for the frame size is more favorable for our application.

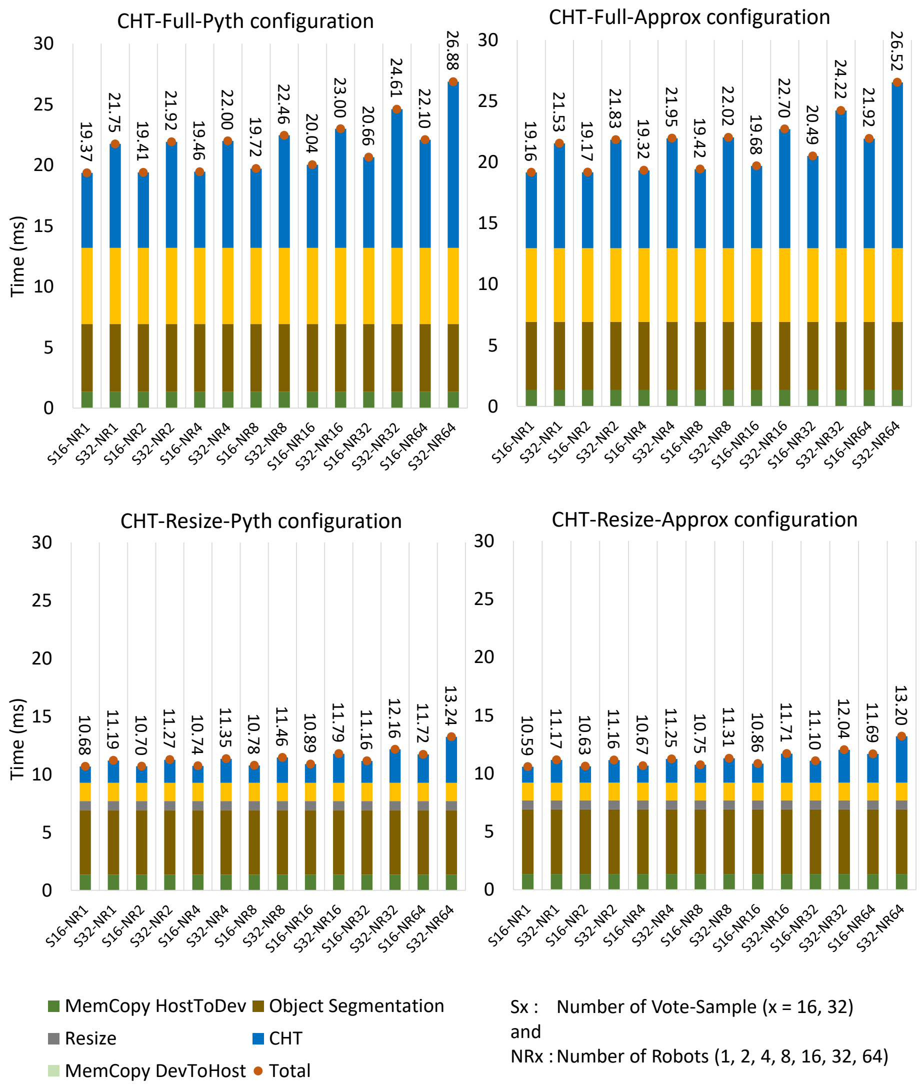


Figure 6.8: GPU computing performances on GTX-580 for implemented kernels. The experiments were performed for different numbers of robots and CHT votes samples, and measured in processing time (ms).

Figure 6.9 presents the execution time of the implemented kernel in the NVIDIA GTX- 780 GPU. Basically, the characteristics of the experiment results in the GTX-780 GPU are similar to the previous results obtained in the GTX-580 GPU. However, using the newer architecture and fabrication process technology, the GTX-780 GPU obtains a faster timing processing. The GTX-780 GPU architecture is built based on NVIDIA's Kepler architecture and manufactured using the 28 nm fabrication process, whereas the GTX-580 GPU is designed based on NVIDIA's Fermi architecture and fabricated using the 40 nm fabrication process.

Comparisons of the computing performances of the GTX-780 and GTX-580 GPUs for implementations of configurations with the CHT algorithm are shown in Figure 6.10. The GTX-780 implementations produce significantly faster execution times for all the scenarios compared to the implementations on the GTX-580. Utilizing its 2304 CUDA core processors, the GTX-780 GPU achieves up to a 30% faster execution time. The proposed design with S16 and S32 CHT vote samples is the most favorable configuration when considering the trade-off between computing performance (Figure 6.10) and detection performance (Table 6.3). Using configurations with 16 and 32 CHT vote samples (S16 and S32), the GTX-780 GPU is able to reach frame rates of 135 and 128 fps, respectively, whereas the GTX-580 obtains frame rates of 94 and 89 fps, respectively.

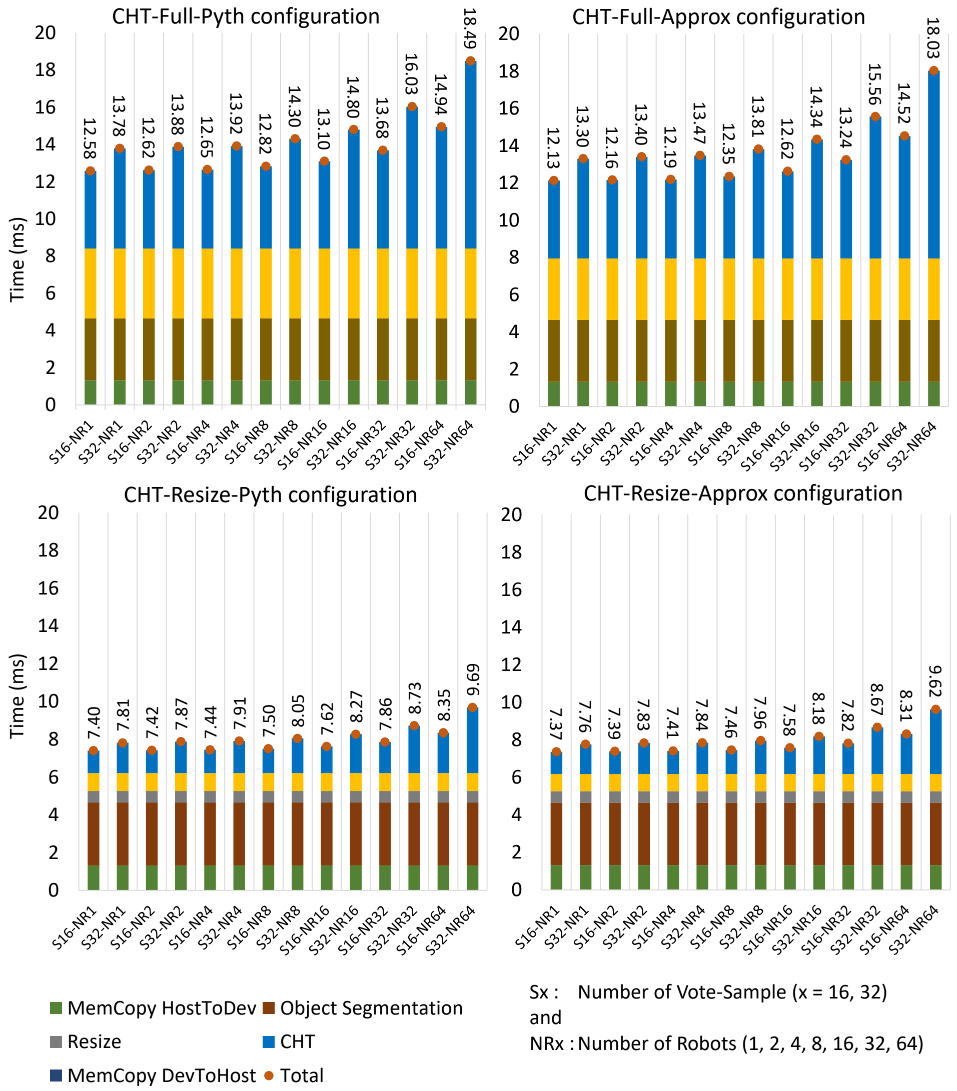


Figure 6.9: GPU computing performances on GTX-780 for implemented kernels. The experiments were performed using different numbers of robots and CHT votes samples, and measured in processing time (ms).

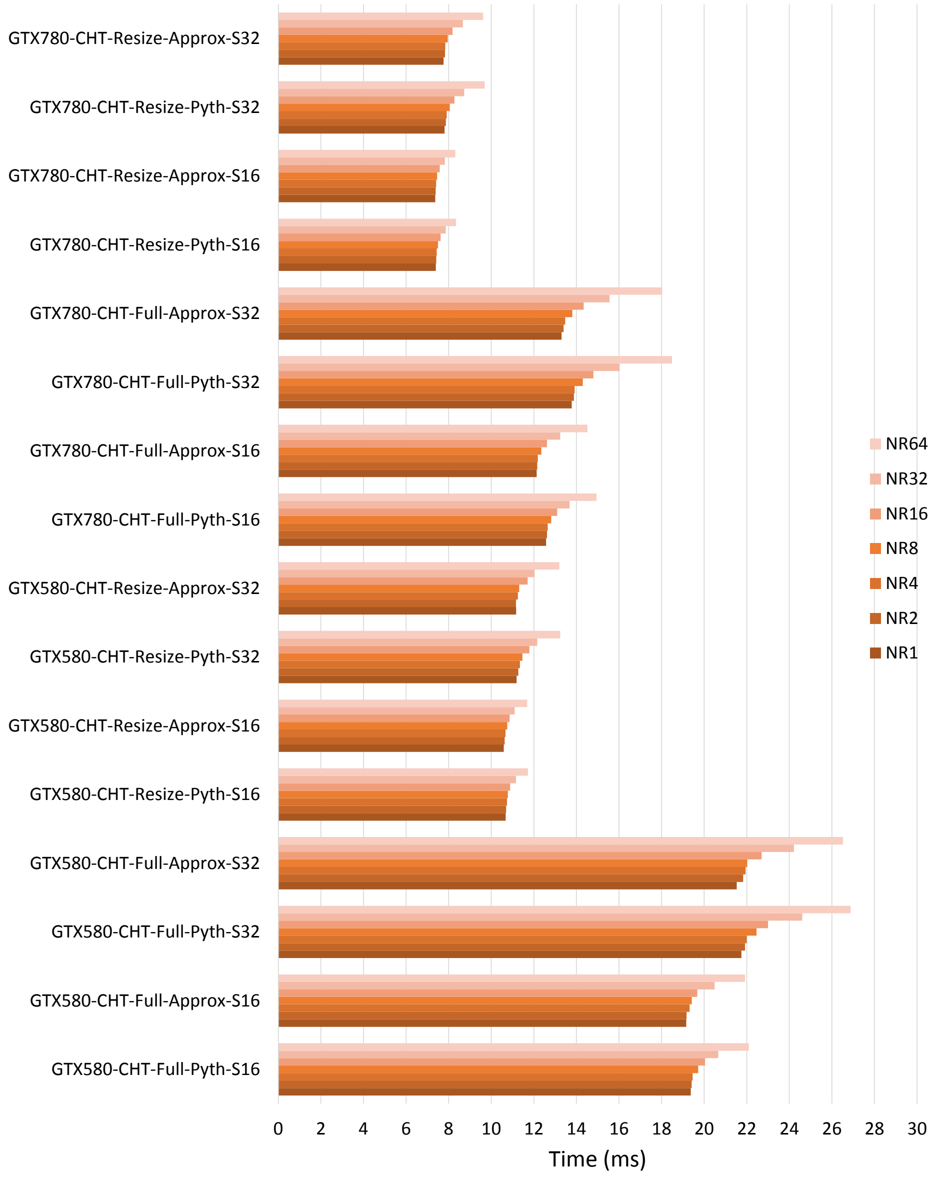


Figure 6.10: GPU computing performances for configurations using CHT algorithm.

