2010-11-17

An Onboard Vision System for Unmanned Aerial Vehicle Guidance

Barrett Bruce Edwards
Brigham Young University - Provo

An Onboard Vision System for Unmanned Aerial Vehicle Guidance

Barrett B. Edwards

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

James K. Archibald, Chair
Dah-Jye Lee
Doran K. Wilde

Department of Electrical and Computer Engineering

Brigham Young University

December 2010

ABSTRACT

An Onboard Vision System for Unmanned Aerial Vehicle Guidance

Barrett B. Edwards

Department of Electrical and Computer Engineering

Master of Science

The viability of small Unmanned Aerial Vehicles (UAVs) as a stable platform for specific application use has been significantly advanced in recent years. Initial focus of lightweight UAV development was to create a craft capable of stable and controllable flight. This is largely a solved problem. Currently, the field has progressed to the point that unmanned aircraft can be carried in a backpack, launched by hand, weigh only a few pounds and be capable of navigating through unrestricted airspace.

The most basic use of a UAV is to visually observe the environment and use that information to influence decision making. Previous attempts at using visual information to control a small UAV used an off-board approach where the video stream from an onboard camera was transmitted down to a ground station for processing and decision making. These attempts achieved limited results as the two-way transmission time introduced unacceptable amounts of latency into time-sensitive control algorithms. Onboard image processing offers a low-latency solution that will avoid the negative effects of two-way communication to a ground station.

The first part of this thesis will show that onboard visual processing is capable of meeting the real-time control demands of an autonomous vehicle, which will also include the evaluation of potential onboard computing platforms. FPGA-based image processing will be shown to be the ideal technology for lightweight unmanned aircraft. The second part of this thesis will focus on the exact onboard vision system implementation for two proof-of-concept applications. The first application describes the use of machine vision algorithms to locate and track a target landing site for a UAV. GPS guidance was insufficient for this task. A vision system was utilized to localize the target site during approach and provide course correction updates to the UAV. The second application describes a feature detection and tracking sub-system that can be used in higher level application algorithms.

Keywords: thesis, BYU, UAV, Robotic Vision Lab, MAGICC Lab, Helios, FPGA, image processing, vision guided landing, feature tracking

ACKNOWLEDGMENTS

I first would like to express sincere appreciation to Wade Fife. Wade was an infinite source of knowledge to my endless stream of questions. He also created the Helios Board, which turned out to be the focus of my graduate education. Much appreciation also goes to Blake Barber, who was my teammate on the UAV vision-guided landing project. Nate Knoebel deserves recognition for acting as a pilot for numerous flight tests. I also must give great appreciation to James Archibald. He always ensured that there was funding for my projects, which meant a lot to me. And for being so patient in waiting for this thesis. Last, but certainly not the least, I must express appreciation to my wife Ashley, for never allowing me to give up.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The dawn of Unmanned Aerial Vehicles (UAV) has come and gone. It is no longer a field with limited application or attention. Without the hampering effects of insufficient technology, mobile vehicles have evolved into advanced sensory platforms. As expendable devices they can be deployed in circumstances unsafe for human life or to unreachable locations. As continual rapid development continues to increase the services that they provide, UAVs will further solidify their permanent place as specialized tools.

However, the technology is still not yet mature. Mobile computing, due to its inherent restrictions, necessitates custom embedded systems with niche strengths and broad weaknesses. Development on such platforms is slow and cumbersome. Each and every new application requires focused tailoring to meet specific system requirements. This specialization comes at a cost. Due to the limited nature of the mobile vehicle computing environment, any tuning of system features or capabilities toward a certain application will limit applicability in others. There will always be more to develop as new applications arise requiring additional functionality. Therefore each new contribution to the field must attempt to not just enable a single use application but to be applicable to the field in general.

1.1 Motivation

The most basic UAV is one that can autonomously navigate unrestricted airspace. This requirement may sound simple, but it has only been achieved through a number of significant developments. It requires a capable airframe, a source of thrust, a global positioning system, accelerometers to determine aircraft attitude, pressure sensors to determine airspeed and altitude, and most of all it requires a tuned autopilot control system. With this component set, this basic UAV can takeoff and fly user controllable waypoints.

The limitation of this basic UAV is that it does not do anything of value. A UAV is a tool that is utilized to accomplish some specific objective, and flying predefined or dynamic waypoints does not provide value. Above and beyond basic aerial navigation ability, a UAV is meant to fulfill two primary functions. First, a UAV can be used to extend the sensory perception of the user by observing the environment and then relaying back specific information. Second, a UAV is a mobile platform that can act in place of the user over a great distance.

1.1.1 Smart Sensor

The primary benefit that a UAV can provide to the user in terms of sensory perception is sight. As remotely controlled aircraft, UAVs can provide a valuable high-altitude vantage point to the user. This may be utilized to provide an eye-in-the-sky view over an area too dangerous or out of reach for human access. Examples of such use would include forest fire perimeter tracking or search and rescue operations. Regardless of the application, sight extension is clearly a valuable service that a UAV can provide.

The quality of such a service can range depending on the technological capabilities of the craft. A simple video camera and radio transmitter can be added to the UAV, which would provide a video feed to the user who could use what was conveyed in the video in decision making. While this may be sufficient for some simple applications, it diminishes the role of the UAV from an autonomous craft to that of simply a remote control flying camera as it cannot act independently from the user. A greater value could be provided if the UAV was utilized as a Smart Sensor, which would require the UAV to have the ability to capture, process, and make decisive judgements with the visual information a camera could provide. Only once a UAV can utilize visual information for decision making without the need for user interaction will it be able to be used to autonomously accomplish any high-level objective.

1.1.2 Delivery Vehicle

In addition to use as a mobile sensory device, a UAV can be used to extend the reach of the user as a delivery vehicle. The size of deliverable payload would depend on the size and load bearing capabilities of the aircraft. Similar to the reasoning behind the value of a

remote sensor, a UAV can deliver a payload to areas unsafe or physically remote from the user. Also as a mechanical vehicle, the UAV can be considered an expendable unit, which may be required in certain payload delivery events.

A remote sensor can only respond to obtained information by controlling the flight path of the aircraft and relaying that information back to the user. A higher ability is for the UAV to act upon information it collects and externally influence the environment around it. An example of such ability would be to deliver food or a radio to a lost or stranded person in a search and rescue operation. This goes over and above just locating the person by extending the reach of the user to the remote location faster than the user could otherwise travel there.

In summary the core service that a UAV must provide to be of any value is to simply sense the surrounding environment. However, to be of greater value it must also be capable of acting upon that perception. What is perceived, and what actions are taken based on observations, will be dependent upon the application. Regardless of what that application is, a UAV will need either one or both of these two capabilities to perform it.

1.2 Problem Description

There is a major obstacle to the general adoption of UAVs. As defined, a smart sensor is one that can perceive and act independently from the user to achieve some high-level objective. This classification requires the UAV to possess sufficient computing power to fulfill its intended purpose. But smart sensors also have to be of physically limited dimensions and comprised of inexpensive components to be considered cost effective.

A large UAV can support powerful but bulky processing equipment, fly farther without refueling, and in general provide greater services to the user. But a large UAV is limited in applicability as it may be physically too large for a person to carry or launch by hand. A small UAV, or Micro Aerial Vehicle (MAV), can provide a light, easily transportable platform. But smaller aircraft have limited payload capacity for sensory processing equipment, which reduces their application scope. This is the tradeoff that constrains UAV development: physical size versus processing ability.

The challenge in UAV development is not to build the larger, flying supercomputer. Achievements in this area are limited only by the amount of fiscal resources that can be thrown at the problem. Rather, the greater challenge is to develop a tool that performs the desired function in the smallest form factor possible. Only with this goal in mind will UAVs be used to their capabilities as smart sensors.

1.2.1 Off-Loading

One major obstacle with small UAV development is the limited amount of computing power that can be contained onboard the aircraft. One solution to this obstacle is to avoid it altogether by off-loading sensory processing to a ground-based computing station. An example implementation of this would be to equip a UAV with a video camera and transmitter and pair that with a laptop and video receiver on the ground. For some applications, this approach would be sufficient as the user may be indifferent to where the sensor data is processed. This approach limits the required onboard devices to sensory and transmission equipment, which may reduce the total physical size of the aircraft.

However, there are limitations imposed by off-loading that may render the aircraft to be insufficient in its ability to achieve the intended behavior. Ground-based processing requires constant radio contact to and from the remote vehicle, possibly requiring a near line of sight transmission path, which may not be realistic for every application. Transmission of raw sensor data would also by definition require a higher bandwidth communication link, which may increase onboard transmission equipment or power consumption more than would be required if the raw data was processed onboard the aircraft. High volumes of raw data transferred over a radio link will be subject to transmission noise, which may complicate ground-based processing that must account for transmission-induced noise in the data. But one of the most significant impacts of off-loading the raw sensor data processing to a ground-based computing station is the introduction of latency.

1.2.2 Latency

The basic system process flow for an autonomous vehicle is to gather sensor data, whether it be GPS location, airspeed, or visual imagery, process that data into a decisive

action, and then implement that action. This cycle is not casually periodic but rather must be completed multiple if not dozens of times every second for the aircraft to maintain vehicle control. In fact, the frequency of this process loop directly influences aircraft stability. This cycle is known as a control loop, which can be extremely sensitive to the affects of latency.

When excessive latency is introduced into a control loop, the physical system may become unstable, progress into violent oscillations, and crash. If latency in the control loop is unavoidable, the vehicle cannot compensate by simply reacting slower to new sensory inputs. A muted response to new information will reduce the agility of the UAV as it cannot rapidly adjust to a changing environment. Therefore, physical systems such as autonomous aircraft that rely on sensory input for vehicle guidance and control must avoid significant sources of latency in their control loops.

The approach of off-loading sensory processing from the UAV is flawed in the fact that it introduces latency. Raw sensor data must be transmitted down to a ground station, processed, a course of action determined, and commands returned to the UAV over another wireless link. With the introduction of this transmission delay, the UAV may not be able to react quickly enough to newly obtained information, which may cause the UAV to fail to achieve its objective. If there is a question whether the UAV would be able to complete its intended objective, it would likely not be deployed in the first place. Consequently the field of vision-guided, autonomous aircraft is lacking in application breadth due to excessive reaction time to newly acquired information.

1.3 Objectives and Applications

It is the objective of this thesis to present a vision system capable of being mounted onboard a small UAV that will enable it to acquire, process, and react to visual sensory data without the need of a ground-based computing station. Central to the completion of this objective is to review current approaches to onboard image processing and to select an ideal computing platform suited for small UAV applications. To demonstrate the effectiveness of the chosen approach, the design methodology, implementation, and successful results of using the chosen approach will be shown in an example UAV application that requires quick visual processing and reaction time.

Two applications were chosen for this demonstration. The primary application is the automated landing of a UAV at a stationary or non-stationary target site. This application was successfully completed and results are included. The secondary application is an investigation into a feature tracking system that is intended to be used as a sub-component for an obstacle avoidance system. Since there are major components of an obstacle avoidance system, such as collision detection algorithms and aerial avoidance maneuvers, which are outside the scope of low-power, embedded image processing, only the image feature tracking sub-component required to support an obstacle avoidance system is presented.

Targeted landing was selected as an example application primarily due to its absolute intolerance of latency and the inadequacy of existing localization technology in succeeding at the task. The basic premise of targeting landing is that a UAV in flight will eventually need to return to earth to refuel or to reach some intended location, which will require the UAV to land at a desired point. While the UAV is in a controlled descent toward the landing location, it must make increasingly rapid course corrections to its flight path to successfully arrive at the intended location. In the case of a moving target landing site, the UAV must continuously detect any change in the landing location and then adapt its flight path accordingly.

Existing efforts in this application area that utilize GPS tracking or off-loaded image processing have proved insufficient. The measured GPS location of the traveling aircraft or the landing site may not be known with sufficient accuracy due to limited sensor technology, which may prevent the aircraft from successfully arriving at the target site. If an onboard image sensor is used to acquire images and transmit them to a ground-based processing station that can analyze the environment and provide course corrections, the latency of the transmission time to and from the ground station may be too excessive to meet the real-time control needs of the increasingly quick adjustments that must be made as the aircraft approaches its target. This application requires a quick vision-based course correction system to enable successful landing.

This thesis will present a vision system that is capable of overcoming the limitations of GPS-based guidance and off-loaded image processing. The approach presented is to use an onboard camera and perform the image processing onboard the aircraft. This approach en-

ables additional sensory perception beyond the limitations of noisy GPS positioners without the added transmission latency of off-board processing.

The feature tracking system application was additionally selected primarily due to its vast applicability as there are a wide number of uses of such a tracking system based on passive video input. Therefore it was included in pursuit of presenting a more complete list of fundamental image processing algorithms that can be used in a variety of applications.

1.4 Outline

This thesis is organized as follows. Chapter 2 describes prior work in the area of targeted landing and image feature tracking. Chapter 3 will provide an evaluation of computing platforms based on their ability to meet the requirements of the application. The system requirements to achieve the chosen applications will be defined and a platform will be selected that meets these requirements. Chapter 4 will discuss image processing methods and architecture implementation techniques on the chosen platform. Chapters 5 and 6 will discuss the exact implementation details of the computing platform that was used to achieve successful results. The results of each application will be included in their respective chapters. Chapter 7 provides a summary of the work, contributions the author has made, and suggestions for future work.

Chapter 2

Background

The technical literature includes several examples of vision-based implementations of autonomous landing. For example, several authors discuss the use of image processing to guide the aerial descent of unmanned hovercraft (e.g., helicopters, quadrotors) onto a landing pad as in [1, 2, 3, 4, 5]. In these applications, vision algorithms are used to estimate the pose of the hovercraft in relation to the landing pad during descent. Adding vision algorithms to the control loop enables precision adjustments to the descent path to be made that provide compensation for GPS position estimation inaccuracies. The results demonstrate that vision-assisted control algorithms are effective in providing the desired course error correction.

Results in [2] show that this approach can be made to work without a predefined and structured landing site marker. However, easily discernible visual target features allow image processing algorithms to be simplified and reduce required computations.

In [5] a landing site marker was used with black and white colored boxes of a fixed size. The onboard vision system found the corners of the boxes, from which the pose of the hovercraft was estimated. Black and white markings were chosen to minimize the effects of varying outdoor lighting conditions. For the vision-guided landing application of this work, a solid color target was used that can be distinguished using color segmentation.

Unlike a hovercraft, a fixed-wing MAV must approach the intended landing site on an angled glideslope with enough forward velocity to maintain flight. Thus, limited time is available for image processing algorithms, which establishes the need for real-time vision computations in this application.

During autonomous landing, neither the direction of approach nor the heading angle at impact are constrained, which implies the markings on the target landing site need not express any orientation information. Ideally, the landing area can be recognized regardless of

approach angle. For this work, a single color cloth target was selected. The important visual information about the target can consequently be summarized by the center of mass (x, y) (in image pixel coordinates) and the size (in number of pixels) of the landmark in each image acquired from the onboard camera. Therefore, the objective of the onboard vision system is to determine the target center of mass (x, y) in each image captured by the onboard camera.

Vision systems for small, autonomous vehicles can be divided into two groups depending on whether computation takes place locally or remotely. Tradeoffs of onboard and off-board computation are discussed in [6]. Crucial to the success of this approach is the choice of onboard computation. While other vehicles have employed local vision processing (e.g., [7, 3, 5]), these were substantially larger rotorcraft. In [7] a PC-104 stack was used and in [5] a 233MHz Pentium embedded system running Linux was used. These embedded platforms were either too large or not powerful enough to serve as the onboard computation platform.

Both [8] and [5] emphasize that they use *off-the-shelf* hardware to perform onboard image processing. They emphasize this design decision to stress the ease of development at the cost of performance and platform size. This work departs from this design goal, seeking instead the most computationally powerful platform for vision-assisted flight control on a small UAV.

Since the early days of reconfigurable computing, Field Programmable Gate Arrays (FPGAs) have been an attractive technology for image processing [9]. The inherent parallelism in typical image processing algorithms and the capability of FPGAs to harness that parallelism through custom hardware make FPGAs an excellent computational match. A tremendous amount of research has explored the use of FPGAs for various vision systems, ranging from simple filtering, to stereo vision systems, to multi-target tracking (see for example [10, 11]).

Unfortunately, much of the vision research on FPGAs has relied on multiple FPGAs placed in large systems as in [12] that cannot fit in small vehicles. However, largely due to Moore's law, FPGA technology has made significant advances since its inception. In recent years FPGA technology has progressed to the point where sophisticated imaging systems

can be implemented on a single FPGA (compare, for example, the 14 FPGAs used in [12] with the single FPGA used in [13] for stereo vision).

Although the performance benefit of FPGAs has been well established for some time, less well known is the relative size and power requirements of modern FPGA-based image processors compared to high-performance, general-purpose CPUs [14]. These characteristics make FPGAs ideal for this work with small autonomous vehicles (see [14, 15, 6, 16]).

Much of the literature has focused on implementing complete imaging systems on FPGAs. The purpose of this work is different in that it aims to deliver a sophisticated, general-purpose, image processing solution in a small, low-power package.

Chapter 3

Platform Selection

Established in Chapter 1 is the premise that UAVs require the comprehension of visual information to fulfill their intended purpose as mobile sensory or delivery devices. The practice of off-loading or remotely processing that visual data was explained to be insufficient at meeting the low-latency requirements of a sensor in the control loop of an autonomous vehicle. If the image data cannot be externally processed in a timely manner then it must be processed onboard the vehicle itself. For this to happen, a mobile computing platform capable of low-latency image processing must be incorporated into the control system of the aircraft.

The purpose of this chapter is to identify a computing platform suitable for onboard image processing. To accomplish this, the evaluation criteria for a mobile image processing platform will first be established in Section 3.1. Possible solutions will then be evaluated according to that criteria, exposing strengths and weaknesses of each particular architecture in Section 3.2. Finally a platform best suited for deployment onboard a small UAV will be selected in Section 3.3.

3.1 System Requirements

The system requirements of a mobile vehicle computing system can be broken down into four areas:

1. Physical size
2. Power consumption
3. Computational ability
4. Development environment

The physical constraints of the image processing platform are driven by the size of the aircraft. The target aircraft for this work is a small UAV that weighs about 5 pounds with a wingspan of about 5 feet. The computational device chosen for the image system must not add an undue burden to the aircraft that would hinder aerial maneuverability. Therefore the vision system must weigh around 1 pound or less. The exact weight down to the ounce is not important to specify as this is only to provide evaluation criteria for potential computing platforms.

The power constraints are also driven by the physical size of the aircraft. As indicated in Chapter 1, the motivation behind this work is to produce a mobile device that can reasonably fly for a few hours at a stretch. Battery power supplies will be limited in this application to what can be carried onboard the aircraft, which implies that the power consumption of the computing system will have an inverse affect on flight time. The more power required by the computing system the shorter the possible flight duration. As a craft that is capable of longer flight durations will be valued more than one that is only capable of shorter durations, a low power computing system should be utilized.

To empirically determine the maximum acceptable power consumption of an onboard computing system, a typical commercially available hobby aircraft battery will be considered. A common battery at the time of this writing is a 12 volt, lithium polymer battery that has a capacity rating of 4 amp-hours, which holds a maximum of about 48 watt-hours. To last a minimum of 3 hours, the image computing platform would therefore be required to consume less than 16 watts. If the specific application requires a longer flight duration, then the platform power consumption threshold would have to be reduced or the battery capacity increased.

The exact computational requirements for the image processing system will vary based on the application. But regardless of the application, image processing is generally computationally intensive. To provide an example, the requirements for the vision guided landing application will be considered. In Sections 5.2.1 to 5.2.3, the following estimates are provided as to the number of raw mathematical computations that are required for a 640×480 image size. The HSV color space conversion would require an estimated 110 million calculations to convert every pixel from the RGB color space to HSV. The color

space segmentation step would require about 65 million calculations to threshold every pixel. While these two steps are performed on every pixel and their exact number of calculations determined a priori, the connected components step cannot be accurately estimated as it depends on the content of the image.

In addition to the first three constraints that center on the runtime environment requirements, the chosen platform must also be well suited as a development platform. As indicated in Chapter 1, the field of vision-based guidance in aircraft is still a developing art. As such, any platform chosen must support a relatively quick development cycle for system designers to design, test, evaluate, and then improve their designs.

While these evaluation criteria will be the primary benchmark to compare possible platforms, some other criteria will also be weighed. For example, the architectures ability to perform computations in parallel, or how easily adaptable is it to new applications will also be considered.

3.2 Platform Evaluation

At the time of this writing there are in general five possible classes of computational devices that could be utilized in an aerial vehicle. Each of these will be individually considered according to the criteria set forth in Section 3.1.

3.2.1 Micro-controller

A micro-controller is commonly implemented as a complete self-contained embedded system. With RAM, PROM, watchdogs, and I/O all built into a single package, it can perform a variety of tasks without extra external equipment, which may reduce the physical size needed for an image processing system. This class of device is often used in small and simple applications, such as toys, automobile sub-systems, and microwave ovens to name only a few. Micro-controllers outsell computer CPUs by roughly a $10:1$ ratio, so availability is abundant and cost is usually low. Power consumption is also low as predictable execution time is emphasized over raw performance.

The most significant drawback to micro-controllers is limited raw performance. An image processing application requires a significant number of mathematical computations

and the current state of micro-controllers cannot provide that level of performance. There is also limited parallelism available as micro-controllers are primarily a single core architecture. Since a micro-controller is computationally limited, it must be ruled out as a potential computing platform for the target applications of this work.

3.2.2 Digital Signal Processor (DSP)

A digital signal processor is basically a micro-controller with specialized computation engines that are optimized for Single Instruction Multiple Data (SIMD) instructions. Since image processing tasks often require the same computation to be performed on every pixel, SIMD instructions can provide a significant performance speed up over single-threaded CPU implementations. These arithmetic units typically excel at multiply-accumulate operations (MAC), which are useful for Fast Fourier Transforms (FFT), matrix operations, and convolutions. Some also utilize a Very Large Instruction Word (VLIW) approach to try and keep multiple arithmetic units busy at the same time.

While DSP architectures are excellent at data level parallelism, they fail to utilize task level parallelism. Each processing step must wait for a turn on the arithmetic units and then use them exclusively until completed. What this implies in image processing is that each mathematical step of the algorithm must be executed sequentially, and there is no ability to have an 'application' pipeline where data flows from one step of the algorithm to the next. Integration with other components required in a mobile image processing system also accentuates this issue. For example, incoming pixel data from a common digital image sensor runs at 54 MHz, which implies that a new pixel would arrive from the camera about every 20 ns, interrupt the DSP, which would then store the pixel in memory since there would be no direct DMA from the camera to main memory. While a DSP would likely have the computational ability for individual image processing tasks, the limited task-level parallelism would make a mobile image processing platform based on a DSP challenging.

3.2.3 Graphics Processing Unit (GPU)

Recently, GPU design has changed from a single direction pipeline to a more modular architecture that can be programmed for General Purpose GPU computing (GPGPU)

through C programming extensions such as the Compute Unified Device Architecture (CUDA) from nVidia. These extensions allow non-graphics applications to utilize the processing power of GPUs for other uses. As a computation device, a GPU is simply a massive array of floating point arithmetic units fed by a wide memory bus. Since GPUs were originally designed to perform image processing algorithms, total device performance is equal to or better than all of the other available architectures. The programming model is also well defined and interfaced in C style commands. This only requires the user to alter single-threaded algorithms to match the parallel architecture of the GPU, which is likely easier than translating it into a completely new language such as VHDL.

The performance of a GPU implementation comes at a high cost, and that is power consumption. The massive number of floating point units draw staggering amounts of power that in some graphics cards reaches into the hundreds of watts. Power consumption this high also requires mechanical cooling devices to dissipate heat. The required power consumption alone limits a GPU-based image processing platform from practical deployment onboard a small UAV. One additional limitation of a GPU-based system is that a GPU cannot operate alone as it needs a regular CPU to issue commands. So any mobile implementation would require not only a power hungry GPU but an additional power consuming CPU just to issue and monitor commands.

3.2.4 Field Programmable Gate Array (FPGA)

The primary strength of FPGAs is that they can be reconfigured to perform various tasks, which enables them to serve well in development environments. The main computational strength of an FPGA is parallel processing that is inherent to the architecture. Commonly, an embedded CPU core within the FPGA is used to run software algorithms, while the reconfigurable fabric is used for accelerator cores that perform specific functions. In image processing applications, individual process steps may not be completed on an FPGA as fast as a GPU or DSP could complete them, but multiple algorithm steps can be overlapped in time through task-level parallelism. Power consumption on FPGAs is not the lowest of the possible architectures, but it can be controlled by the complexity of the design and can fit within the 16 watt limit estimated in Section 3.1. An FPGA will not be as small

as an ASIC implementation due to raw die size; however, similar to an ASIC an FPGA can implement internally some functionality that would otherwise require discrete external components, which helps limit the total physical size of the entire computing system (e.g., internal FPGA Block RAM (BRAM) could be used in place of an external SRAM).

The downside of FPGA development over pure software development is time. FPGA hardware has a different programming model and is generally viewed as more difficult to program, verify, and debug. The reconfigurable fabric in FPGAs is also generally clocked at a slower frequency than other architectures, which reduces the potential performance of any given single atomic computation such as a multiply or addition operation. Even though an FPGA-based computing platform is not the best at any individual selection criteria as stated in Section 3.1, it provides an acceptable level of performance in each area. Most importantly, there is no critical flaw that excludes FPGAs from use on a small UAV.