Configuration with CSW-based algorithm

Figure 6.11 shows the execution times of the implemented kernels on the GTX-580 and GTX-780 GPUs for CSW-based configurations with different numbers of CSW vote samples (Sx). The CSW kernel performance was determined by the video frame size and number of CSW vote samples. Additionally, increasing the CSW vote-samples from 16 to 32 significantly increased the execution time, particularly for configurations with the full frame size approach. In contrast to the CHT kernel, the CSW kernel computing performance does not depend on the number of robots (NRx). It supports a scalable number of robots without affecting the execution time. A scenario with one robot has the same execution time as a scenario with four, sixty-four (64), or even a higher number of robots. Therefore, the CSW kernel is more favorable for an application that uses a large number of robots.

Configurations with the resize (downscaling) technique have significantly higher computing performances than those with the full frame approach, as shown in Figure 6.11. Reducing the frame size of the segmented image contributes an improvement of up to 75% for the CSW and Sobel kernels operations. Overall, it provides a speed-up factor of approximately two times for the GPU implementation compared to the full frame size approach; yet, this technique only produces a small reduction in the detection performance. This particularly refers to configurations where the number of CSW sample votes are 16 and 32 (S16 and S32), as shown in the Table 6.4. Therefore, the proposed designs with S16 and S32 CSW vote samples are the favorable configurations considering the trade-off between the computing performance (Figure 6.11) and detection performance (Table 6.4). The fastest execution time is achieved when using 16 CSW vote samples (S16) combined with the downscaling technique.

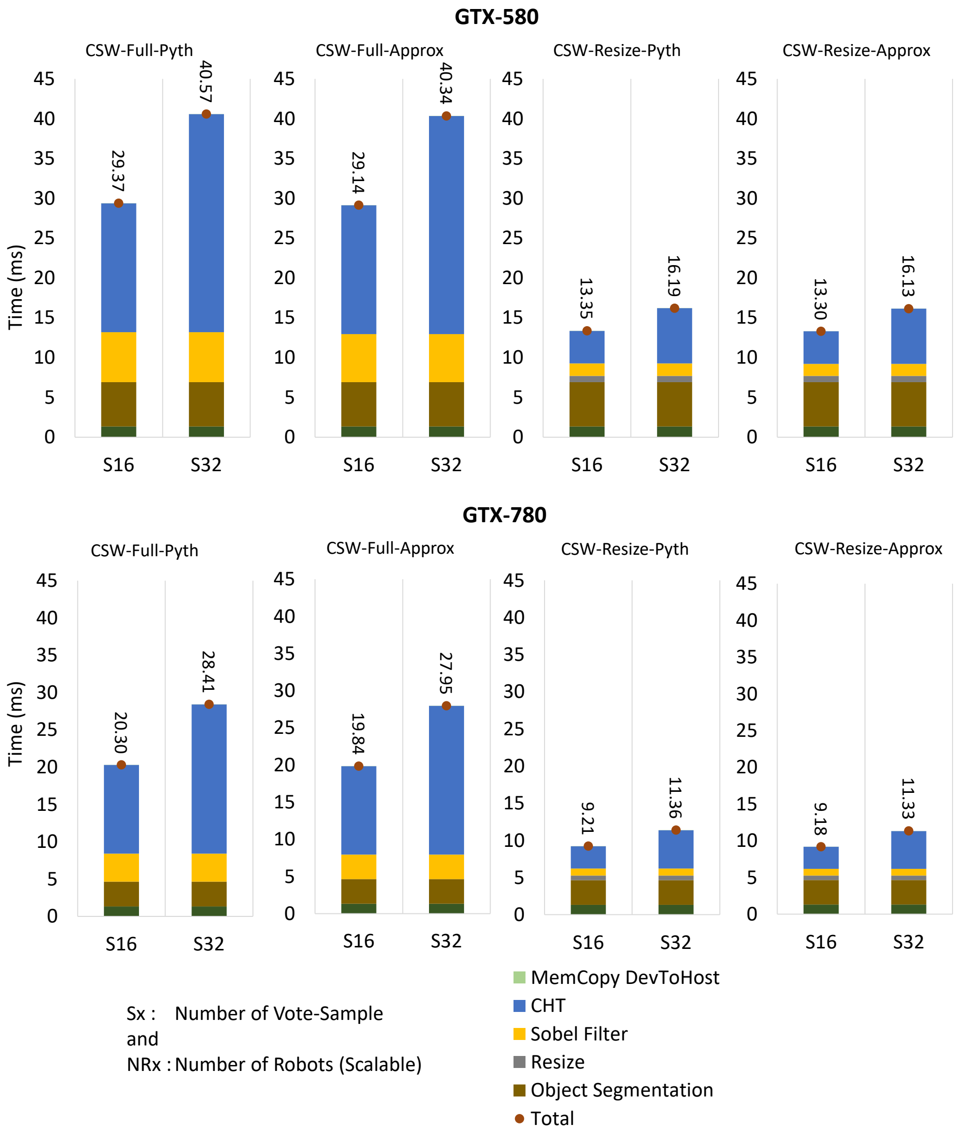


Figure 6.11: GPU computing performances for configurations using CSW algorithm in GTX-580 and GTX-780 GPU.

Computing performance comparison between CHT and CSW-based configurations

Figure 6.12 presents the computing performances of the proposed design for the CHT- and CSW-based configurations. For a similar number of vote sample and robots (up to 64), the CHT-based approach can achieve a faster execution time than the CSW-based design. A small number of robots produces large differences in the execution times between the CHT- and CSW-based configurations. However, this difference becomes smaller with an increase in the number of robots. This is because the execution time in the CSW-based design is constant for any number of robots, whereas the execution time in the CHT-based design increases exponentially with an increase in the number of robots, as shown in Figure 6.12 (top). Because the operation in the CHT algorithm depends on the active edge pixels, its execution time could be higher than the CSW-based design when the number of robots is significantly high (e.g., 100). Additionally, if there are many objects (such as obstacles) in the robot arena with the same color as the circle color of the robot marker (e.g., red), the execution time of the CHT-based design could potentially increase and become higher than the outcome shown in Figure 6.10 and Figure 6.12.

In the GPU implementation, the CPU is used to process the graph clustering algorithm. As depicted in Figure 6.12, its performance rate depends on the number of circle center candidates output from the GPU. For a small number of robots (e.g., NR1 up to NR16), the execution time is very short and insignificant compared to the execution time on the GPU. However, it is essential for a high number of robots such as 32 (NR32) or above. The execution time of the graph clustering operation could be decreased by increasing the threshold value of the circle center candidates on the GPU, with a consequence that this adjustment can reduce the detection performance.

Figure 6.13 depicts a comparison of the computing performances between the GPU-CPU implementation and CPU-only implementation for multi-robot localization. The GTX-580- and GTX-780-based designs achieve speed-ups of about 7 and 10, respectively, as compared to the multithreaded processor-based implementation. This performance is obtained using only a single GPU. However, a higher computing performance could be achieved using the multiple GPU approach when necessary.

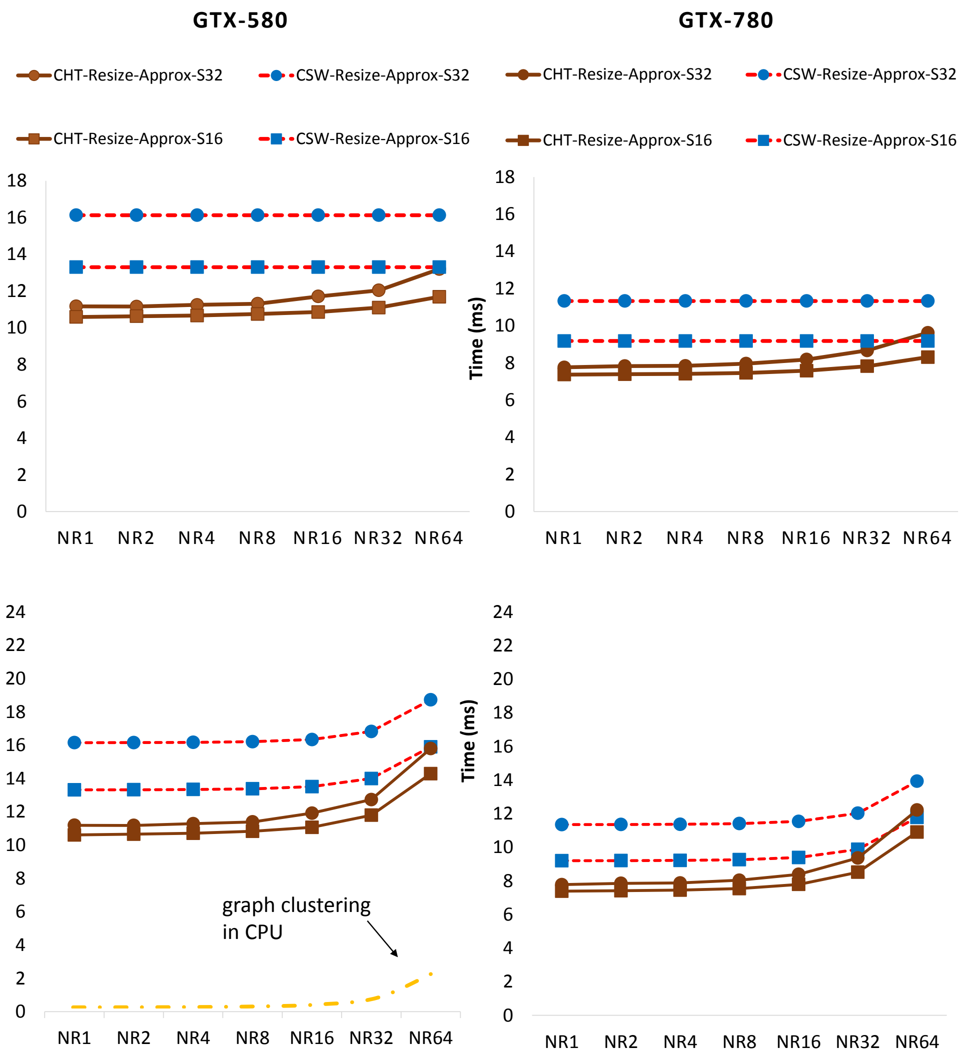


Figure 6.12: Computing performances of proposed design on GTX-580 and GTX-780 GPUs for CHT- and CSW-based configurations. Top: without clustering in CPU and Bottom: with clustering in CPU.