3.2.5 Application Specific Integrated Circuit (ASIC)

An ASIC offers the best performance in the first three selection criteria. A custom-made piece of silicon can, within reason, be designed to include almost any computational ability from fully functional embedded CPUs, to specialized computational units, to I/O modules, and customized interconnect. Performance of any implemented image processing algorithm can be precisely tailored to match exactly the real-time needs of the application. Power consumption will also be as low as any other solution here considered since the hardware included is custom tailored for the intended application. An ASIC will also be the smallest physical system as the entire system may be integrated onto a single component package.

The critical limitation with an ASIC design is cost. ASIC development requires extensive design and verification techniques to be performed that will add to development time. Once laid out, the hardware of the ASIC is also not adaptable beyond changing software code on any embedded CPUs. Even though it would rank the highest on the first three selection criteria, these limitations essentially eliminate the use of an ASIC as a development platform in the applications targeted in this work.

3.3 Chosen Solution

Based on the selection criteria stated in Section 3.1 and the individual platform evaluations in Section 3.2, an FPGA-based platform was chosen as the platform for the target applications of this work. As stated in the platform evaluation, FPGA-based platforms are not the highest performing, lowest power, or smallest physically of the available options. However, FPGAs do not have a critical flaw that eliminates their use in light-weight aircraft and instead are a sufficient mix of the selection criteria.

The onboard vision system utilized in this work was the Helios robotic vision platform developed at Brigham Young University [14] which is shown in Figure 3.1. Even though all the use cases described in this work are for image processing applications, Helios is in fact a general purpose FPGA-based development platform. The main component of the Helios platform is a Xilinx Virtex-4 FX FPGA, which contains up to two 400MHz embedded PowerPC processors. This specific FPGA is ideal for embedded image processing as it provides both a CPU for standard C code execution and reconfigurable FPGA fabric for hardware-accelerated algorithm implementation.

Other additional peripheral components include a 32 MB SDRAM that serves as the main memory for the system, while a 4 MB SRAM is available to be used as a local memory cache for custom image processing tasks. A 16 MB flash memory is also included. A USB 2.0 port provides high-speed data transfer, including live video to a desktop computer during system development and debugging. A RS-232 serial port provides low-speed connectivity, which realistically is too slow for image transfers, but is perfectly satisfactory for debugging or for a command interface.

The Helios platform was designed with the intention to be as flexible as possible. Therefore the system design was constructed of a main board with all the primary high-speed devices, and interchangeable daughter cards that can be customized for any given application. A 120-pin expansion header is used to connect the Helios main board to specialized daughter boards. Daughter boards are essentially I/O boards with device-specific connectors for cameras, servos, communication devices, etc. As a computing platform, it serves as a rich development environment with a small physical footprint (2.5" x 3.5" at 37 grams) that is well suited for mobile image processing applications.

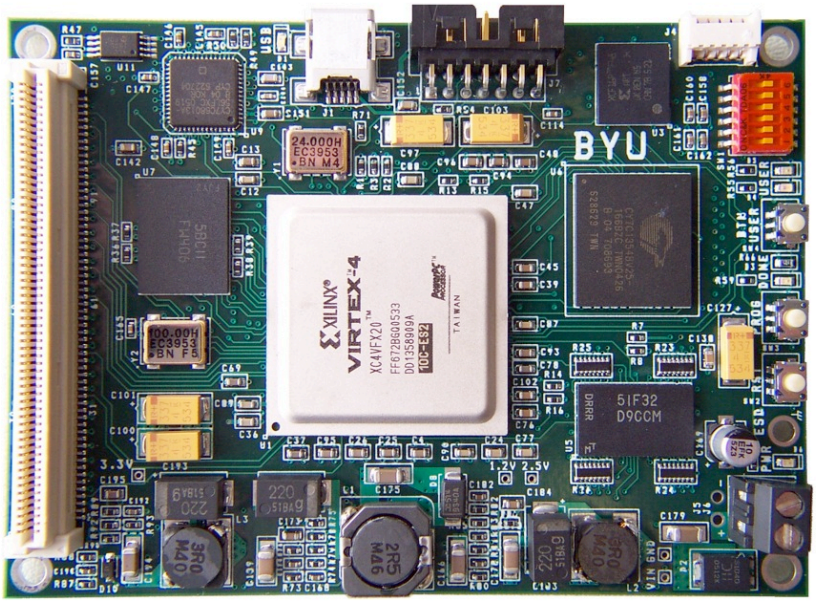

Figure 3.1: Helios robotic vision platform

This work was a joint effort between the Robotic Vision Lab and the Multiple Agent Intelligent Coordination and Control Lab (MAGICC Lab) at Brigham Young University. The author was responsible for the implementation of the vision system and Blake Barber from the MAGICC lab was responsible for adapting the flight control algorithms of the existing Kestrel autopilot. Those control algorithms will not be discussed here, but the aircraft that was used in these applications is shown in Figure 3.2. 'Ray' as this aircraft was called is an all electric mono-wing design constructed out of a styrofoam core that is wrapped in kevlar. The leading edge of the body was modified to provide a viewport that could house the Helios board and a fixed mounted camera.

With the computing architecture chosen as an FPGA-based platform, the next step in the process is to evaluate the given techniques that can be used for real time image processing.

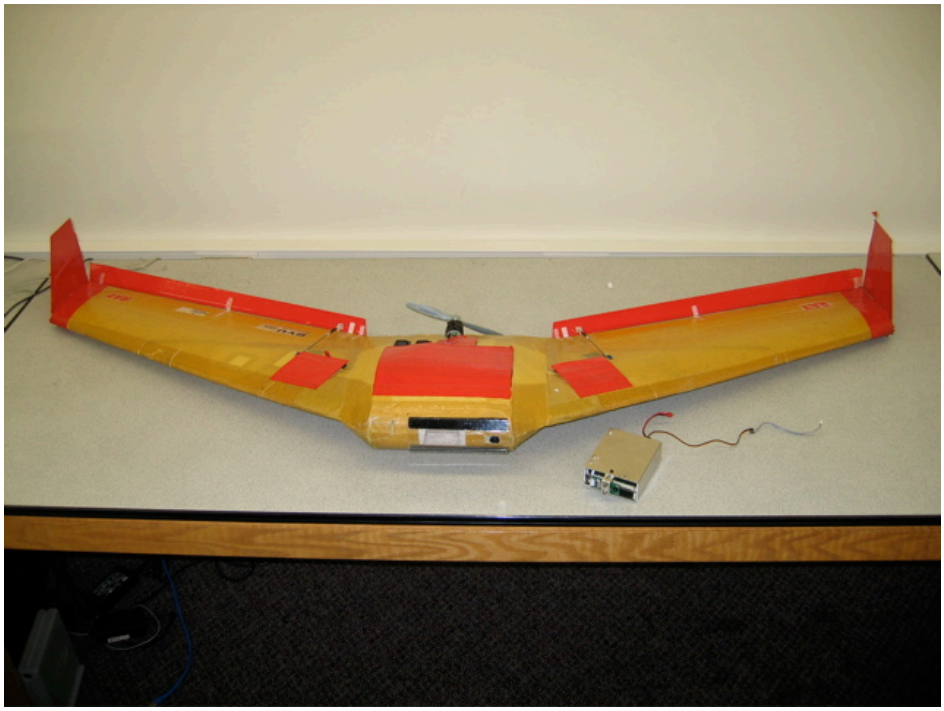

Figure 3.2: Micro air vehicle test platform

Chapter 4

Architecture

All architectures have constraints. This is true no matter the computing platform being considered. Limitations of the system play a major role in the design and implementation of an application. They set the bounds of what the system designer can utilize to fulfill desired functionality. The constraints of the system also limit the methods and techniques available for the system implementation.

All architectures also have strengths. The unique abilities of different architectures are what distinguish one from the next. Some are highly parallel. Others are functionally specialized with high performance. Regardless of the platform, it will have distinct abilities that must be utilized in the methods and techniques used in the system implementation.

The purpose of this chapter is to identify and illustrate the methods and techniques available for real-time image processing on an FPGA-based computing platform. The outline of this chapter is as follows. Section 4.1 describes the basic refinement principle that image processing must accomplish. Section 4.2 reviews foundation level information that must be understood when designing a system that processes digital images such as shutter modes, sensor design, and image data formats. Section 4.3 provides an example-based description of the three techniques used to implement image processing algorithms on FPGAs, which are inline, offline, and cached inline.

4.1 Process Flow

Video images contain a near infinite amount of information available for interpretation, which also means that in raw form, video images are essentially useless. To be of any value, a system that processes video images must capture raw images, analyze them, and produce useful, actionable information. The vision system must be thought of as a process

that refines large quantities of raw data and extracts specific information. The summarized data must therefore be physically less data, possibly even a single value, to be useful in decision making.

To demonstrate this process, a sample application will be considered. A common need in image processing is to find the location of an object in an image. The theoretical number of potential images can be determined by computing the *variant space* of the incoming image. An 8-bit color scale for each of the R, G and B channels would produce a 24-bit color space for each pixel in the image. A standard definition size image of 640×480 contains 307,200 pixels, which results in a variant space of

$$\left(2^{24}\right)^{(307,200)}.$$

The result of the image processing required for this work is a single numeric pair (x, y) of 10-bit values, which has a variant space of

$$2^{(10\times 2)}.$$

The resulting reduction in data is effectively infinite.

However, a vision system cannot summarize all information. With a near infinite input vector space the potential number of output vectors are also infinite. The system must focus on specific things to look for and specific results to summarize. And to be applicable to a real-time environment, the process flow must be continually run on a never-ending stream of images.

4.2 Camera Basics

Before discussing the details of image processing techniques, the characteristics of the data input need to be understood. Regardless of the silicon or optical technology used, all cameras function essentially the same. They expose a 2D silicon array of transistors to light, measure the amount of light received in a given period of time, convert that measurement into a numerical value for each pixel, and transmit the pixel values out over a data bus. Due to practical considerations, there is not a data port for each pixel off of the image sensor.

Therefore, pixel data does not come out all at once, but rather it comes out slowly over a serial bus.

Image sensors come with a few different shutter modes. The most common are: progressive, interlaced, and global. Progressive cameras expose one row of pixels of the image sensor at a time. Interlaced sensors expose all the odd or even rows at the same time, and global shutter cameras expose the entire frame at once. The primary difference in the construction of these sensors is the amount of RAM built into the sensor.

4.2.1 Progressive Scan Shutter

Progressive scan image sensors are the most basic of the three types. They have only a single row buffer built in, which implies that only a single row of pixels can be 'captured' at a given time. Progressive scan sensors expose each row in the array for the same amount of time, but begin the exposure of each row at different times, with the separation in the exposure about equal to the amount of time required to clock out the prior row.

The most significant aspect of progressive scan image sensors to machine vision applications is the impact of frame skew. Since the image sensor captures each row at separate times, there is a time delta between the capture of the first row and the last row of the image. If the scene is near static, implying there is little horizontal, vertical, or rotational motion in the scene, then the impact of frame skew will be negligible. If there is motion in the scene, any subject items in the image will be 'skewed.'

An example of frame skew can be found in Figures 4.1 and 4.2. Figure 4.1 shows a picture of the Washington monument without any frame skew, which implies that it was captured while the camera was stationary. Figure 4.2 depicts what would be recorded if the camera panned to the right during capture. It can be seen that the monument 'leans' to the right, because the monument moved to the left in the camera's field of view as the camera panned to the right. The upper rows of the image represent what was in the field of view of the camera when the frame capture began, and the bottom rows represent what was in the field of view when the frame capture ended. While the skew in these figures is exaggerated for demonstration purposes, the impact of frame skew on image processing algorithms must be considered when evaluating a progressive scan camera for any mobile vehicle application.

Figure 4.1: Washington monument with no frame skew. Camera was stationary during capture.

Figure 4.2: Washington monument with frame skew. Camera panned to the right during capture.

4.2.2 Interlaced Shutter

Interlaced cameras were developed with the intent to convey motion more fluidly to the human eye. This was achieved by doubling the frame rate but only capturing and transmitting half of the total image each cycle. The single image is split into fields of the odd and even rows of the image. During a single field exposure, all of the odd or even rows in the image are exposed and captured at the same time. The two separate fields are then assembled into a single image. And due to the fact that the single composite image is comprised of two exposures at separate times, the motion is blurred and is perceived by the human eye as more fluid.

While an interlaced image conveys motion more fluidly to the human eye, it, however, is problematic for machine vision applications. For example, if there is a moving object in an image, it will be centered at one location in the odd rows of the image, and a different location in the even rows of the image. The odd and even fields are essentially two separate images, each of which has a vertical resolution of 1/2 of the total composite image with a 1 pixel vertical shift between the two. Therefore, when running some image processing

algorithms, the two would have to be processed separately, or one of the two fields would have to be ignored. In either case, the result is a source image that has a 50% reduction in vertical resolution, which skews the pixel aspect ratio from square to rectangular.