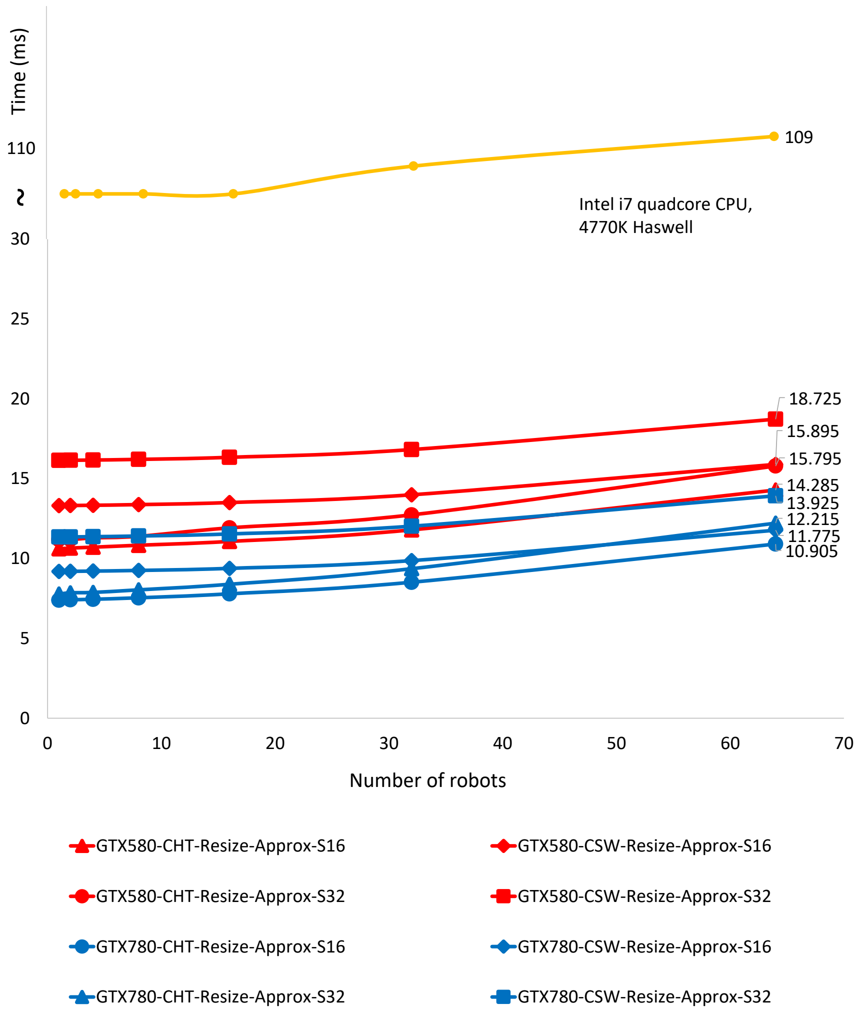


Figure 6.13: Comparison of computing performances between GPU-accelerated computing system and CPU-based system for detecting different numbers of robots (1 to 64), measured in processing time (ms).

6.2.3 Comparisons

Figure 6.14 presents a comparison of the computing performances of the FPGA- and GPU-based designs. The execution time in the FPGA-based design is relatively independent of the number of robots. Processing a small number of robots in the FPGA-based system takes the same time as processing a larger number of robots. For the GPU-based implementation, its execution time increases with the number of robots. As explained in the previous section, related to the GPU-based computing performance, the execution times of both the CHT-kernel in the GPU and graph-clustering algorithm in the CPU gradually increase with the number of robots. According to this characteristic, the FPGA-based design is more compatible than the GPU-based design with a system that has the scalability requirement, particularly in terms of the number of robots.

Despite using only a single GPU, the GPU-based design produces a higher computation performance than the FPGA-based implementation that applies a single stream hardware accelerator, as implemented in configuration A. The GPU-based design, which runs on a higher frequency than the FPGA-based design, can process the robot marker detection algorithm in a shorter time. However, if the amount of parallelism in the FPGA is increased, by adding streaming hardware accelerators, the FPGA performance surpasses the GPU performance.

As can be seen in Figure 6.14, if the number of hardware accelerators in the FPGA-based design is doubled, as implemented in configuration C, the FPGA-based design achieves a significantly higher computing performance than both the GTX-580 and GTX-780 GPU-based implementations. However, it should be noted that in this comparison the GPU-based design only uses the single GPU approach. When necessary, there is a feasible way to considerably enhance the computing performance of the GPU-based design using the multiple GPU approach by adding an extra GPU-card in the PCIe slot. Of course, the use of more FPGAs is also a feasible approach for enhancing the performance of the FPGA implementation, because it can increase the number of streaming hardware accelerators. Yet, this dissertation does not include both topics.

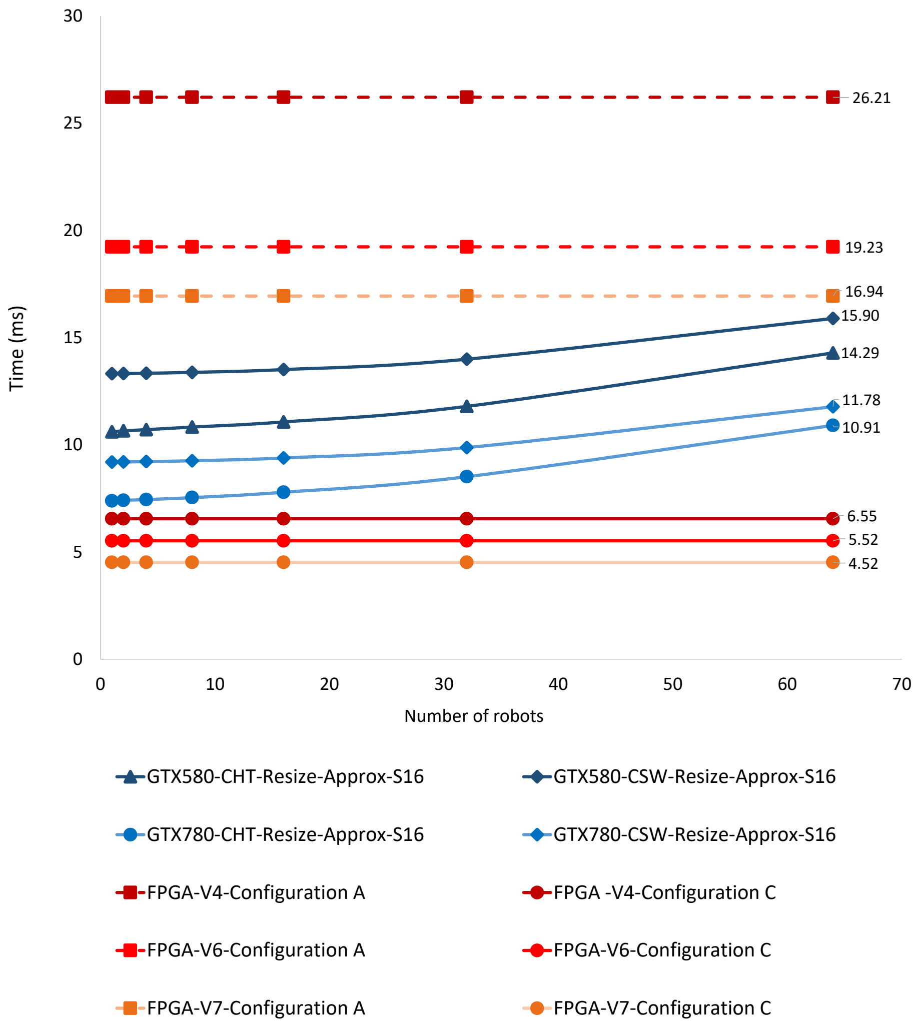


Figure 6.14: Comparison of computing performances between FPGA- and GPU-based designs, measured in processing time (ms). The execution time in the FPGA-based design is independent of the number of robots, whereas that of the GPU-based design gradually increases with the number of robots.

Figure 6.15 shows a computing performance comparison between the CPU, FPGA, and GPU for detecting 64 robots using a total frame resolution of 2048 × 2048 pixels. Both the FPGA- and GPU-based hardware accelerators have much higher computing performances than the Intel i7 4770K quadcore CPU.

The GPU implementation has the highest frame rate compared to the Intel i7 4770K quad-core CPU implementation and the Virtex-4 FPGA implementation with a single stream hardware accelerator (configuration A). However, the FPGA design in configuration C, which uses four stream hardware accelerators, surpasses the GTX-580 and GTX-780 GPU designs, which are implemented on a single GPU unit.

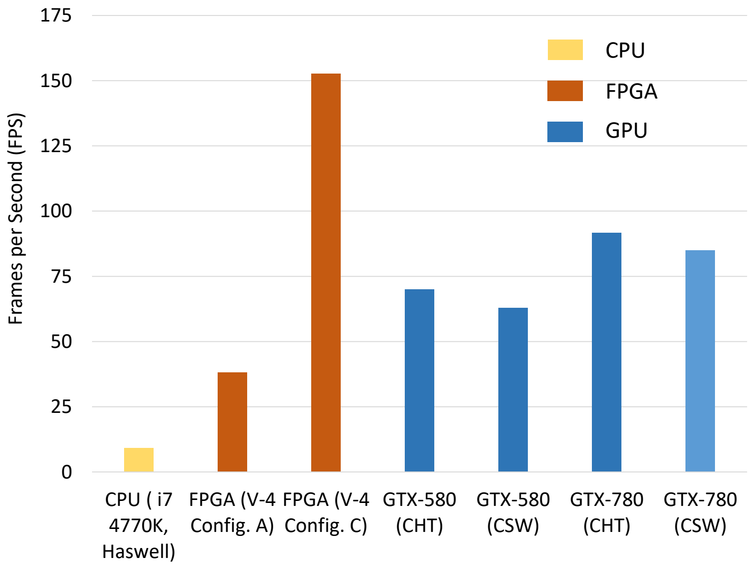


Figure 6.15: Comparison of computing performances between CPU, FPGA, and GPU for detecting 64 robot markers, measured in frames per second (fps).

The FPGA technology uses its flexibility, inherent parallel structure, and customized design to increased the computing performance. The GPU performance can also be increased by using multiple units, performing the algorithm on multiple GPUs. Because both hardware accelerators are scalable, the power consumption and power efficiency become important issues to determine which technology has greater benefits. Therefore, in the following section, the power consumption and power efficiency values of the CPU, FPGA, and GPU are evaluated.

6.2.4 Power Efficiency Evaluation

The power efficiency metric refers to the ratio between the computing performance and the required power for processing computations. The efficiency measurement is calculated based on the number of performances per watt (power consumption), where the computing performance here is equal to the achieved frame rate per second (fps).

Power consumption measurements were done to obtain the wattages for computing the algorithm in the CPU, FPGA, and GPU. The CPU implementation was performed on a host PC with an Intel i7 CPU (Haswell 4770K), running at 3.5 GHz with a multi-thread approach. The FPGA-design was implemented on a RAPTOR development board using a single Virtex-4 daughter board and an additional Ethernet board, while the GPU implementation was performed in the GTX-580 and GTX-780 GPUs. The idle state, as shown in Figure 6.16, was the condition where there was no computation. For the host PC, the power consumption in the idle state included the power of all the integrated devices (e.g., hard-disk, memory, etc). While for the FPGA, the idle state was a condition where there was no programming file uploaded to the FPGA. The power consumed (used) on the CPU, FPGA, and GPU was calculated by subtracting the power in the idle state from that measured in the active state (the computation was running).

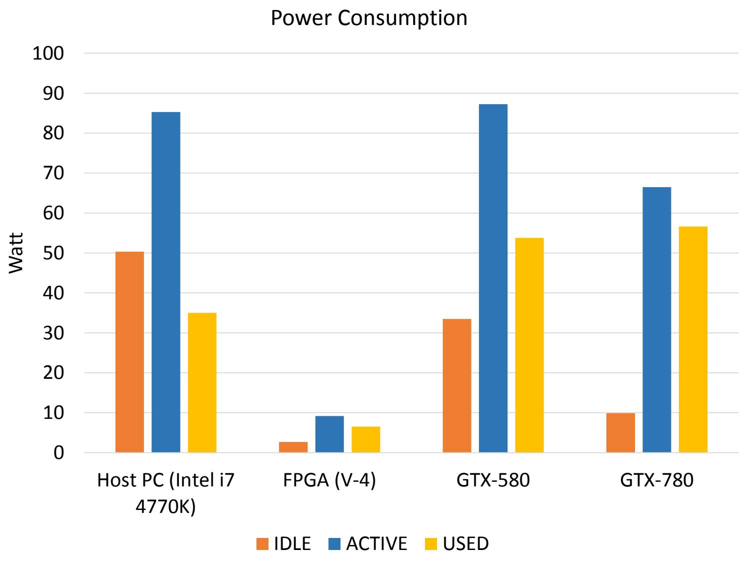


Figure 6.16: Power consumption comparison between CPU, FPGA, and GPU for detecting 64 robots on frame size of 2048 × 208 pixels.

The power efficiency analyses in the CPU, FPGA, and GPU were done without including the power consumption in the post-processing algorithm. This work focused on the computationally intensive algorithm for detecting the robot marker locations. Additionally, because all the platforms used the same algorithm for the post-processing and computing this algorithm in the same host PC, the power consumption difference between platforms for computing the post-processing algorithm will be very small, particularly between the FPGA and GPU platforms. The power consumption and power efficiency measurements are presented in Figure 6.16 and Figure 6.17, respectively.

Figure 6.16 shows the power consumption test results for the CPU, GPU, and FPGA. The FPGA-based hardware accelerator has a significantly lower power consumption than the other devices. It is approximately six to eight times lower. This means that the FPGA implementation provides a lower power consumption when it is used as an alternative hardware for accelerating the computation in vision-based multi-robot tracking. The CPU implementation uses less power than the GPU implementation. However, the GPU has a significantly higher computing performance than the CPU, as previously shown in Figure 6.15. The GTX-580 implementation has a slightly lower power consumption than the GTX-780, but it consumes significantly more power than the GTX-780 in the idle state. This indicates that the newer GPU technology in the GTX-780 offers less power consumption in the idle condition.

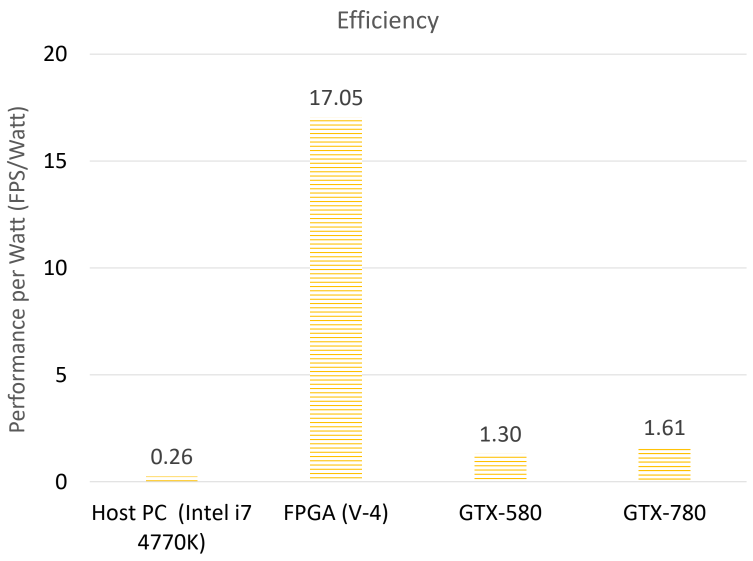


Figure 6.17: Power efficiency comparison between CPU, FPGA, and GPU for vision-based multi-robot tracking.

Figure 6.17 presents the results of the power efficiency comparison between the CPU (intel i7 4770K), Virtex-4 FPGA, and GPU for computing the algorithm to detect the locations of multiple robots. The CPU has the lowest power efficiency compared to the other devices. Although the CPU requires less power than the GPU, it has a much slower computing performance than the GPU. Therefore, it has only a small power efficiency. Additionally, the results in Figure 6.17 show that both hardware accelerator devices (FPGA and GPU) can provide better performances per watt than the CPU. This means that both the FPGA and GPU hardware accelerators can be used to enhance the computing performance per watt when processing the vision-based multi-robot tracking algorithm.

The GPU provides a higher power efficiency than the CPU. Additionally, the newer generation of GPU (GTX-780) provides better performance than the older generation of GPU (GTX-580). As shown in Figure 6.17, the Virtex-4 FPGA provides the highest performance per watt compared to the other devices. This indicates that the FPGA is very efficient and very suitable for systems that require less power consumption and high computing performance.

6.3 Analysis

The previous sections have comprehensively presented the detection performance, computing performance, and power efficiency evaluations of the proposed design. The evaluations focused on the computationally intensive parts of the vision-based multi-robot tracking algorithm, which are performed in the hardware accelerators (FPGA and GPU).

According to the detection's test results for the FPGA- and GPU-accelerated computing systems discussed in the previous sections, both designs are able to provide detection performances (precise and recall) of about 99% for multi-robot localization. This means that the hardware accelerators and implemented algorithms are capable of providing a high detection performance. Additionally, the post-processing in the host PC is also used to improve the coordinates from the hardware accelerator (FPGA or GPU), by detecting the pentagon located in the center of the circle in the robot marker. In this work, the detection performance only focused on the ability of the algorithm and hardware accelerators' architectures to detect the robot locations. Scenarios where some robots are crossing between two cameras were not taken into account. Therefore, additional operations are needed in the CPU to handle this scenario.

Regarding the computing performance, both the FPGA- and GPU-based hardware accelerators have significantly higher computing performances than the Intel i7 4770K quad-core CPU. This means that the inherent parallel structure of the FPGA and the SIMT approach of the GPU hardware accelerators can be used to significantly enhance the vision-based multi-robot tracking algorithm.

The FPGA-based hardware accelerator implementation can reach a frame rate of 154 fps with a total resolution of 2048 × 2048 pixels using a Xilinx Virtex-4 FX100-11 FPGA. The achieved frame rate is optimized by utilizing four streaming hardware accelerators, working in parallel. Furthermore, the computation performance can be increased when using newer FPGA technology. For example, the designs were implemented on the Xilinx Virtex-6 XC6SX475T-2 and Virtex-7 VX690T-2 to estimate the computing performances of the proposed design in the more recent FPGA technology. The Virtex-6 and Virtex-7 FPGAs are able to achieve maximum clock frequencies of 190 MHz and 230 MHz, respectively. These results are significantly higher than the maximum frequency achieved on the Virtex-4 FPGA. Accordingly, both newer generations of FPGAs demonstrate higher frame rates compared to the Virtex-4 FPGA. In addition to obtaining a faster maximum clock frequency, using the newer FPGA technology also means that more logic resources can be utilized. Hence, greater parallelism and scalability can be supported using these newer FPGA devices.

Meanwhile, the implementation of GPU-based hardware accelerator, using the GTX-580 and GTX-780 GPUs, produces maximums of 70 fps and 91 fps, respectively, with a total resolution of 2048 × 2048 pixels. This means that both GPUs reach higher computation performances compared to the FPGA-based implementation that applies a single stream hardware accelerator. However, these performances are still lower than the implementation with four streaming hardware accelerators in the FPGA. Nevertheless, in this comparison, the GPU-based design only used the single GPU approach. If required, the GPU performance can also be increased using multiple units and performing the algorithm on multiple GPUs.

Despite the fact that the FPGA and GPU are able to achieve very high computation performances, the post processing in the CPU to some extent could limit the overall performance. To acquire the robots' orientations and IDs, the advantage of the multi-core architecture in the CPU is employed, processing all the sub-images (where an image consists of a robot marker) in a multi-thread approach. Because the size of the sub-images is very small (40 × 40 pixels), they can easily be processed by a typical host PC in real time. To some extent, the scalability is limited when using a large number of robots. For the targeted 64 robots, there is no problem with processing in the host PC. It uses four threads for the computation of the robots' orientations and IDs; hence, the CPU can simultaneously process four sub-images. Based on the experiments, a processing time of approximately 12 ms can be reached for 64 robots. Meanwhile, for one robot, it takes only about 0.72 ms. This computation performance can be increased when more threads are utilized, upgrading the CPU to the latest version with more cores. Another solution could involve using a many-core processor in the proposed system. This is a special processor with an architecture containing dozens to hundreds of lightweight CPU cores such as CoreVA [55; 97].

A comparison of the proposed design with other architectures discussed in literature is shown in Table 6.7. As can be seen, our design supports a higher resolution (2048 × 2048), more cameras (4), higher number of robots (64), larger robot arena (6 m × 6 m), and faster execution time (6.55 ms) compared to other implementations. The achieved frame rate in the FPGA design is optimized by utilizing four streaming hardware accelerators, working in parallel. Meanwhile, the GPU implementation, which operates on a high frequency and successfully employs its many cores, produces a higher computation performance than the FPGA implementation, which applies a single stream hardware accelerator.

Table 6.7: Comparison with existing architectures.

Arch.	**Tech.**	**Resolution (pixels)**	**Exec. Time (ms)**	**Arena ($meter^2$)**	**Robots**	**Cam.**
[92]	Stratix II	640 × 480	8.6	-	-	1
[42; 43]	Virtex-5	640 × 480	30	-	1	1
[119; 120]	Cyclone II	1280 × 1024	29.4	1.2 × 1.6	min. 3	1
[9; 34]	Cyclone IV	640 × 480	7.8	1.5 × 1.3	6 to 22	1
Our design	Virtex-4	2048 × 2048	6.55 to 26.2	6 × 6	64	4
Our design	GPU GTX780	2048 × 2048	10.9	6 × 6	64	4

Both the FPGA- and GPU-based designs are scalable to support higher computing performance. Therefore, the power consumption and power efficiency become important issues to determine which technology provides greater benefits. For instance, these issues are essential when multiple hardware accelerators are used in the computing systems.

The GTX-580 and GTX-780 GPU implementations have higher power consumptions than the host PC (Intel i7 4770K, quadcore CPU) implementation. However, because the GPU implementations have significantly higher computing performances than the CPU, they also have higher power efficiencies (fps/watt). The power consumption issue is the main drawback of the current GPU implementation. This issue can limit the scalability of the GPU-accelerated computing systems. However,newer GPU technology always improves this power consumption issue, and the power of a GPU becomes lower from generation to generation without reducing its computing performance.

Regarding the power consumption in the FPGA, the FPGA-based hardware accelerator has a significantly lower power consumption than the CPU and GPU. It is approximately six to eight times lower. Additionally, using its flexibility, inherent parallel structure, and customized design, the FPGA design also has high computing performance. Therefore, the FPGA-based hardware accelerator provides the highest efficiency or computing performance per watt (fps/watt). This means that the FPGA-based hardware accelerators are very efficient and very suitable for systems that require less power consumption with high computing performance. The proposed FPGA-accelerated computing system is limited by the interface to the CPU. Currently, it utilizes a PCI interface to transfer the data from the FPGA to host PC and vice versa. The PCI interface is very slow and is the bottleneck for the system. Therefore, this interface should be upgraded to the PCI-express, which is already used in the GPU hardware accelerator. This issue technically can be fixed because FPGAs are customizeable.