4.2.3 Global Shutter

Global shutter cameras expose all pixels in the image at the same time. Every cycle, the camera resets all of the photon sensing transistors and starts capturing light. After the exposure time has elapsed, the level of each pixel is captured and converted to a digital value. Then the image is 'read out' pixel-by-pixel, and row-by-row, just like progressive and interlaced cameras.

These are less common in consumer applications as the video is perceived to be 'jerky' when compared to a progressive or an interlaced camera. But in machine vision applications, a global shutter camera is the ideal camera type. With the entire image exposed at the same moment there is no frame skew, as demonstrated in Figures 4.1 and 4.2. There is also no reduction of vertical resolution found with interlaced images. Each image is a representative sample 'snapshot' of the scene, which can be considered to be exactly what was seen at a single moment in time.

But a global shutter does not eliminate all the affects of motion. Since the entire image is exposed simultaneously, any motion in the scene will be 'blurred' in the captured image. This blurring effect is accentuated by faster movement of objects in the scene, or by longer exposure times. Therefore, to minimize the blurring effect of a moving scene, the global camera should be operated at the highest possible frame rate (i.e., shortest exposure time) that the processing system can sustain.

4.2.4 Image Sensor Design

Color values in images are typically represented as the combination of the brightness of the red, green, and blue channels at that point in the image. Therefore a color pixel is defined by the data set (R, G, B). However, image sensors do not capture all three of the R, G, and B channels at each pixel. In fact image sensors are capable of capturing only one color channel at each pixel location.

An image sensor is an array of color filers placed on top of transistors similar to the graphic shown in Figure 4.3. Each block in Figure 4.3 represents a single transistor, which will only measure the amount of light received through the color filter as shown in Figure 4.4. There exist several variants of this pattern, which utilize filters of different colors or arrangements of the filters on the transistor array, but the 'RGBG' Bayer pattern shown in Figure 4.3 is the most basic and will suffice for this discussion. Regardless of the array pattern used, the consequences are still the same: the image sensor will not produce all three color channels for each pixel. Bayer pattern images do not have distinct (R, G, B) values for each pixel and must be converted into an image that has an (R, G, B) value set for each pixel. This process is called 'demosaicing' and will be discussed in an example later in this chapter.

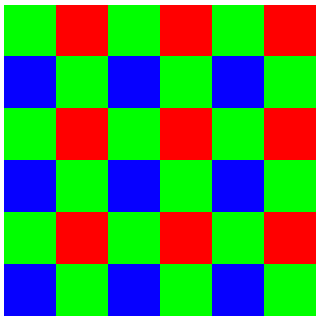

Figure 4.3: Bayer pattern

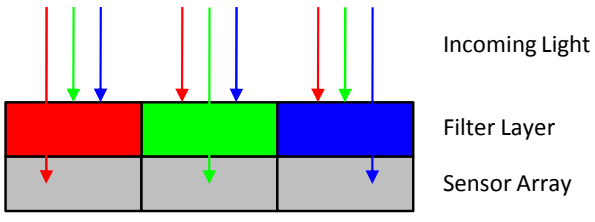

Figure 4.4: Bayer pattern profile

4.2.5 Data Formats

There exist many data formats that can be used to represent an image. However, in the field of mobile image processing applications only a few of the possible image data formats are produced by mobile image sensors. Each has advantages for specific applications and these will briefly be discussed here. The image formats that are commonly produced by small image sensors are:

1. Grayscale
2. Bayer
3. RGB 888
4. RGB 565
5. YUV

Grayscale

Grayscale images, which are often mistakenly called 'Black and White' images, are the simplest image format. The pixel values in the image represent the brightness or the total amount of light received at that point. Common value ranges for grayscale images are 8-bit and 10-bit, which allow a maximum of 256 and $1,024$ different brightness levels respectively. Grayscale is often useful for algorithms that compare two images together or search for image features such as the Harris feature detector algorithm which will be discussed in Chapter 6.

Bayer

Bayer pattern color images as discussed previously are essentially useless in their original form. They only exist as an intermediate format. Even though most cameras can output the captured image in this format, it must be converted into another format before any image processing can be performed.

RGB 888

The RGB 888 format implies that each pixel in the image is defined by the value set (R, G, B), and each of those channels are 8-bit values. This is a very common image

representation. For example, Bitmap images (with a .BMP extension), use this format. Since the RGB 888 format is comprised of three 8-bit channels the total color space that is representable by each pixel is 2^{24}, which is commonly called a 24-bit image. This format provides the most accuracy in image processing as it is the highest quality that can be obtained from an 8-bit color image sensor.

RGB 565

The RGB 565 format is a reduction from the RGB 888 format. The purpose of this reduction is to reduce memory requirements from 3 bytes per pixel down to 2 bytes per pixel. The 565 definition specifies that 5-bit values are used to store the R and B channels, and 6-bits are used to store the G channel. The green channel gets the extra bit because the human eye is more sensitive to green light than red or blue. Conversion from RGB 888 to RGB 565 can be achieved simply by dropping or rounding the last 2 or 3 bits of the 8-bit values.

YUV

The YUV color format was invented as a way to add color information to the grayscale format without breaking backward compatibility with devices that could only receive grayscale images. The Y or luma channel represents the brightness of the pixel, while the U and V channels represent blue and red chrominance values. The Y component is comparable to the simple grayscale format, while the chrominance values are difference values that represent how much brighter or darker the blue and red channels are from the total pixel brightness. Many cameras are capable of delivering this format as it is used in NTSC video transmissions for standard definition televisions. Unless the image processing application requires the transmission of video in an NTSC format, this format is not useful in image processing algorithms. This is primarily due to the fact that many image processing algorithms are based on (R, G, B) values.

4.3 Techniques

An FPGA-based computing environment provides unique abilities that enable multiple image processing techniques to be utilized. The computational structure of an FPGA is one massive fabric of individual computational elements that can be connected and configured in essentially limitless ways. The elements can be connected in series to form an arithmetic pipeline, grouped together into processing units, or function as memory buffers. However, the most significant advantage of the fabric, as it pertains to image processing, is not just the configurability, but the ability to have system components operate in parallel.

Real-time image processing is essentially performing the same algorithmic operations again-and-again on a continuous series of images. Processing video image data can therefore be classified as 'stream processing.' Regardless of the data format, the techniques for embedded image processing operations can be grouped into the following categories:

1. Inline
2. Offline
3. Cached Inline

These classifications were not found in the existing literature at the time this work was performed. The author defined these classifications while reflecting on the general design structure of the multiple FPGA logic components that were developed as part of this work. The intent of listing them here is to provide the reader with a framework to guide future FPGA logic development. It is important to note that the image processing techniques listed here only apply to the algorithms used to reduce the information contained in raw images to specific, actionable information, and not the tracking or decision making processes that would use that specific information in a mobile autonomous vehicle.

4.4 Inline

Inline processing is the most basic of the image processing techniques. It is also the fastest. All data processing operations are organized into a single pipeline. Image data is pushed through the pipeline one pixel after another. A diagram of the computational architecture is shown in Figure 4.5.

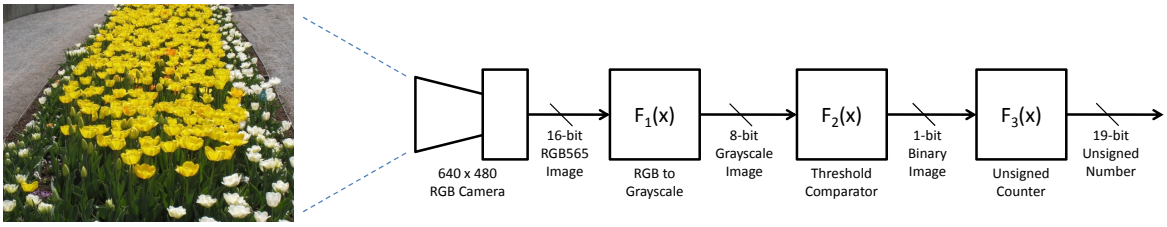

Figure 4.5: Inline thresholded counter

As shown in the diagram, pixel data is clocked out of the camera 1 pixel at a time as a continuous stream. Each pixel is then clocked through the computational pipeline at a rate of 1 pixel per clock cycle. This can be viewed as pre-processing since the image data is passed through a processing engine prior to being stored in memory. The output of the inline processing unit can be a pixel stream of the same dimensions as the input, or it could be as simple as a single value. Example applications that can be implemented using the inline technique are those with independent pixel operations where each pixel can be processed without looking at adjacent pixels, such as color space conversion, thresholding, and segmentation.

4.4.1 Example: Threshold Counter

To demonstrate an inline processing unit, consider a simple application that counts the number of pixels in an image that are brighter than a threshold value. This application requires the conversion of the input image stream from the RGB color space into grayscale, the detection of whether the pixel is brighter than a threshold value, and the counting of the number of pixels above that value. This can all be accomplished using the inline processing approach.

Figure 4.5 includes a source camera which outputs a 640×480 image in 16-bit RGB 565 format. The image captured from the camera is clocked out one pixel per clock cycle over the camera's data bus. The speed of that bus is not relevant for this example, as it only

focuses on the flow of data. Following the camera are three computational pipeline stages labeled: $f_1(x)$, $f_2(x)$, and $f_3(x)$.

The first computational stage, $f_1(x)$, converts the incoming 16-bit color RGB 565 stream into grayscale. This is required to determine the overall brightness value of a single pixel. This is accomplished using

$$Y = 0.3 \times R + 0.59 \times G + 0.11 \times B \tag{4.1}$$

where Y is the grayscale value and R, G, and B are the individual red, green and blue values

This algorithm does not depend on any future or prior values and only requires the data of each pixel to compute the grayscale brightness at each pixel and therefore meets the definition of an inline processing unit.

The second stage, $f_2(x)$, performs the thresholding. If the 8-bit grayscale value coming out of the first stage is above the threshold value then the $f_2(x)$ stage outputs a binary 1, otherwise it outputs a 0 according to

$$Result = \begin{cases} 1, & \text{if } Y > Value \\ 0, & \text{if } Y <= Value \end{cases}. \tag{4.2}$$

As this equation only requires the comparison of the incoming pixel value with a static value, it meets the criteria of an inline processing unit.

The third stage, $f_3(x)$, counts the number of pixels that are above the threshold. Since the input image dimensions are 640×480, the total pixel count in the image is $307,200$. Theoretically, every pixel in the image could have a grayscale brightness value above the threshold, and therefore the counter would need to be able to count up to at least $307,200$. For an unsigned binary counter, this implies a minimum of 19 bits. The counter in $f_3(x)$ is incremented each time the output of $f_2(x)$ is a binary 1 and is reset to zero at the start of each image.

In this example, a hypothetical scenario was used to demonstrate the inline processing design technique. The advantage of this technique is speed. Each stage only adds the latency required to implement its arithmetic function. And each of the three stages in this example

design can be implemented in FPGA logic that only requires one or two clock cycles to complete. Therefore, these pipeline stages would implement the thresholded counter with the addition of only a few clock cycles of latency.

The disadvantage to the inline approach is that it can only be used to implement a limited set of algorithms. Each processing unit can only look at a single pixel at a time in the order that they were received. There can be no re-ordered or random access to the stream. Therefore, the applications that it can implement must be simple operations that can be considered 'pre-processing.'

4.5 Offline

The offline technique requires that the whole image be captured and stored in memory *before* processing begins. The 'offline' designation is meant to imply that the processing is not directly connected with the image data pipeline that streams pixel data out of the camera. Since the entire image is captured into memory before processing begins, the algorithms utilized can access any part of the image at random as there is no constrained sequential access pattern. Therefore the typical process flow is to capture the complete image to some form of memory and then turn that memory space over to a processing unit that accesses pixel data as needed.

Offline processing is the method used by CPU-based image processing systems. In these systems, an image is first captured from a camera device through some type of 'frame grabber.' This device interfaces with the camera, captures a full image streaming out of the camera, and transfers it to main memory. The image is then handed over to the program running on the CPU, which can access the entire image at random.

The major advantage of the offline method is the ability to access at random any data in the image. This allows for selective analysis of a limited portion of the image that would take too much time if attempted on every pixel location in the image or iterative access to pixel locations that are not known until runtime. The most significant drawback of offline processing is the requirement to wait until the full image is captured before processing can begin. This imposes additional latency equal to the full capture time of the image, which may impact real-time decision making in a vision-guided vehicle.

4.5.1 Example: Image Normalization

To demonstrate the offline processing technique, an image normalization implementation will be considered. Normalization is a technique to more evenly distribute the pixel values in the image across the possible range of pixel values. This is sometimes referred to as contrast stretching or dynamic range expansion. The main goal of this algorithm is to correct the affects of bad lighting in images.