Determining the best technology for an application should be based not only on some quantitative issues (e.g., computing performance, power consumption, and power efficiency) but also on qualitative parameters such as the development process. This development process is also related to the design complexity, development time, and time to market issues. The development process in a GPU is relatively easier and faster than in an FPGA, but more difficult than the CPU. This is because debugging and interactive simulations, as the main factor in the development process, are fully accommodated in a GPU development system, as described in chapter 3. Meanwhile, the FPGA development process is more complicated and time-consuming than that of the GPU. In the FPGA design, the image processing algorithm cannot be developed directly on the targeted FPGA device. This is because the development cycles (e.g., synthesize, translate, map, place, and route) require too much time. Therefore, in the FPGA design for image processing applications, it becomes impracticable to have an interactive design.

6.4 Summary

This chapter has presented the results and analysis of the proposed design. In particular, the detection performance, computation performance, power consumption, and power efficiency for both the FPGA- and GPU-accelerated computing systems have been considered. Regarding the detection performance, this chapter has shown that both the FPGA- and GPU-based designs are able to provide detection performances (precision and recall) of about 99% for multi-robot localizations. Comparisons between the

CPU, FPGA, and GPU implementations for the computation of vision-based multi-robot tracking algorithm were also presented in detail in this chapter.

In conclusion, the implementations of vision-based multi-robot tracking in different technologies (CPU, FPGA, and GPU) are illustrated in Figure 6.18. The CPU technology provides the fastest development time and easiness of programming, but its implementation has issues with the computing performance, power consumption, and power efficiency. Meanwhile, the GPU technology is suitable for implementations that require a high computing performance, good power efficiency, and adequate development time. However, the implementation of the GPU technology in this application is limited by the power consumption. Finally, the FPGA technology offers a high power efficiency, very good (low) power consumption, and high computing performance. The development time and complexity of the programming implementation are the main drawbacks of the FPGA implementation.

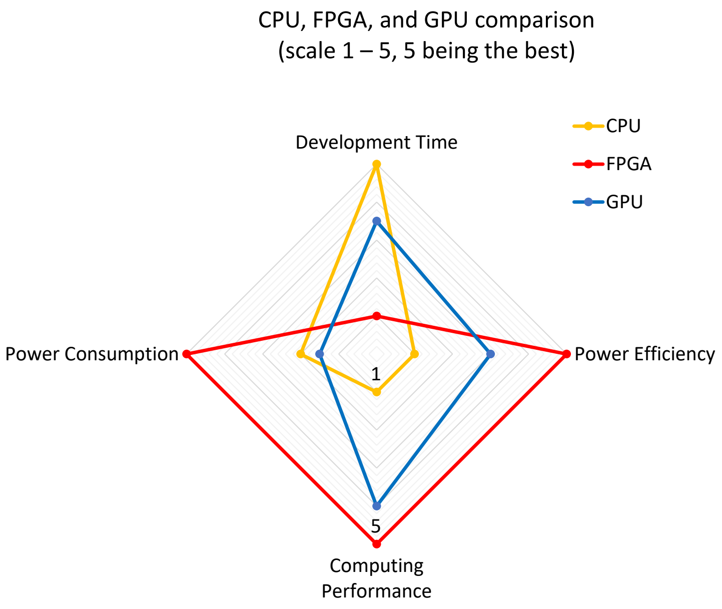


Figure 6.18: Comparison of CPU, FPGA, and GPU implementations for vision-based multi-robot tracking application.

7 Conclusions and Outlook

In this thesis, FPGA- and GPU-accelerated computing systems for vision-based multi-robot tracking were proposed. These designs refer to heterogeneous computing systems that combine a CPU and hardware accelerator, either the FPGA or GPU. In many cases of vision-based robot tracking systems, the computational requirements for extracting the relevant information (e.g., locations, orientations, and identities (IDs) of robots) from video data increase along with the number of tracked robots, the video frame size, and the number of operated cameras. In contrast to the development of the previous computing systems, which typically used several high-performance workstations for the parallel processing of data from multiple cameras, the heterogeneous computing system approach releases the host computer from the computation-intensive tasks by utilizing the FPGA or GPU.

This thesis emphasizes the implementations of two distinct heterogeneous computing systems for vision-based multi-robot tracking applications, encompassing the use of the FPGA and GPU as hardware accelerators. The implementations on the FPGA- and GPU-based heterogeneous computing systems have been demonstrated in chapter 4 and chapter 5, respectively. Based on the modular and parallel architecture of the FPGA, a collection of video processing modules was developed, capable of detecting the locations of multiple robots using individual markers. The video processing modules involve two unique architectures for the circle detection of the robot's marker. The first one integrates a combination of the CHT and graph cluster algorithms, while the second architecture combines the CSW technique with a graph cluster algorithm. Meanwhile, the GPU implementation relies on a large number of lightweight programmable cores that concurrently execute the vision processing algorithm.

Considering the differences between the FPGA and GPU, this work compared and analyzed the FPGA- and GPU-based computing systems to find the optimal system for multi-robot tracking applications. In particular, this thesis implemented FPGA- and GPU-accelerated heterogeneous computing systems, compared the results, and determined the advantages that could be achieved using both computing systems for vision-based multi-robot tracking applications. In doing so, this thesis focused on the system architecture, detection performance, computing performance, and power efficiency. The examinations and analysis of the proposed systems were discussed in chapter 6 using different generations of FPGAs (e.g., Xilinx Virtex-4, Virtex-6, and Virtex-7) and GPUs (e.g., GTX-580 and GTX-780).

7.1 Conclusions

This thesis has described details about the basic concept of a vision-based robot tracking system and the related work. It has shown the need for a computing system that uses the benefits of the CPU and hardware accelerators (e.g., FPGA and GPU) to enhance the computing performance of a vision-based multi-robot tracking algorithm. The architectures of heterogeneous computing systems for vision-based multi-robot tracking and their design flows, both in the FPGA-and GPU-accelerated platforms, have been presented in this thesis.

The result of this thesis show that the FPGA- and GPU-based hardware accelerators strongly enhance the computational performance of the computing system for vision-based multi-robot tracking. These hardware accelerators release the host computer from the computationally intensive tasks, complementing the CPU's function to perform comprehensive vision-based multi-robot tracking applications. Furthermore, both the FPGA- and GPU-based hardware accelerators can achieve high accuracy, computational performance, and power efficiency.

According to the detection performance, this thesis have shown that the proposed designs can handle multi-robot localization with a typical detection performance (precision and recall) of 99% under well-defined lighting conditions, as reported in section 6.1. This means that the proposed hardware accelerators and implemented algorithms achieve a high detection performance for detecting the robot locations. Both the CHT-graph cluster and CSW-graph cluster methods produce high detection performances for detecting multiple robots. However, the CSW technique is more favorable than the CHT for both the FPGA and GPU implementations, because of its detection performance and robustness for robot collision situations.

This thesis shows that both the FPGA- and GPU-based hardware accelerators have significantly higher computing performances than the Intel i7 4770K quad-core CPU. Therefore, both hardware accelerators are very good alternatives to enhance the computational performance of a computing system for vision-based multi-robot tracking applications. The FPGA-based hardware accelerator can reach up to 154 fps with a total resolution of 2048 × 2048 pixels using a Xilinx Virtex-4 FX100-11. The achieved frame rate is optimized by utilizing four streaming hardware accelerators, working in parallel. Furthermore, the computational performance could be increased by using newer FPGA technology. The GPU implementation, which operates on a higher frequency than the

FPGA design and employs many cores using the SIMT approach, surpasses the computational performance of an FPGA single stream hardware accelerator implementation. Additionally, the development process in a GPU is relatively easier and faster than in an FPGA. This is because of the debugging and interactive simulations, which are the main factors in the developing process.

Regarding the power consumption and power efficiency, this thesis has shown that the FPGA-based hardware accelerator has a significantly lower power consumption than the CPU and GPU, which is approximately six to eight times lower. Additionally, it also has a high power efficiency because the FPGA implementation can achieve a high computation performance per watt (fps/watt). Therefore, the FPGA-based designs are very suitable for systems that require less power consumption with high computing performance. Meanwhile, the GPU-based hardware accelerator (GTX-580 and GTX780) implementations were better than the host PC (Intel i7 4770K, quadcore CPU) implementation. The power consumption issue was the main drawback for the GPU. However, newer GPU technology always improves the power consumption issue and its power consumption becomes lower from generation to generation. Fortunately, the GPU implementations have higher power efficiencies than the CPU implementation because they have significantly higher computation performances (fps). Therefore, the GPU technology is suitable for implementations that require high computing performance, good power efficiency, and adequate development time.

7.2 Outlook

Although the FPGA-based hardware accelerator implementation on the heterogeneous system for vision-based multi-robot tracking shows the ability to enhance the computation performance, two issues should be considered. First, the interconnection between the FPGA and CPU. Second, the complexity and time- consumption of the development process. The interconnection between the FPGA and CPU has to support a high-speed transfer such as PCIe. Therefore, the PCI interface in the RAPTOR development board must be upgraded to a PCIe interface. Concerning the development process issue, the implementation of the FPGA algorithm using a hardware description language (HDL) requires deep knowledge and special skill related to the FPGA design. One of the solutions regarding this issue could be using SystemC, which provides a fast FPGA implementation with less knowledge of the target system. Unfortunately, this approach currently generates higher resource usage in the FPGA and a lower performance regarding the computation speed compared to a manual approach using hardware description language (HDL) [107].

In contrast to the FPGA-based heterogeneous computing system for vision-based multi-robot tracking, the GPU-CPU interconnection and the development process of the GPU are relatively small issues. This is because most GPUs are already equipped with the PCIe interface and are supported by a steady development tool with an interactive simulation. To increase the computational performance, an additional GPU card can be attached in the PCIe slot of the host PC. The trend of GPU technology is not only increasing the computation performance but also reducing the power consumption. Therefore, the latest GPU technology could be a solution to increase the computational performance with a reasonable power consumption.