In visual imagery, this is used to 'brighten' under-exposed images or 'darken' over-exposed images, which allows the human eye to perceive more of the image content. This prevents areas that would be 'too dark' or 'too bright' to be meaningful. In machine vision algorithms such as those used on a mobile vehicle, the justification is slightly different. Normalization of images in a video stream helps minimize lighting variation between images. Maintaining consistency in the relative spectral distribution in video stream images aids image analysis algorithms by reducing the amount of variation that needs to be tracked and compensated. For example, image processing algorithms that utilize static thresholding, such as the example described in Section 4.4.1, would require an input image stream that is not adversely affected by lighting changes.

Figure 4.6: Original cave image. Left portion is too dark to perceive content

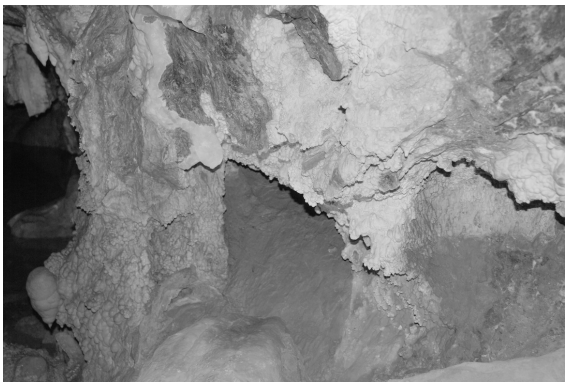

Figure 4.7: Normalized cave image. Left portion is now plainly visible

Shown in Figure 4.6 is an unmodified image taken inside a cave. As can clearly be seen, the image is dark and some of the content, particularly the left side of the image, is not

discernible. The histogram of this image is shown in Figure 4.8, which also shows that the image is skewed toward the darker side of the spectrum. This dark-shift of the image was caused by the use of insufficient lighting equipment when the image was taken. While there is no method to reconstruct image data that was not captured, the image can be normalized to shift the content that is too dark to discern into a more visible spectrum.

Figure 4.8: Histogram of original image. Pixel brightness is skewed and not evenly spread across spectrum.

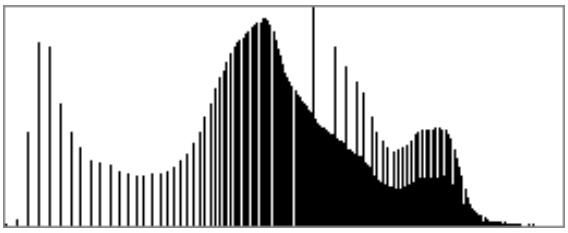

Figure 4.9: Histogram of normalized image. Pixel brightness is evenly spread across the spectrum.

Figure 4.7 shows the cave image with image normalization applied. As can been seen in the image, the once 'dark' areas of the image, which contained imperceptible content, are now plainly visible. Also, the histogram shown in Figure 4.9 shows that the image content has been spread more evenly across the total grayscale spectrum. The histogram does show gaps which are a result of the linear spreading algorithm. If desired, these can be eliminated through the use of a slight blur filter.

The algorithm to implement image normalization can vary in complexity from simple linear scaling to complex histogram distribution functions. In this example implementation, a simple linear scaling algorithm will be considered. Prior to computation, the maximum and minimum grayscale pixel values in the image need to be computed. These are utilized in

$$N(x,y) = (I(x,y) - MIN) \times \left(\frac{255}{Max - Min}\right) \tag{4.3}$$

where $N(x,y)$ refers to a pixel at location (x,y) in the normalized image, and $I(x,y)$ refers to a pixel at location (x,y) in the original image.

In this algorithm, all the pixel values are shifted down by the minimum value and are then scaled up to fill the entire 255 value range in typical 8-bit grayscale images. For example if the image only contains pixel values in the range $[30-200]$, then every pixel value is first shifted down by 30, which results in a range of $[0-170]$. Then the image is scaled to fill the available range by multiplying each pixel value by $\frac{255}{170}$, which results in a full range of $[0-255]$.

The FPGA circuit implementation of the image normalization algorithm is shown in Figure 4.10. In this implementation the image data that is streamed out of the camera is stored in some form of memory. In addition to being streamed to memory, the image pixel data is fed through inline pre-processing blocks to determine the maximum and minimum values.

Once the entire image has been captured into the memory, the actual processing begins. First the maximum and minimum values are pushed through a subtractor to compute the $(Max-Min)$ parameter, which is then pushed into a fixed numerator divider unit that computes the $\frac{255}{(Max-Min)}$ parameter. The image is then read out of memory one pixel at a time, and is fed through the arithmetic blocks. The pixel data first passes through the subtractor which computes the $(pixel-Min)$ parameter, which is then passed into the multiplier unit which computes the final normalized value.

In review, this example was intended to demonstrate the offline processing technique. In this technique, the entire image was captured to memory before processing began. The normalization algorithm requires the entire image to be fully read out of the camera before beginning the normalization conversions, because it needs to find the maximum and minimum pixel values in the image, which may possibly be the last pixel in the image. As previously indicated, completely capturing the image to memory adds at a minimum latency equal to the time required to completely clock the image out of the camera.

4.6 Cached Inline

Cached inline processing is the next step up in complexity from pure inline processing. Computation is still performed in a pipeline, as in inline processing, but the functional units can store prior data that has passed through the pipeline and use that data in implemented

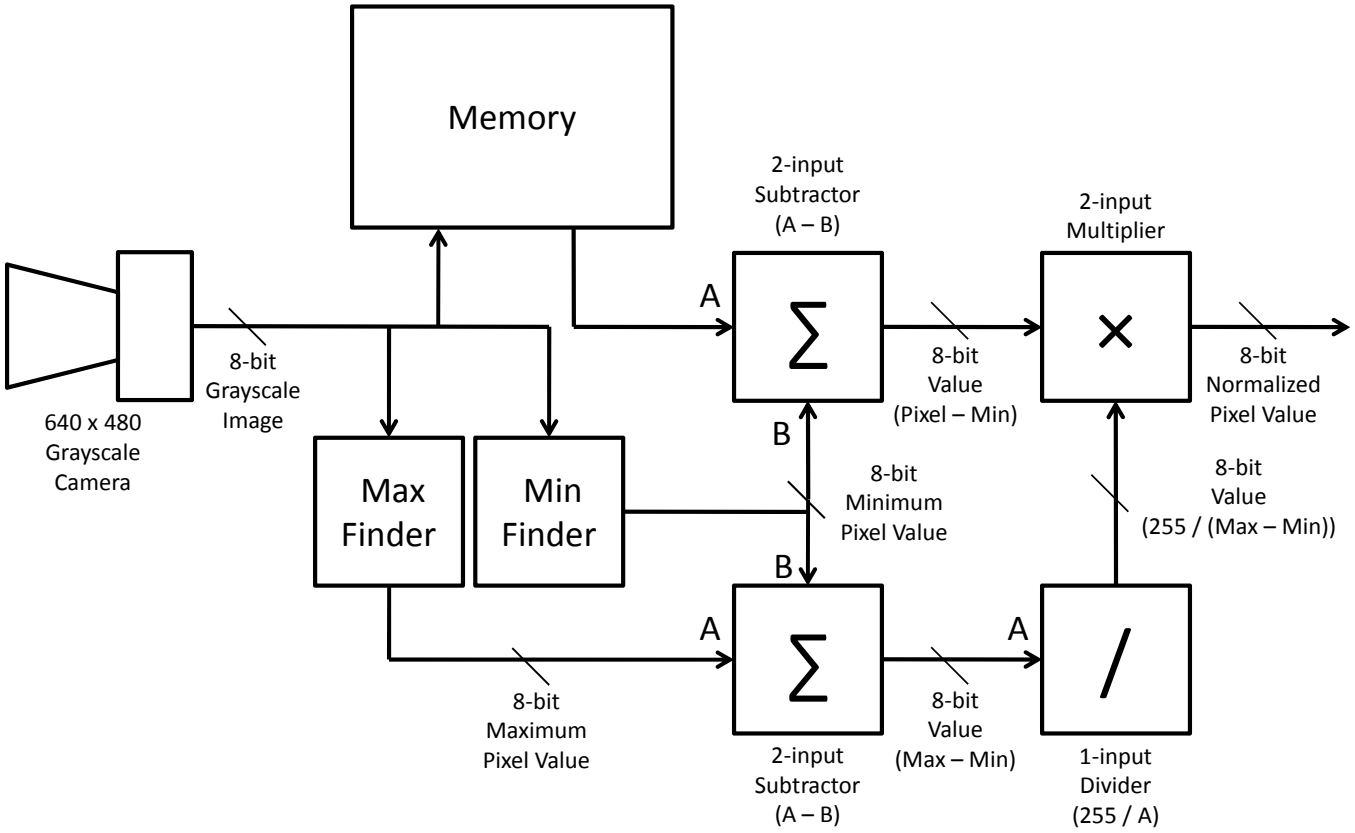

Figure 4.10: Normalization circuit

algorithms. This implies that the pipeline utilizes memory elements to store and retrieve data.

The offline technique also utilizes memory to store the incoming image data, but what separates the cached inline approach from the offline approach is the timing of when the actual computations begin. In offline processing, the entire image is slowly acquired from the camera and computations begin *after* it has been completely stored in memory. However, in cached inline processing the computations begin *prior* to the entire image being read out of the camera. What this timing implies is that there is an additional delay or time shift in the output of the system beyond the latency required to simply clock the data straight through the pipeline.

The size of the memory is not defined and can be as small as a single pixel or as large as the entire image. Situations most often encountered by the author involve storing a few rows of the image. Algorithms that require the buffering of a few rows of the image often involve kernel operations. These are 3×3 or 5×5 blocks of the original image that are centered around a pixel of interest. When buffering multiple rows of the image to perform kernel operations, latency is added in chunks. Each row buffered of the image will add the latency to clock out each pixel of that row from the camera. The effect of this latency simply delays the time between the capture of the image and the output of the image from the system.

However much data is buffered, this approach still computes the algorithms prior to the resulting image being completely stored in main memory, and is still a pre-processing technique. The main drawback of this approach is memory utilization. FPGA systems have limited block RAM available for internal memory storage systems, which implies that an external memory may be required to satisfy the design goals of the intended application. To demonstrate the use of the cached inline technique, an example will be explored.

4.6.1 Example: Bayer Pattern Demosaicing

Converting a Bayer pattern image into an RGB format is a common task when dealing with a raw image sensor. There are many algorithms to choose from that have varying degrees of arithmetic complexity and visual accuracy. For this example, the simplest algorithm will

be demonstrated which is Bilinear Interpolation. The general procedure for the bilinear method is that for each missing channel at a given pixel, the average of that channel in the neighboring pixels is used to fill in the missing values. This is demonstrated mathematically in

$$G\,(i,j) = \frac{1}{4}\sum_{\text{(m,n)= \{(-1,-1), (-1,1), (1,-1), (1,1)\}}} G(i+m, j+n). \tag{4.4}$$

Equation 4.4 describes how to compute the green channel value at a red or blue pixel in the Bayer image. The value is simply the average of all the neighboring green pixels. As shown in Figure 4.11, the green channel value at location 1 would be the average of the green pixels to the left, right, up and down from the red pixel at location 1. The blue value at location 1 would be the average of the blue pixels on the corners of location 1.

Computation of the red and green channels at a blue pixel such as at location 3 are similar to those for the red pixel. But the computation of the red and blue channels at a green pixel such as at location 2 are different based on what row of the image the location is in. At location 2, the red channel is the average of the two red pixels on the left and right side of location 2, and the blue channel is the average of the blue pixels above and below location 2. But at location 4, the red channel is the average of the two red pixels above and below location 4, and the blue channel is the average of the two blue pixels on the left and right of location 4. Therefore the implementation of the Bayer algorithm must distinguish between GR rows and BG rows.

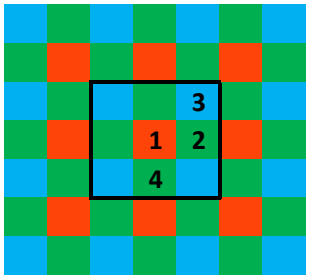

Figure 4.11: Bayer grid

An FPGA system implementation of the Bayer demosaicing algorithm is shown in Figure 4.12. In this implementation there are four main sections: the row buffers, the sliding pixel window registers, the control logic, and the arithmetic logic. The objective of this implementation is to create a 3×3 'sliding window' around each pixel point in the image. This is done because the computation of the missing channels at any given pixel location requires the neighboring pixels that touch the side or corners of the given pixel. Each of these individual units will be discussed.

The row buffers each store 1 full row of the source image, which, in this example, consists of 640 pixels. Ignoring the other logic elements in the system implementation, the row buffers as a whole are intended to store 3 complete rows of the image. The image data is cascaded one pixel at a time from one row buffer to the next as each new pixel is clocked out of the camera.