List of Figures

List of Tables

Abbreviations

ALU	Arithmetic Logical Unit
AMiRo	Autonomous Mini Robot
APIC	Advanced Programmable Interrupt Controller
AXI	Advanced Exensible Interface
BDC	Binary Distance Calculation
BRAM	Block RAM
CAT	Category of cable
CCD	Charge-Coupled Device
CHT	Circular Hough Transform
CPU	Central Processing Unit
CSW	Circle Scanning Window
CUDA	Compute Unified Device Architecture
D-FF	Data Flip Flop
DBM	Daughter Board Module
DDR2	Double Data Rate version 2
DP-RAM	Dual Port - RAM
DSP	Digital Signal Processor
FIFO	First-In First-Out
FN	False Negative
FP	False Positive
FPGA	Field Programmable Gate Array
fps	frames per second
FPU	Floating Point Unit
FSM	Finite State Machine
GMII	Gigabit Media Independent Interface
GPS	Global Positioning System
GPU	Graphics Processing Unit

HSV	Hue Saturation Value color space
HT	Hyper-Threading
HW/SW	Hardware / Software
I/O	Input/Output
IC	Integrated Circuit
ID	Identity
IO/EC	Input Output Embedded Controller
LAN	Local Area Network
LED	Light-Emitting Diode
LL	LocalLink
LUT	Lookup Table
MC_GigEV	Multi-Camera GigE Vision
MPMC	Multi-Port Memory Controller
NPI	Native Peripheral Interface
OS	Operating System
PC	Personal Computer
PCH	Platform Controller Hub
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PHY	Physical layer
PLB	Processor Local Bus
RAM	Random Access Memory
RGB	Red Green Blue color space
ROI	Region of Interest
ROM	Read Only Memory
SATA	Serial AT Attachment
SDMA	Soft Direct Memory Access
SDRAM	Synchronous Dynamic RAM
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
SSL	Small Size League
TEMAC	Tri-mode Ethernet Media Access Controller
TP	True Positive

TPM	Trusted Platform Module
UDP/IP	User Datagram Protocol / Internet Protocol
USB	Universal Serial Bus
VFBC	Video Frame Buffer Controller
VGA	Video Graphics Array
VHDL	Very High Speed Integrated Circuit Hardware Description Language

References

[1] A. B. Abdallah. *Multicore Systems On-Chip: Practical Software/Hardware Design*. Atlantis Publishing Corporation, 2013. ISBN: 9491216910, 9789491216916.

[2] G. Afonso, Z. Baklouti, D. Duvivier, R. B. Atitallah, E. Billauer, and S. Stilkerich. "Heterogeneous CPU/FPGA Reconfigurable Computing System for Avionic Test Application". In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*. 2013, pp. 260–267. DOI: 10.1109/IPDPSW.2013.111.

[3] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators". In: *25th IEEE International Parallel & Distributed Processing Symposium*. 2011. URL: https://hal.inria.fr/inria-00547614.

[4] M. Alawieh, M. Kasparek, N. Franke, and J. Hupfer. "A High Performance FPGA-GPU-CPU Platform for a Real-Time Locating System". In: Zenodo, 2015. DOI: 10.5281/zenodo.35824.

[5] F. F.-t. Alim, K. Messaoudi, S. Seddiki, and O. Kerdjidj. "Modified circular Hough transform using FPGA". In: *2012 24th International Conference on Microelectronics (ICM)*. 2012, pp. 1–4. DOI: 10.1109/ICM.2012.6471412.

[6] K. Amma, Y. Yaguchi, Y. Niitsuma, T. Matsuzaki, and R. Oka. "A comparative study of gesture recognition between RGB and HSV colors using time-space continuous dynamic programming". In: *Awareness Science and Technology and Ubi-Media Computing (iCAST-UMEDIA), 2013 International Joint Conference on*. 2013, pp. 185–191. DOI: 10.1109/ICAwST.2013.6765431.

[7] Automated Imaging Association (AIA). *GigE Vision - True Plug and Play Connectivity*. URL: http://www.visiononline.org (visited on 05/09/2016).

[8] D. G. Bailey. *Design for Embedded Image Processing on FPGAs*. 1st. Wiley Publishing, 2011. ISBN: 0470828498, 9780470828496.

[9] D. G. Bailey, G. S. Gupta, and M. Contreras. "Intelligent Camera for Object Identification and Tracking". In: *Robot Intelligence Technology and Applications 2012: An Edition of the Presented Papers from the 1st International Conference on Robot Intelligence Technology and Applications*. Ed. by J.-H. Kim, T. E. Matson, H. Myung, and P. Xu. Springer Berlin Heidelberg, 2013, pp. 1003–1013. DOI: 10.1007/978-3-642-37374-9_97.

[10] T. Balch, Z. Khan, and M. Veloso. “Automatically Tracking and Analyzing the Behavior of live Insect Colonies”. In: *Proceedings of the fifth international conference on Autonomous agents - AGENTS ’01* (2001), pp. 521–528. DOI: 10.1145/375735.376434.

[11] D. H. Ballard. “Readings in Computer Vision: Issues, Problems, Principles, and Paradigms”. In: ed. by M. A. Fischler and O. Firschein. Morgan Kaufmann Publishers Inc., 1987. Chap. Generalizing the Hough Transform to Detect Arbitrary Shapes, pp. 714–725. ISBN: 0-934613-33-8.

[12] M. Balzer, M. Birk, R. Dapp, H. Gemmeke, E. Kretzek, S. Menshikov, M. Zapf, and N. V. Ruiter. “3D ultrasound computer tomography for breast cancer diagnosis”. In: *Real Time Conference (RT), 2012 18th IEEE-NPSS*. 2012, pp. 1–4. DOI: 10.1109/RTC.2012.6418198.

[13] D. S. Banerjee and K. Kothapalli. “Hybrid algorithms for list ranking and graph connected components”. In: *2011 18th International Conference on High Performance Computing*. 2011, pp. 1–10. DOI: 10.1109/HiPC.2011.6152655.

[14] B. Bayer. *Color imaging array*. US Patent 3,971,065. July 1976. URL: https://www.google.com/patents/US3971065.

[15] BERTEN-DSP. *GPU vs FPGA Performance Comparison*. White Paper : BWP001 v1.0. May 2016. URL: http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf.

[16] K. Bhaskaran-Nair, W. Ma, S. Krishnamoorthy, O. Villa, H. J. J. van Dam, E. Aprà, and K. Kowalski. “Noniterative Multireference Coupled Cluster Methods on Heterogeneous CPU–GPU Systems”. In: *Journal of Chemical Theory and Computation* 9.4 (2013). PMID: 26583545, pp. 1949–1957. DOI: 10.1021/ct301130u. eprint: http://dx.doi.org/10.1021/ct301130u.

[17] R. A. Bianchi and A. H. Reali-Costa. “Implementing Computer Vision Algorithms in Hardware: An FPGA/VHDL-Based Vision System for a Mobile Robot”. In: *RoboCup 2001: Robot Soccer World Cup V*. Ed. by A. Birk, S. Coradeschi, and S. Tadokoro. Springer Berlin Heidelberg, 2002, pp. 281–286. DOI: 10.1007/3-540-45603-1_31.

[18] M. Birk, M. Balzer, N. Ruiter, and J. Becker. “Comparison of processing performance and architectural efficiency metrics for FPGAs and GPUs in 3D Ultrasound Computer Tomography”. In: *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*. Vol. "" 2012, pp. 1–7. DOI: 10.1109/reconfig.2012.6416735.

[19] M. Birk, S. Koehler, M. Balzer, M. Huebner, N. V. Ruiter, and J. Becker. “FPGA-Based Embedded Signal Processing for 3-D Ultrasound Computer Tomography”. In: *IEEE Transactions on Nuclear Science* 58.4 (Aug. 2011), pp. 1647–1651. DOI: 10.1109/TNS.2011.2159017.

[20] M. Birk, S. Koehler, M. Balzer, M. Huebner, N. V. Ruiter, and J. Becker. "FPGA-based embedded signal processing for 3D ultrasound computer tomography". In: *Real Time Conference (RT), 2010 17th IEEE-NPSS*. 2010, pp. 1–5. DOI: 10.1109/RTC.2010.5750384.

[21] M. Birk, E. Kretzek, P. Figuli, M. Weber, J. Becker, and N. Ruiter. "High-Speed Medical Imaging in 3D Ultrasound Computer Tomography". In: *Parallel and Distributed Systems, IEEE Transactions on* (2015). DOI: 10.1109/TPDS.2015.2405508.

[22] C. Blair, N. M. Robertson, and D. Hume. "Characterizing a Heterogeneous System for Person Detection in Video Using Histograms of Oriented Gradients: Power Versus Speed Versus Accuracy". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 3.2 (June 2013), pp. 236–247. DOI: 10.1109/JETCAS.2013.2256821.

[23] P. A. Blume. *The LabVIEW Style Book (National Instruments Virtual Instrumentation Series)*. Prentice Hall PTR, 2007. ISBN: 0131458353.

[24] J. Bruce. *CMVision Library*. 2002. URL: http://www.cs.cmu.edu/~jbruce/cmvision/.

[25] J. Chai, H. Su, M. Wen, X. Cai, N. Wu, and C. Zhang. "Resource-efficient utilization of CPU/GPU-based heterogeneous supercomputers for Bayesian phylogenetic inference". In: *The Journal of Supercomputing* 66.1 (2013), pp. 364–380. DOI: 10.1007/s11227-013-0911-1.

[26] Z. Chai, H. Zhou, Z. Wang, and D. Wu. "Using C to implement high-efficient computation of dense optical flow on FPGA-accelerated heterogeneous platforms". In: *Field-Programmable Technology (FPT), 2014 International Conference on*. 2014, pp. 260–263. DOI: 10.1109/FPT.2014.7082789.

[27] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach. "Accelerating Compute-Intensive Applications with GPUs and FPGAs". In: *Application Specific Processors, 2008. SASP 2008. Symposium on*. 2008, pp. 101–107.

[28] S. Che and K. Skadron. "BenchFriend: Correlating the performance of GPU benchmarks". In: *International Journal of High Performance Computing Applications* 28.2 (2014), pp. 238–250. DOI: 10.1177/1094342013507960. eprint: http://hpc.sagepub.com/content/28/2/238.full.pdf+html.

[29] R. Chen and V. K. Prasanna. "Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016, pp. 212–219. DOI: 10.1109/FCCM.2016.62.

[30] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. 1st. John Wiley and Sons, Inc, 2014. ISBN: 978-1-118-73932-7.

[31] Chester F. Carlson. *Lecture 10: Hough Circle Transform, Rochester Institute of Technology*. Nov. 2005.

[32] J. Chudoba, J. Faigl, M. Kulich, T. Krajník, K. Košnar, and L. Přeučil. "A TECHNICAL SOLUTION OF A ROBOTIC E-LEARNING SYSTEM IN THE SYROTEK PROJECT". In: *Proceedings of the 3rd International Conference on Computer Supported Education*. 2011, pp. 412–417. DOI: 10.5220/0003341404120417.

[33] K. Compton and S. Hauck. "Reconfigurable Computing: A Survey of Systems and Software". In: *ACM Comput. Surv.* 34.2 (June 2002), pp. 171–210. DOI: 10.1145/508352.508353.

[34] M. Contreras, D. G. Bailey, and G. S. Gupta. "FPGA Implementation of Global Vision for Robot Soccer as a Smart Camera". In: *Robot Intelligence Technology and Applications 2: Results from the 2nd International Conference on Robot Intelligence Technology and Applications*. Ed. by J.-H. Kim, . E. T. Matson, H. Myung, P. Xu, and F. Karray. Springer International Publishing, 2014, pp. 657–665. DOI: 10.1007/978-3-319-05582-4_56.

[35] M. Contreras, D. Bailey, and G. S. Gupta. "Robot Identification Using Shape Features on an FPGA-Based Smart Camera". In: *Proceedings of the 29th International Conference on Image and Vision Computing New Zealand*. IVCNZ '14. ACM, 2014, pp. 282–287. DOI: 10.1145/2683405.2683437.

[36] DoD. *System Engineering Fundamentals*. Ed. by S. M. C. D. of Defense. DoD, 2001.

[37] R. O. Duda and P. E. Hart. "Use of the Hough Transformation to Detect Lines and Curves in Pictures". In: *Commun. ACM* 15.1 (Jan. 1972), pp. 11–15. DOI: 10.1145/361237.361242.

[38] A. Dziekonski, A. Lamecki, and M. Mrozowski. "Tuning a Hybrid GPU-CPU V-Cycle Multilevel Preconditioner for Solving Large Real and Complex Systems of FEM Equations". In: *IEEE Antennas and Wireless Propagation Letters* 10 (2011), pp. 619–622. DOI: 10.1109/LAWP.2011.2159769.

[39] A. Elhossini and M. Moussa. "Memory efficient FPGA implementation of hough transform for line and circle detection". In: *Electrical Computer Engineering (CCECE), 2012 25th IEEE Canadian Conference on*. 2012, pp. 1–5. DOI: 10.1109/CCECE.2012.6335003.

[40] J. Faigl, J. Chudoba, K. Kosnar, M. Kulich, M. Saska, and L. Preucil. "SyRoTek - A Robotic System for Education". In: *Robotics in Education, 2010 International Conference on*. 2010, pp. 37–42. DOI: 10.1109/ICBR.2013.6729272.

[41] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice (2Nd Ed.)* Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0-201-12110-7.

[42] A. Ghorbel, N. B. Amor, M. Jallouli, and L. Amouri. "A HW/SW implementation on FPGA of a robot localization algorithm". In: *Systems, Signals and Devices (SSD), 2012 9th International Multi-Conference on*. 2012, pp. 1–7.

[43] A. Ghorbel, M. Jallouli, N. B. Amor, and L. Amouri. "An FPGA based platform for real time robot localization". In: *Individual and Collective Behaviors in Robotics (ICBR), 2013 International Conference on*. 2013, pp. 56–61. DOI: `10.1109/ICBR.2013.6729272`.

[44] A. Godil, R. Bostelman, W. Shackleford, T. Hong, and M. Shneier. *Performance Metrics for Evaluating Object and Human Detection and Tracking Systems*. July 2014. URL: `http://dx.doi.org/10.6028/NIST.IR.7972`.

[45] T. Graber, S. Kohlbrecher, J. Meyer, K. Petersen, O. von Stryk, and U. Klingauf. *RoboCupRescue 2013 - Robot League Team Hector Darmstadt (Germany)*. Tech. rep. 2013.

[46] P. Grabust. "The choice of metrics for clustering algorithm". In: *Proceedings of the 8th International Scientific and Practical Conference*. Vol. I1. Environment. Technology. Resources, 2011, pp. 70–76.

[47] Z. Guo, J. Han, and J. Chen. "Fast face recognition on GPU". In: *Software Engineering and Service Science (ICSESS), 2015 6th IEEE International Conference on*. 2015, pp. 783–786. DOI: `10.1109/ICSESS.2015.7339173`.

[48] Q. P. Ha, Y.-H. Yu, and N. K. Quang. "FPGA-based cooperative control of indoor multiple robots". In: *International Journal of Advanced Mechatronic Systems* 4.5-6 (2012). PMID: 52220, pp. 248–259. DOI: `10.1504/IJAMECHS.2012.052220`. eprint: `http://www.inderscienceonline.com/doi/pdf/10.1504/IJAMECHS.2012.052220`.

[49] S. S. Hampton, S. R. Alam, P. S. Crozier, and P. K. Agarwal. "Optimal Utilization of Heterogeneous Resources for Biomolecular Simulations". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. IEEE Computer Society, 2010, pp. 1–11. DOI: `10.1109/SC.2010.37`.

[50] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[51] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations". In: *Euro-Par 2010 - Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*. Ed. by P. D'Ambra, M. Guarracino, and D. Talia. Springer Berlin Heidelberg, 2010, pp. 235–246. DOI: `10.1007/978-3-642-15291-7_23`.

[52] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. "Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems". In: *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. 2012, pp. 88–98. DOI: `10.1109/ISPASS.2012.6189209`.

[53] C. Hollitt. "Reduction of computational complexity of Hough transforms using a convolution approach". In: *2009 24th International Conference Image and Vision Computing New Zealand*. 2009, pp. 373–378. DOI: `10.1109/IVCNZ.2009.5378379`.

[54] P. V. C. Hough. "Machine Analysis Of Bubble Chamber Pictures". In: *Proceedings, 2nd International Conference on High-Energy Accelerators and Instrumentation, HEACC 1959: CERN, Geneva, Switzerland, September 14-19, 1959*. Vol. C590914. 1959, pp. 554–558. URL: `http://inspirehep.net/record/919922/files/HEACC59_598-602.pdf`.

[55] B. Hübener, G. Sievers, T. Jungeblut, M. Porrmann, and U. Rückert. "CoreVA: A Configurable Resource-Efficient VLIW Processor Architecture". In: *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*. 2014, pp. 9–16. DOI: `10.1109/EUC.2014.11`.

[56] A. Humphrey, Q. Meng, M. Berzins, and T. Harman. "Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System". In: *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*. XSEDE '12. ACM, 2012, 4:1–4:8. DOI: `10.1145/2335755.2335791`.

[57] W.-m. Hwu, K. Keutzer, and T. G. Mattson. "The Concurrency Challenge". In: *IEEE Des. Test* 25.4 (July 2008), pp. 312–320. DOI: `10.1109/MDT.2008.110`.

[58] O. W. Ibraheem, A. Irwansyah, J. Hagemeyer, M. Porrmann, and U. Rueckert. "A Resource-Efficient Multi-Camera GigE Vision IP Core for Embedded Vision Processing Platforms". In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015. ISBN: 9781467394062.

[59] Intel. *, "Desktop 4th Generation Intel Core Processor Family*. 2015.

[60] Intel. *, "New Microarchitecture for 4th Gen Intel Core Processor Platform*. 2013. URL: `http://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-family-mobile-brief.pdf`.

[61] A. Irwansyah, O. W. Ibraheem, J. Hagemeyer, M. Porrmann, and U. Rueckert. "FPGA-based circular hough transform with graph clustering for vision-based multi-robot tracking". In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2015, pp. 1–8. DOI: `10.1109/ReConFig.2015.7393313`.

[62] A. Irwansyah, O. W. Ibraheem, J. Hagemeyer, M. Porrmann, and U. Rückert. "FPGA-based Multi-Robot Tracking". In: *Journal of Parallel and Distributed Computing* (2017). DOI: 10.1016/j.jpdc.2017.03.008.

[63] A. Irwansyah, O. W. Ibraheem, D. Klimeck, M. Porrmann, and U. Rückert. "FPGA-based Generic Architecture for Rapid Prototyping of Video Hardware Accelerators using NoC AXI4-Stream Interconnect and GigE Vision Camera Interfaces". In: 2014.

[64] J. R. Jen, M. C. Shie, and C. Chen. "A Circular Hough Transform Hardware for Industrial Circle Detection Applications". In: *Industrial Electronics and Applications, 2006 1ST IEEE Conference on*. 2006, pp. 1–6. DOI: 10.1109/ICIEA.2006.257148.

[65] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123814723, 9780123814722.

[66] J. Kleinberg and É. Tardos. "Approximation Algorithms for Classification Problems with Pairwise Relationships: Metric Labeling and Markov Random Fields". In: *J. ACM* 49.5 (Sept. 2002), pp. 616–639. DOI: 10.1145/585265.585268.

[67] M. Kulich, J. Chudoba, K. Kosnar, T. Krajnik, J. Faigl, and L. Preucil. "SyRoTek - Distance Teaching of Mobile Robotics". In: *IEEE Transactions on Education* 56.1 (Feb. 2013), pp. 18–23. DOI: 10.1109/TE.2012.2224867.

[68] F. Lecron, S. A. Mahmoudi, M. Benjelloun, S. Mahmoudi, and P. Manneback. "Heterogeneous Computing for Vertebra Detection and Segmentation in X-ray Images". In: *Journal of Biomedical Imaging* 2011 (Jan. 2011), 5:1–5:12. DOI: 10.1155/2011/640208.

[69] S. Li, X. Liu, M. Mao, H. H. Li, Y. Chen, B. Li, and Y. Wang. "Heterogeneous systems with reconfigurable neuromorphic computing accelerators". In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016, pp. 125–128. DOI: 10.1109/ISCAS.2016.7527186.

[70] W. Limprasert, A. Wallace, and G. Michaelson. "Real-Time People Tracking in a Camera Network". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 3.2 (June 2013), pp. 263–271. DOI: 10.1109/JETCAS.2013.2256820.

[71] D. Liu, R. Li, X. Gu, K. Wen, H. He, and G. Gao. "Fast Snippet Generation Based on CPU-GPU Hybrid System". In: *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. 2011, pp. 252–259. DOI: 10.1109/ICPADS.2011.63.

[72] H. Liu, Z. Li, B. Wang, Y. Zhou, and Q. Zhang. “Table tennis robot with stereo vision and humanoid manipulator II: Visual measurement of motion-blurred ball”. In: *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. 2013, pp. 2430–2435. DOI: 10.1109/ROBIO.2013.6739835.

[73] Y. Liu, A. Fedorov, R. Kikinis, and N. Chrisochoides. “Real-Time Non-rigid Registration of Medical Images on a Cooperative Parallel Architecture”. In: *Proceedings of the 2009 IEEE International Conference on Bioinformatics and Biomedicine*. BIBM ’09. IEEE Computer Society, 2009, pp. 401–404. DOI: 10.1109/BIBM.2009.10.

[74] T. Lochmatter, P. Roduit, C. Cianci, N. Correll, J. Jacot, and A. Martinoli. “SwisTrack - a flexible open source tracking software for multi-agent systems”. In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2008, pp. 4004–4010. DOI: 10.1109/IROS.2008.4650937.

[75] H. H. Lund, E. D. V. Cuenca, and J. Hallam. “A Simple Real-Time Mobile Robot Tracking System”. In: *Technical Paper*. 41. University of Edinburgh, 1996, pp. 1–8.

[76] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal. “Optimizing Tensor Contraction Expressions for Hybrid CPU-GPU Execution”. In: *Cluster Computing* 16.1 (Mar. 2013), pp. 131–155. DOI: 10.1007/s10586-011-0179-2.

[77] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, A. J. Miller, and M. Upton. “Hyper-Threading Technology Architecture and Microarchitecture”. In: *Intel Technology Journal* 6.1 (Feb. 2002), pp. 4–15. ISSN: 00419907.

[78] P. L. Mazzeo, L. Giove, G. M. Moramarco, P. Spagnolo, and M. Leo. “HSV and RGB color histograms comparing for objects tracking among non overlapping FOVs, using CBTF”. In: *Advanced Video and Signal-Based Surveillance (AVSS), 2011 8th IEEE International Conference on*. 2011, pp. 498–503. DOI: 10.1109/AVSS.2011.6027383.

[79] P. Meng, M. Jacobsen, and R. Kastner. “FPGA-GPU-CPU heterogenous architecture for real-time cardiac physiological optical mapping”. In: *FPT*. IEEE, 2012, pp. 37–42. ISBN: 978-1-4673-2846-3.

[80] S. Mittal and J. S. Vetter. “A Survey of CPU-GPU Heterogeneous Computing Techniques”. In: *ACM Comput. Surv.* 47.4 (July 2015), 69:1–69:35. DOI: 10.1145/2788396.

[81] A. Mohanty, N. Suda, M. Kim, S. Vrudhula, J. s. Seo, and Y. Cao. “High-performance face detection with CPU-FPGA acceleration”. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016, pp. 117–120. DOI: 10.1109/ISCAS.2016.7527184.