The sliding pixel window registers are what store the 3×3 pixel grid. As each pixel is shifted through the row buffers, the pixel data represented in the pixel registers slides one pixel from left to right in the image. The window begins in the upper left corner of the image and continues shifting right until it hits the right edge of the image and then it advances down one row in the image, starting again at the left edge. The limitation of this sliding window algorithm is that the results for the top and bottom row and the left and right column are undefined as a 3×3 window at those locations runs off the available image data. Therefore the resulting image is 2 pixels less in each dimension; a 640×480 image would become a 638×478 image. If a 640×480 result image size is required by the application, a workaround to this limitation can be to simply increase the capture window on the camera up by 2 pixels in each dimension or a 642×482 image in this case.

The purpose of the control logic in this circuit is to keep track of the row and column of the pixel of interest, which is the pixel located in the center $(2, 2)$ location of the sliding window registers in Figure 4.12. This is done so that the control logic can specify to the arithmetic logic what operations are to be performed and what registers to draw values from to compute the missing channel information. For example, if the center pixel is a red pixel such as that at location 1 in Figure 4.11, then the control logic would instruct the arithmetic

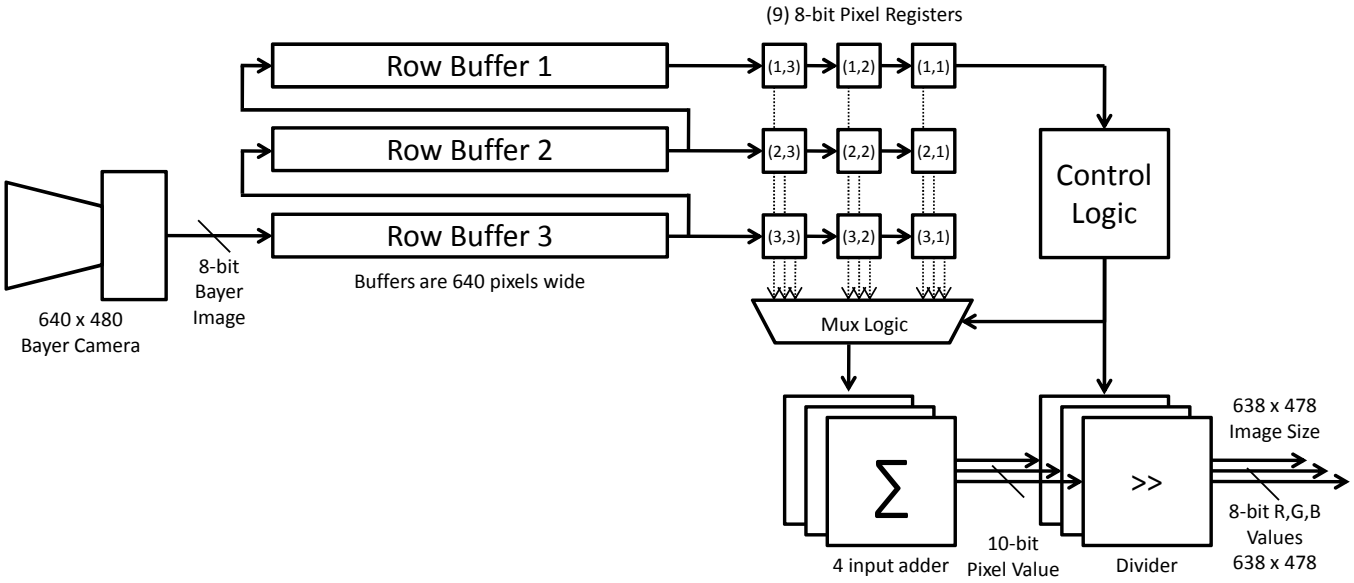

Figure 4.12: Bayer demosaicing logic circuit

logic to compute the green channel value from the average of the side pixels and the blue channel value from the average of the corner pixels.

The arithmetic logic consists of two 10-bit, 4-input adders, and a configurable divisor unit. The 4-input adders sum up either two or four values from the pixel registers depending on what type of pixel is located in the center $(2, 2)$ location of the sliding window registers in Figure 4.12. The configurable divisor unit is really just a simple multiplexor that can divide by 2 or divide by 4, which is all that is necessary for this example circuit. If the divisor is required to divide by 2, it simply 'shifts' the 10-bit result down one bit, and if it is required to divide by 4, it shifts the 10-bit result down by two bits.

In review, the purpose of this example was to demonstrate the cached inline image processing technique. In this technique, a portion of the image is buffered for use in inline processing. As shown in this example, three rows of the image were buffered internally, and the image processing arithmetic computations begin prior to the entire image being read out of the camera, which qualifies this example as a cached inline technique.

Chapter 5

Application: Vision-Guided Landing

5.1 Introduction

The problem addressed in this chapter is the autonomous landing of a fixed-wing MAV, such as the one shown in Figure 3.2, on a visually recognizable target at an approximately known GPS location. Using GPS coordinate-based navigation, the MAV may over- or undershoot the landing site due to inaccurate location measurement of the landing site or due to inaccurate position estimation of the MAV while in flight [1]. To compensate for this measurement error, an onboard vision system is added to the MAV, consisting of an image sensor and the computational hardware required to support real-time machine vision algorithms. Using an onboard vision system, the target landing site can be visually identified during approach, allowing the MAV's trajectory relative to the intended landing site to be determined. By including this information in the local control loop, an onboard autopilot can make the required trajectory corrections in real-time, enabling the MAV to land at the intended location.

A key part of this work has been the design and development of a custom computing solution specifically for small autonomous vehicles based on FPGAs. The FPGA employed in this design combines – on the same chip – one or more processor cores running conventional software with hardware resources that can be configured to implement application-specific circuitry. Much of the effort associated with the implementation of the vision system described in this chapter involved building, essentially from the ground up, an embedded software environment on an FPGA.

To demonstrate the capabilities of this vision guidance system in real-time adaptation within a dynamic environment, two vision-guided landing test scenarios were constructed. The first was to land the MAV at a static landing site location, visually distinguished by the

use of a red cloth target. This situation is a demonstration of the ability of the entire system, both vision and control, to guide the flight path of a MAV using visually acquired information. The second scenario was to land the MAV in the back of a moving vehicle. Success in this scenario is essentially impossible using only GPS information, but it is achievable using visually acquired information.

Sections 5.2 and 5.3 describe the computational platform and discuss how it is used to realize the algorithms employed in this system. Section 5.4 reports field test results, including outcomes from flight tests.

5.2 System Design

The autopilot board used in this vision-targeted landing application is the Kestrel autopilot from Procerus Technologies. The Kestrel autopilot and the Helios vision platform are the only two processing units onboard the MAV. The Helios board is responsible for all image processing tasks, and the autopilot handles the flight control of the MAV. In combination, these two boards constitute a two stage system for vision processing and vehicle control.

As stated in Section 5.1, the MAV vision system must perform the following actions. First, it must recognize the location of a red cloth target that identifies the landing site. Second, it must track and filter the location of the target. And finally, it must transmit the (x, y) image location of the target center to the autopilot via a serial link.

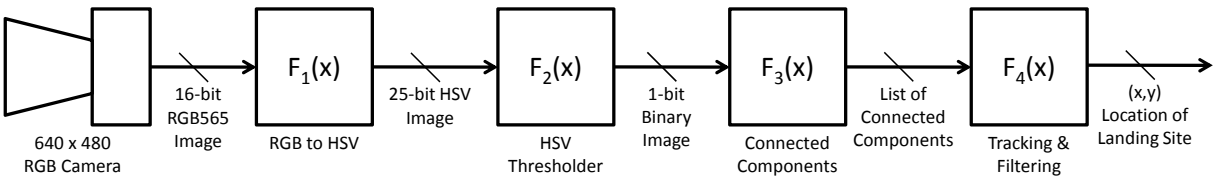

Figure 5.1: VGL 4 step process

The vision system processes each frame of video from the onboard camera to identify the target using color-based segmentation followed by the execution of a connected components algorithm. In this implementation, this process was broken down into four separate steps, which are shown in Figure 5.1. First, the 640×480 RGB 565 images captured by the digital camera were converted to the Hue-Saturation-Value (HSV) color space. Second, color segmentation was performed on each pixel using static HSV thresholds producing a binary image with a 1 representing each red pixel and a 0 representing each non-red pixel. Third, a connected components algorithm was run on the binary image, which also computed the center of mass of each red blob found. The resulting list of blobs were then tracked to ensure consistency between successive frames.

With an FPGA-centered design, a Configurable System on a Chip (CSoC) was available to implement these four image processing tasks. Before actual implementation on the Helios vision system was begun, the four tasks were analyzed to see which of them could take advantage of reconfigurable hardware on the FPGA. The guiding philosophy in this system design was to choose a software implementation of the task if it could run in real-time with available CPU processing resources, and otherwise to employ custom FPGA hardware to accelerate the computation so that it could run in real-time. Real-time operation is critical for this application, as is any application involving the control of an aircraft in flight. For the information about the location of the landing site to be useful in correcting the flight path of the MAV, it must be delayed as little as possible.

The following subsections analyze each of the processing steps in terms of whether they are best implemented in software or in reconfigurable hardware.

5.2.1 Color Space Conversion

Each pixel in the image was converted from the RGB color space to the HSV color space according to

$$H = \begin{cases} \text{undefined}, & \text{if } Max = Min \\ 60^\circ \times \frac{G-B}{Max-Min} + 0^\circ, & \text{if } Max = R \\ & \text{and } G \geq B \\ 60^\circ \times \frac{G-B}{Max-Min} + 360^\circ, & \text{if } Max = R \\ & \text{and } G < B \\ 60^\circ \times \frac{B-R}{Max-Min} + 120^\circ, & \text{if } Max = G \\ 60^\circ \times \frac{R-G}{Max-Min} + 240^\circ, & \text{if } Max = B \end{cases}, \tag{5.1}$$

$$S = \begin{cases} 0, & \text{if } Max = 0 \\ 1 - \frac{Min}{Max}, & \text{otherwise} \end{cases}, \tag{5.2}$$

and

$$V = Max. \tag{5.3}$$

This calculation requires a total of 12 operations per pixel: 5 compares, 1 add, 3 subtracts, 1 multiply, and 2 divides. If these calculations were to be performed in software on a video stream of 640×480 images at 30 Hz, then 110 million calculations would be required each second. Each of these calculations will at best compile into a minimum of three machine instructions per pixel on a conventional CPU.

This constitutes a significant volume of machine instructions for a single task, since the embedded PowerPC processor can execute at most 400 million instructions per second. Since a software implementation of color space conversion is infeasible, custom FPGA hardware must be used.

Figure 5.2 shows the pipelined implementation of the RGB to HSV color conversion core. In this implementation, the HSV color space conversion was divided into five steps in FPGA hardware. Each step is implemented in a single pipeline stage, so the processing

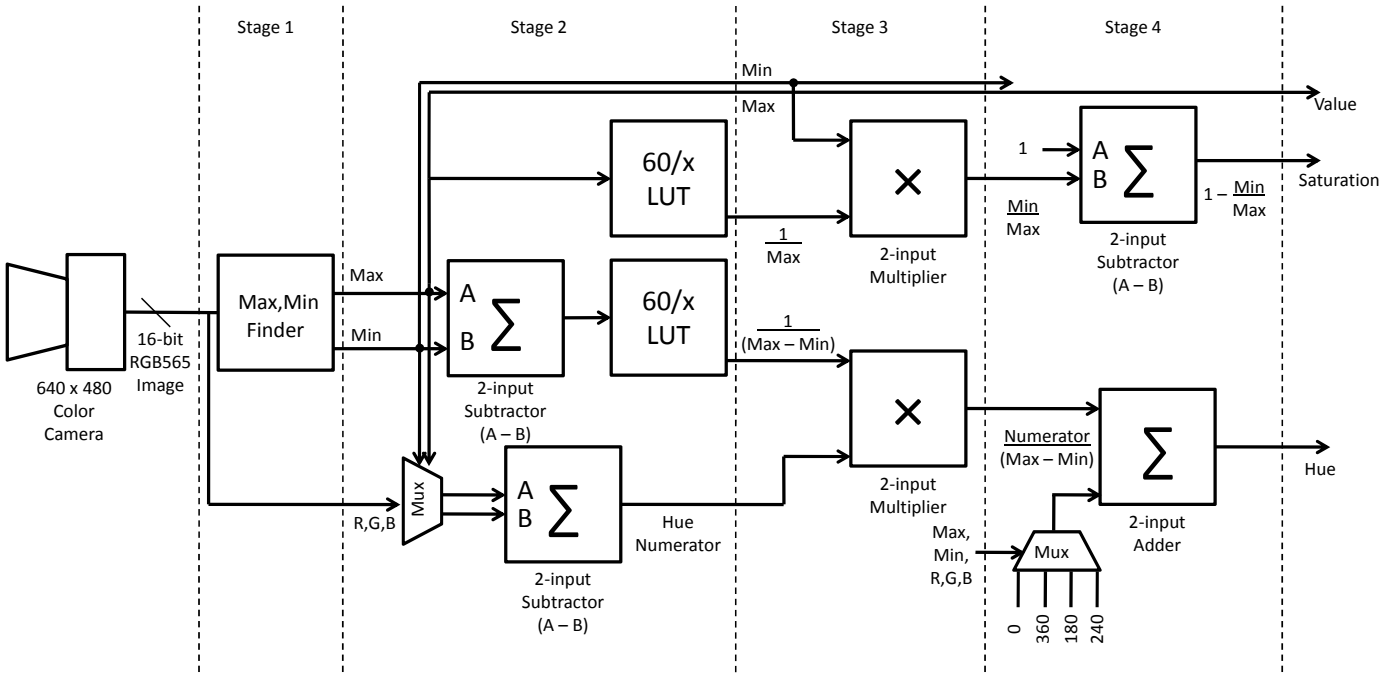

Figure 5.2: RGB to HSV color conversion core

adds just five clock cycles of latency after the pixel is read out of the camera. The first stage captures the incoming pixel data in a register. The second stage determines which of the three R, G, and B channels are the maximum and minimum values. The third stage uses two look-up-tables to perform division and determines the numerator of the Hue value. The fourth stage multiplies the $\frac{1}{x}$ values with their numerators using DSP slices on the FPGA. The fifth and final stage computes the Saturation value by subtracting the $\frac{Min}{Max}$ value from 1 as well as shifting the Hue value back up into the $[0-360]$ degree range.