[82] J.-i. Muramatsu, T. Fukaya, S.-L. Zhang, K. Kimura, and Y. Yamamoto. “Acceleration of Hessenberg Reduction for Nonsymmetric Eigenvalue Problems in a Hybrid CPU-GPU Computing Environment”. In: *International Journal of Networking and Computing* 1.2 (2011), pp. 132–143. ISSN: 2185-2847.

[83] B. Nam, S.-i. Kang, and H. Hong. “Pedestrian detection system based on stereo vision for mobile robot”. In: *Frontiers of Computer Vision (FCV), 2011 17th Korea-Japan Joint Workshop on*. 2011, pp. 1–7. DOI: `10.1109/FCV.2011.5739758`.

[84] Newton Laboratories. *Cognachrome image capture device*. URL: `http://www.newtonlabs.com/cognachrome/` (visited on 05/09/2016).

[85] NVIDIA. *CUDA C Programming Guide*. 2017. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[86] NVIDIA. *NVIDIA’s Next Generation, Fermi*. 2009.

[87] NVIDIA. *NVIDIA’s Next Generation, Kepler*. 2012.

[88] C. Oh, S. Yi, and Y. Yi. “Real-time face detection in Full HD images exploiting both embedded CPU and GPU”. In: *2015 IEEE International Conference on Multimedia and Expo (ICME)*. 2015, pp. 1–6. DOI: `10.1109/ICME.2015.7177522`.

[89] J. Oh, E. J. Im, and K. Yoon. “Optical flow computation on a heterogeneous platform”. In: *Ubiquitous Robots and Ambient Intelligence (URAI), 2011 8th International Conference on*. 2011, pp. 68–73. DOI: `10.1109/URAI.2011.6145935`.

[90] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 3rd. Morgan Kaufmann Publishers Inc., 2007. ISBN: 0123706068, 9780123706065.

[91] M. Porrmann, J. Hagemeyer, C. Pohl, J. Romoth, and M. Strugholtz. “RAPTOR – A Scalable Platform for Rapid Prototyping and FPGA-based Cluster Computing”. In: *Parallel Computing: From Multicores and GPU’s to Petascale, Advances in Parallel Computing*. Vol. 19. IOS press, 2010, pp. 592–599.

[92] F. Rinnerthaler, W. Kubinger, J. Langer, M. Humenberger, and S. Borbely. “Boosting the performance of embedded vision systems using a DSP/FPGA co-processor system”. In: *2007 IEEE International Conference on Systems, Man and Cybernetics*. 2007, pp. 1141–1146. DOI: `10.1109/ICSMC.2007.4413943`.

[93] H. de Ruiter and B. Benhabib. “Colour-Gradient Redundancy for Real-time Spatial Pose Tracking in Autonomous Robot Navigation”. In: *The 3rd Canadian Conference on Computer and Robot Vision (CRV’06)*. 2006, pp. 20–20. DOI: `10.1109/CRV.2006.22`.

[94] O. L. Şavkay, E. Cesur, N. Yıldız, M. E. Yalçın, and V. Tavşanoğlu. “Realization of processing blocks of CNN based CASA system on CPU and FPGA”. In: *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2014, pp. 2081–2084. DOI: 10.1109/ISCAS.2014.6865576.

[95] S. E. Schaeffer. “Survey: Graph Clustering”. In: *Comput. Sci. Rev.* 1.1 (Aug. 2007), pp. 27–64. DOI: 10.1016/j.cosrev.2007.05.001.

[96] E. Serrano, G. Bermejo, J. G. Blas, and J. Carretero. “High-performance X-ray tomography reconstruction algorithm based on heterogeneous accelerated computing systems”. In: *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. 2014, pp. 331–338. DOI: 10.1109/CLUSTER.2014.6968781.

[97] G. Sievers, J. Ax, N. Kucza, M. Flaßkamp, T. Jungeblut, W. Kelly, M. Porrmann, and U. Rückert. “Evaluation of interconnect fabrics for an embedded MPSoC in 28 nm FD-SOI”. In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2015, pp. 1925–1928. DOI: 10.1109/ISCAS.2015.7169049.

[98] A. Sirota. “Robotracker - a system for tracking multiple robots in realtime”. In: *Technical Report*. Technion Israel Institute of Technology, December2004.

[99] M. A. Souki, L. Boussaid, and M. Abid. “An embedded system for real-time traffic sign recognizing”. In: *2008 3rd International Design and Test Workshop*. 2008, pp. 273–276. DOI: 10.1109/IDT.2008.4802512.

[100] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter. “The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures”. In: *Proceedings of the 9th Conference on Computing Frontiers*. CF ’12. ACM, 2012, pp. 103–112. DOI: 10.1145/2212908.2212924.

[101] G. Stockman and L. G. Shapiro. *Computer Vision*. 1st. Prentice Hall PTR, 2001. ISBN: 0130307963.

[102] Sukanya.r, Swaathikka.k, and Soorya.r. “Article: Enhancing Computational Performance using CPU-GPU Integration”. In: *International Journal of Computer Applications* 111.7 (Feb. 2015). Full text available, pp. 18–22.

[103] A. Tanoto, H. Li, U. Rückert, and J. Sitte. “Scalable and flexible vision-based multi-robot tracking system”. In: *2012 IEEE International Symposium on Intelligent Control*. 2012, pp. 19–24. DOI: 10.1109/ISIC.2012.6398261.

[104] A. Tanoto, U. Rueckert, and U. Witkowski. “Teleworkbench: a teleoperated platform for experiments in multi-robotics”. In: *Web-based Control and Robotics Education*. Ed. by S. g. Tzafestas. Vol. 38. isbn: 978-90-481-2504-3. Springer Verlag, 2009. Chap. 12, pp. 287–316.

[105] *The OpenCV Reference Manual*. 2.4.9.0. Itseez. Apr. 2014.

[106] D. B. Thomas, L. Howes, and W. Luk. “A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation”. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’09. ACM, 2009, pp. 63–72. DOI: 10.1145/1508128.1508139.

[107] T. Tiemerding, C. Diederichs, C. Stehno, and S. Fatikow. “Comparison of different design methodologies of hardware-based image processing for automation in microrobotics”. In: *2013 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*. 2013, pp. 565–570. DOI: 10.1109/AIM.2013.6584152.

[108] Y. Umuroglu, D. Morrison, and M. Jahre. “Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform”. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–8. DOI: 10.1109/FPL.2015.7293939.

[109] C. Vömel, S. Tomov, and J. Dongarra. “Divide and Conquer on Hybrid GPU-Accelerated Multicore Systems”. In: *SIAM Journal on Scientific Computing* 34.2 (2012), pp. C70–C82. DOI: 10.1137/100806783. eprint: http://dx.doi.org/10.1137/100806783.

[110] Y. Wang, H. Du, M. Xia, L. Ren, M. Xu, T. Xie, G. Gong, N. Xu, H. Yang, and Y. He. “A Hybrid CPU-GPU Accelerated Framework for Fast Mapping of High-Resolution Human Brain Connectome”. In: *PLoS ONE* 8.5 (May 2013), pp. 1–14. DOI: 10.1371/journal.pone.0062789.

[111] F. Werner, U. Rückert, A. Tanoto, and J. Welzel. “The Teleworkbench: A Platform for Performing and Comparing Experiments in Robot Navigation”. In: *Proceedings of the Workshop on The Role of Experiments in Robotics Research*. 2010.

[112] Wgsimon. *Transistor Count and Moore’s Law*. 2011. URL: https://commons.wikimedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg.

[113] Q. Wu, Y. Ha, A. Kumar, S. Luo, A. Li, and S. Mohamed. “A heterogeneous platform with GPU and FPGA for power efficient high performance computing”. In: *2014 International Symposium on Integrated Circuits (ISIC)*. 2014, pp. 220–223. DOI: 10.1109/ISICIR.2014.7029447.

[114] Xilinx. *Block Memory Generator v8.2*. 2015.

[115] Xilinx. *Introduction to FPGA Design with Vivado High-Level Synthesis*. 2013.

[116] Xilinx. *LogiCORE IP Multi-Port Memory Controller (v6.05.a) Datasheet (DS643)*. 2011.

[117] Xilinx. *LogiCORE IP XPS LL TEMAC (v2.03a) Datasheet*. 2010.

[118] Xilinx. *User Guide: XtremeDSP for Virtex-4 FPGAs*. 2008.

[119] Y.-H. Yu, N. Kwok, and Q. Ha. “Color tracking for multiple robot control using a system-on-programmable-chip”. In: *Automation in Construction* 20.6 (2011). Selected papers from the 26th {ISARC} 2009, pp. 669–676. DOI: `http://dx.doi.org/10.1016/j.autcon.2011.04.013`.

[120] Y.-H. Yu, N. Kwok, and Q. Ha. “FPGA-Based Real-Time Color Tracking for Robotic Formation Control”. In: *Proceedings of the 26th International Symposium on Automation and Robotics in Construction*. ISARC ’09. 2009, pp. 252–258.

[121] C. Zhang, R. Chen, and V. Prasanna. “High Throughput Large Scale Sorting on a CPU-FPGA Heterogeneous Platform”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 148–155. DOI: `10.1109/IPDPSW.2016.117`.

[122] X. Zhou, Y. Ito, and K. Nakano. “An Efficient Implementation of the One-Dimensional Hough Transform Algorithm for Circle Detection on the FPGA”. In: *2014 Second International Symposium on Computing and Networking*. 2014, pp. 447–452. DOI: `10.1109/CANDAR.2014.32`.

[123] S. Zickler, T. Laue, O. Birbach, M. Wongphati, and M. Veloso. “SSL-Vision: The Shared Vision System for the RoboCup Small Size League”. In: *RoboCup 2009: Robot Soccer World Cup XIII. RoboCup International Symposium (RoboCup-09), June 29 - July 5, Graz, Austria*. Ed. by J. Baltes, M. G. Lagoudakis, T. Naruse, and S. Shiry. Vol. 5949. Lecture Notes in Artificial Intelligence, LNAI. Springer, 2010, pp. 425–436.

Author’s Publications

[58] O. W. Ibraheem, A. Irwansyah, J. Hagemeyer, M. Porrmann, and U. Rueckert. “A Resource-Efficient Multi-Camera GigE Vision IP Core for Embedded Vision Processing Platforms”. In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015. ISBN: 9781467394062.

[61] A. Irwansyah, O. W. Ibraheem, J. Hagemeyer, M. Porrmann, and U. Rueckert. “FPGA-based circular hough transform with graph clustering for vision-based multi-robot tracking”. In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2015, pp. 1–8. DOI: 10.1109/ReConFig.2015.7393313.

[62] A. Irwansyah, O. W. Ibraheem, J. Hagemeyer, M. Porrmann, and U. Rückert. “FPGA-based Multi-Robot Tracking”. In: *Journal of Parallel and Distributed Computing* (2017). DOI: 10.1016/j.jpdc.2017.03.008.

[63] A. Irwansyah, O. W. Ibraheem, D. Klimeck, M. Porrmann, and U. Rückert. “FPGA-based Generic Architecture for Rapid Prototyping of Video Hardware Accelerators using NoC AXI4-Stream Interconnect and GigE Vision Camera Interfaces”. In: 2014.