This hardware implementation is substantially faster than any possible software version as processing a single image would take a minimum of 330 million sequential instructions in software. This implementation uses 103 slices of the FPGA hardware, or about 1.2% of the reconfigurable logic available in the FPGA.

5.2.2 Color Segmentation

Once each pixel has been converted to the HSV color space, it must be compared to HSV thresholds according to

$$image(y,x) = \begin{cases} 1, & \text{if } H_{Min} \leq H(x,y) \leq H_{Max} \\ & \text{and } S_{Min} \leq S(x,y) \leq S_{Max} \\ & \text{and } V_{Min} \leq V(x,y) \leq V_{Max} \\ 0, & \text{otherwise} \end{cases}. \tag{5.4}$$

H_{Min}, H_{Max}, etc. are the thresholds that define a region in the HSV color space that identifies the specific color of the landing site marker.

The HSV thresholds consist of max and min values for each of the hue, saturation, and intensity channels. Pixels must lie between these thresholds to be considered a possible match for the target. These threshold values were chosen to be Hue: $[330° - 10°]$, Saturation: $[40-100]$, and Value: $[100-255]$ after analyzing images of the target acquired from the MAV while in flight. If the incoming pixel is within the thresholds, then the color segmentation algorithm labels that pixel as a 1 implying that it is a 'red' pixel. If the pixel values are not within the thresholds, the pixel is labeled with a 0. Segmenting a single pixel in HSV color

space requires seven comparison operations. For a 640×480 video stream at 30 Hz, this would require about 65 million calculations per second, or a minimum of about 195 million CPU instructions.

Based on this analysis, color segmentation could potentially be performed in software, but it would require about half of the CPU resources available. In consideration of the processing demands of other required software tasks, the color segmentation algorithm was implemented in hardware.

The FPGA implementation of the color segmentation task is a relatively simple core that compares an incoming pixel to the HSV thresholds. The color segmentation hardware core consumes just 60 FPGA slices, or less than 1% of the FPGA. Due to the simplicity of the core, the comparison operations were implemented in only a single pipeline stage. In other words, an FPGA core that can threshold HSV pixel values was added at the cost of only a single cycle of delay to the incoming pixel values. In effect, this reduced the computation time from a minimum of 195 million CPU clock cycles (for a software implementation) to zero.

5.2.3 Connected Components

To identify the landing site marker, the 1's in the binary image produced in the color segmentation task must be grouped together as a connected component 'blob.' Each blob found in the image represented a patch of red pixels in the original image captured from the camera. The only information needed from each blob is the center of mass and the number of red pixels in the blob. To accomplish this, a connected components algorithm was modified to include the center of mass calculations as each red pixel of the blobs was found in the image.

The actual number of computations for a connected component algorithm varies widely depending on the number of red pixels in the image. Therefore, the decision to implement the algorithm in software or FPGA hardware was based on the length of time required to compute the components in a worst-case situation. If the software implementation of the connected components algorithm required longer than 33 ms to compute the components in a single 640×480 image, then it would be classified as unsuitable for real-time

operation (given a 30 Hz video rate). Although an absolute worst-case scenario was not constructed, hypothetical cases were able to be constructed in which the connected components algorithm was unable to complete in real-time.

To enable real-time computation of connected components, the algorithm was broken into two parts, one of which would be implemented in FPGA hardware and the other in software on the PowerPC processor. The first part of the algorithm, streak finding, requires the vast majority of the computations for the algorithm. This part of the algorithm scans each row of the binary input image one pixel at a time. If a red pixel is found (pixel is a binary 1), then the streak finding algorithm records the starting (x, y) pixel location and then counts the number of consecutive red pixels in the current row. When the end of the row is reached, or a non-red pixel is found (pixel is a binary 0), the streak finder stops counting and adds a streak to a list. Each streak in this list is represented by three values: (*y, xstart, xend*). This part of the connected components algorithm was well suited for implementation in FPGA hardware.

A streak finding core was added to the FPGA, requiring only 70 slices, less than 1% of the FPGA's reconfigurable resources. By implementing this in custom hardware, CPU computations that typically required around 18 ms (55% of the available 33 ms) per image in software now can be completely overlapped with other processing – effectively this part of the connected components algorithm requires zero time.

The second part of the connected components algorithm is streak grouping, which is computed in software. This part, mainly list processing, takes as input the list of streaks computed by the FPGA streak finding hardware core. It then groups the streaks together into blobs and computes the center of mass of each blob. This process usually runs in less than 1 ms and peaks at around 5 ms in near worst case situations, so it is suitable for real-time operation.

5.2.4 Tracking and Filtering

This portion of the vision system determined if any of the recognized red blobs could be classified as the intended landing site. The goal of this system is to filter out small or sporadic red particles in the image and provide a higher level of confidence that an identified

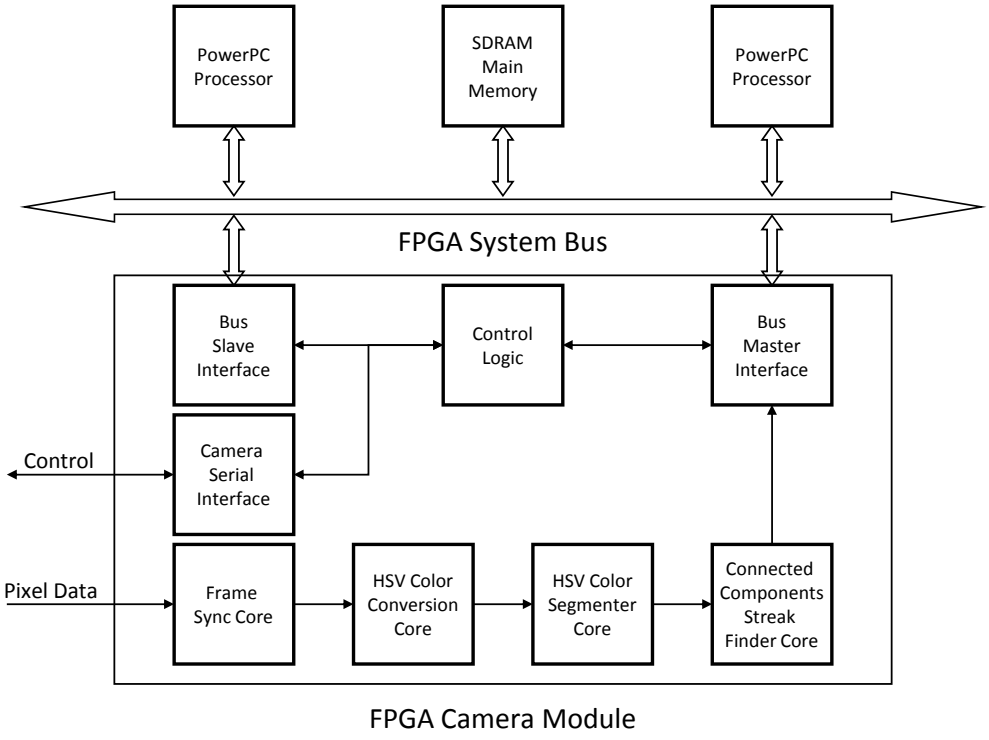

Figure 5.3: FPGA image processing core

red blob is indeed the target landing site. In this filter, the blobs are first sorted in order of size, and then the centers of mass of the largest eight blobs are tracked between frames. If the center of a blob moves a distance greater than a maximum threshold, then that blob is ruled out as a possible landing site. Once inconsistent blobs are filtered out, the largest blob is classified as the intended landing site if its size exceeds a given threshold. This processing step requires relatively few calculations and the software implementation easily runs in real-time. Therefore, no FPGA hardware was required for tracking or filtering.

5.3 FPGA System Architecture

The FPGA image processing core implemented for this vision system is depicted in Figure 5.3. In this architecture, the pixel data is received from the digital camera one pixel at a time. Each pixel is then directed into the pipelined RGB to HSV color conversion core. Five clock cycles later, the pixel is output from the HSV conversion core as a Hue-Saturation-Value pixel. That HSV pixel is then fed into the color segmentation core where

HSV thresholds are applied. If the pixel is within the range of colors being searched for, the color segmentation core outputs a binary 1, otherwise it outputs a 0. That result is then fed into the connected components streak finder core. The streak-finding core finds the continuous horizontal streaks of red pixels in the image and outputs three values for each streak found: the row of the streak in the image (y), the starting column of the streak ($xstart$), and the ending column of the streak ($xend$). Each of the streak values are then transferred to the main system memory over the system bus on the FPGA.

5.4 Results

An initial vision-assisted landing system used an off-board computation approach. An NTSC camera was mounted onboard a MAV which transmitted live video to a ground station where a laptop computer would perform any image processing tasks required. This configuration was simple to implement due to the use of off-the-shelf hardware. Using an off-board computation approach, on average, only three or four flight path adjustment commands could be sent from the ground station to the MAV during descent due to the transmission delay between the aircraft and the ground station. Experimentally, this limitation prevented achievement of a successful landing. The transmission delay between the MAV and the ground station is what dictated the need for an onboard vision system.

The test platform for onboard vision processing was a fixed-wing MAV (Figure 3.2) with dimensions 60" x 23" x 4.75" and a weight of 5 pounds. The MAV used a Kestrel autopilot (version 2.2) from Procerus Technologies to implement GPS guidance and vehicle control as well as the Helios vision system for image processing.

Test flights took the following form. First, the MAV was hand launched, after which it would climb to a specified altitude. The MAV would then fly to a specified GPS waypoint about 300 meters away from the landing site. Once this waypoint was reached the MAV would begin an autonomous GPS guidance controlled descent toward a roughly estimated GPS position of the landing site. When the visually identifiable marker for the landing site was in the field of view of the fixed, forward-looking camera, the Helios vision system would be enabled. The Helios would then begin transmitting the (x, y) image location of the target landing site to the autopilot. The autopilot then switched over from GPS guidance to

vision assisted control algorithms. The vision-based control algorithms on the autopilot then adjusted the pitch and heading of the MAV to correct the flight path toward the targeted landing site.

Flight tests were conducted in two scenarios. In the first, the MAV was directed to land at a static location specified by a red cloth target. The second scenario was to land the MAV in the back of a moving pickup truck. The results of each of these will now be presented.

5.4.1 Static Landing Site

This scenario was intended to demonstrate the capability of correcting the flight path of a MAV using visual information. To accomplish this, a red cloth landing site marker was placed at one location, and the MAV autopilot was then instructed to perform a GPS-based autonomous landing maneuver to a location 25 m away from the actual location of the target. By giving the autopilot a GPS location that was 25 m away from the actual target situations were produced where the MAV had to perform significant flight trajectory corrections to reach the target landing site.

Figures 5.4 through 5.9, show images captured from the Helios board during an automated landing approach. Both the RGB and color segmented images are shown. These images are a representative sample of what the Helios board 'sees' during a landing approach. In this landing trial, the target landing site was in an empty field, which was chosen for its level terrain and lack of obstacles. While the red cloth target is barely distinguishable in the RGB images, it is clearly identified using the HSV color segmentation process. While color segmentation may appear to be a simple process, the segmented images in Figures 5.4 through 5.9 clearly show that it is successful at locating the target landing site.

Figure 5.10 is a GPS plot of the flight path of the MAV while it is approaching the target landing site, where the two axis are N/S and E/W cardinal directions and are both delineated in meters. In the plot of Figure 5.10, it can be seen that the GPS landing target location was set at $(50, 0)$, while the static red cloth target was placed at $(25, 0)$. The MAV began its approach from about 350 m north and 100 m west of the intended landing site. Once the MAV reached about 50 m north of the landing site, the autopilot began using vision

Figure 5.4: Unprocessed RGB image captured during flight (≈100 m to target)

Figure 5.5: Color segmented image captured during flight (≈100 m to target)

Figure 5.6: Unprocessed RGB image captured during flight (≈50 m to target)

Figure 5.7: Color segmented image captured during flight (≈50 m to target)

Figure 5.8: Unprocessed RGB image captured during flight (≈15 m to target)

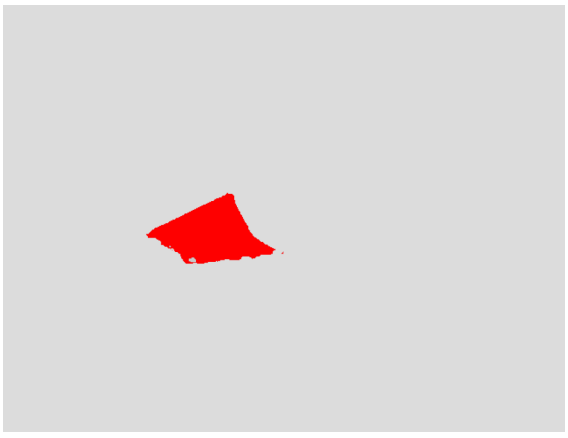

Figure 5.9: Color segmented image captured during flight (≈15 m to target)

assisted control. It is at this point that the MAV breaks from its original flight path and implements a new flight path that enabled the MAV to successfully land on the intended landing site. The purpose of Figure 5.10 is to show the trajectory change that occurred when the vision system was enabled, which provides evidence that the vision system was responsible for the successful landing and was not simply an idle passenger on a successful GPS-guided landing.

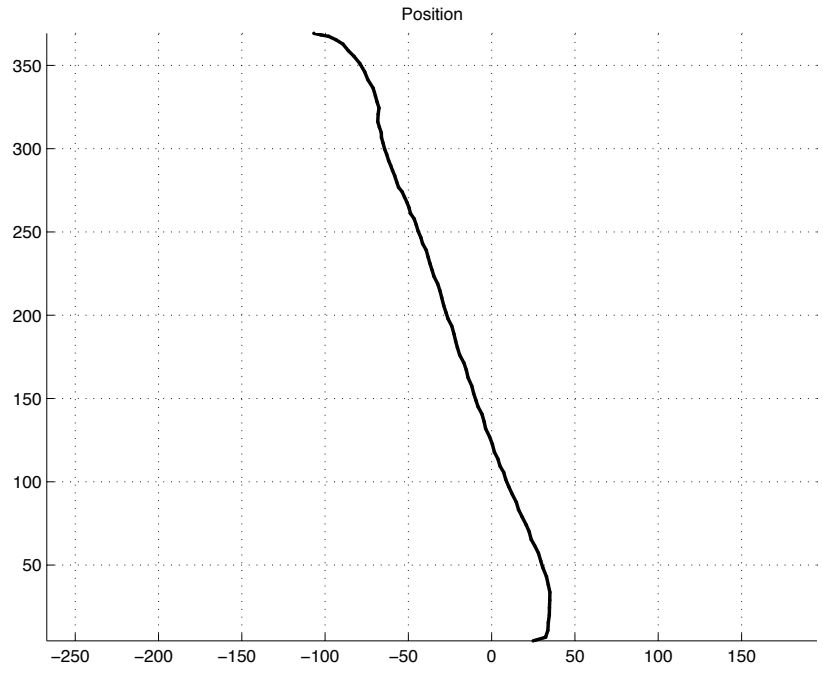

Figure 5.10: Flight data plot of MAV flight path GPS location for static landing site scenario. Target landing site is at location $(25, 0)$.

Figure 5.11 shows a plot of the image coordinates where the target landing site was identified in the incoming image stream. This figure identifies the initial location of the target landing site in the upper right corner of the image. The autopilot then adjusts to this information and steers the MAV to the right. After the course correction the target landing site bounces around the center and then slightly off to the right of center as the autopilot yaws into the wind to constantly adjust the desired flight path. This course correction is also noted in Figure 5.10.

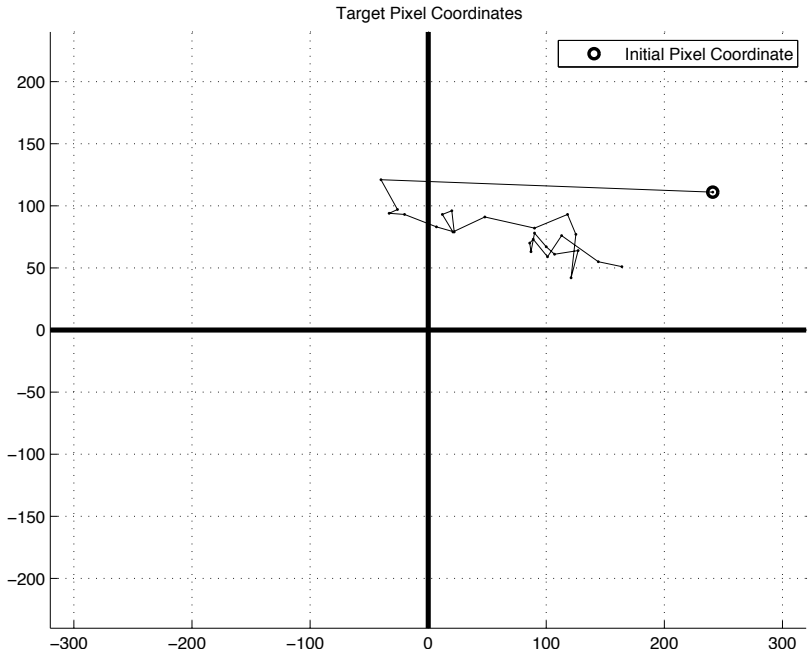

Figure 5.11: Flight data plot of identified target landing site in incoming video images

Images captured from the ground from this successful test flight can be found in Figures 5.12 and 5.13. As can be seen in Figure 5.12 the red cloth target was propped up using a short pole so that the red surface was more orthogonal to the approaching flight path of the UAV, which provided an easier target to visually locate. The MAV, which is highlighted in the yellow oval, can be seen in flight approaching the target and is traveling at about 30 meters per second. Figure 5.13 was captured immediately before the MAV struck the cloth target. After impact the cloth enveloped the plane as it decelerated while sustaining no damage. This result was consistently and successfully repeated through multiple trials. Therefore, the first flight test scenario, which was to land the MAV at a static landing location, can be stated as achieved.

5.4.2 Moving Landing Site

The second flight test scenario was to land the MAV in the back of a moving vehicle. In the first scenario, the main challenge being overcome by using vision in the control loop was inaccurate GPS position information. This situation introduces a moving target that

Figure 5.12: MAV approaching target landing site

Figure 5.13: MAV landing at target landing site

is not at any fixed GPS location. The challenge here is to locate a moving target at an approximate GPS location and then to dynamically adapt the MAV flight trajectory to the target's movement.

To land the MAV in the back of the vehicle, the bed of a pickup truck was lined with with the same red cloth target used in the static location scenario. The MAV autopilot was configured to approach the estimated location of the truck using GPS guidance. During the gradual descent of the MAV, the truck began driving forward at a speed between 10 and 15 mph, as shown in Figure 5.14. During the decent of the aircraft toward the moving truck, the operator enabled the vision system when it was about 100 meters behind the moving truck.

The consequence of this forward movement was that it slowed down the approach time and increased the amount of time available for trajectory adjustments. Consequently, it provided a different challenge for the vision and control system. In the static landing example, the challenge being overcome was the need for quick trajectory adjustments within the one or two seconds before impact. However, in the moving truck example, the landing process was slow and drawn out over about 10 seconds. In this situation, the challenge for the vision system was to consistently track the landing location over the extended time period and allow the autopilot to execute a slow and controlled decent.

In this flight test, the onboard vision system located the red target marker and was able to track it over 10 seconds until the MAV was able to successfully land in the back of the truck. Figure 5.15 shows the MAV approaching the truck approximately 1 second prior to successful landing. The goal of the second scenario was to land the MAV on a moving target. As can be seen in the figures, this goal was successfully achieved by landing the plane in the back of a moving pickup truck. With the success of both flight test scenarios, the landing of a MAV at a static and moving landing location, the original objectives for the vision guided landing application have been successfully completed.

Figure 5.14: MAV approaching moving vehicle landing site

Figure 5.15: MAV landing on moving vehicle

Chapter 6

Application: Feature Tracking

6.1 Introduction

The intent of this second application was to design a system that could produce meaningful information that could be used as the foundation for a variety of other algorithms. To determine what information to produce, a set of machine vision algorithms was examined that are potentially useful in the control of a small autonomous vehicle. In this examination, common image processing steps were searched for that require significant computation. Specifically looked for were computational components that typically prevent the algorithms from being implemented on small autonomous vehicles, where processing options are limited by the size, weight, and power constraints of the vehicle. Two related image processing steps were identified in many machine vision algorithms. They were feature detection and correlation.

Image feature identification or detection can mainly be described as the process of finding points in an image that can be located in the next image. This step is required due to the realistic limitation that not every pixel in an image can be correlated frame-to-frame. This is often caused by homogeneous regions in the image where a particular pixel is not easily distinguishable from neighboring pixels. The similarity of a pixel with its neighbors reduces the accuracy of correlation methods since the pixel may be easily confused with a neighboring pixel.

A feature correlation algorithm provides the ability to follow identified points through a sequence of images. This algorithm provides a rough estimation of the motion of objects in the image or the motion of the camera relative to the environment. However, the motion of the correlated points must be interpreted in some way to infer desired motion information.

It is important to note that the vision system is not a vehicle control system. The vision system produces information about the environment and does not determine how to control the vehicle to react to that environment. The implementation of control algorithms is left to a control system that uses the information produced by the vision system to make control decisions. Therefore, the design of a vision system must focus specifically on the implementation of vision algorithms and other image processing techniques. This effectively classifies a vision system as a sub-component of a larger control system. The application described in this chapter focuses only on an image feature tracking sub-system that can be utilized as part of a larger control system. The evaluation of this feature tracking vision system implementation will therefore not be based on successful vehicle control, but rather it will be judged based on its ability to autonomously identify and correlate image features frame-to-frame.

Section 6.2 describes the algorithms chosen to be implemented in this vision system application. The details of the vision system architecture are presented in Section 6.3. Section 6.4 reports on the results achieved by this implementation.

6.2 System Design

Once it had been decided to implement feature detection and correlation algorithms, a vision system then had to be selected that could perform these tasks. While the VGL application described in Chapter 5 used a Xilinx FX20 FPGA, the larger FX60 version was required for this application due to the increased need for FGPA configurable logic. The specific FPGA embedded system that was used for this system was the Helios robotic vision platform (Figure 3.1) developed at Brigham Young University [14]. The central component of the Helios platform is a Xilinx Virtex-4 FX series FPGA with dual PowerPC embedded processors that run at up to 450 MHz. This specific FPGA is ideal for embedded image processing as it provides both an embedded CPU for standard C code execution and reconfigurable FPGA fabric for hardware accelerated algorithm implementation. As a computing platform, it serves as a rich development environment with a small physical footprint that is well suited for mobile image processing applications.

To detect feature points in the image suitable for correlation, a Harris feature detector [17] is used, which is defined by

$$C(G) = det(G) + k \times trace^2(G) \tag{6.1}$$

and

$$G = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}. \tag{6.2}$$

The I_x and I_y values in Equation 6.2 are the gradients in the x and y directions in the image. This method was chosen for its reasonable computational requirements and ease of FPGA implementation. Figure 6.1 provides an example of x and y gradients in an image. The x gradient is the first derivative of the image taken in the horizontal direction, and the y gradient is the first derivative taken in the vertical direction. For example the x gradient at the image location $I(320, 240)$ would be $I(321, 240) - I(319, 240)$.

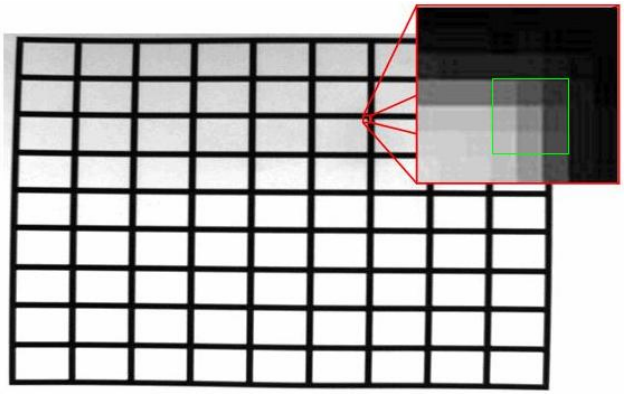

Figure 6.1: Example of x and y image gradients

Figure 6.1 is a picture of a white piece of paper with crisscrossing black lines on it. The highlighted section shows the inner bottom left corner of an intersection of a vertical and horizontal line. As can be seen in the figure, the transition from black to white is gradual and not immediate, which implies that there is a non-trivial horizontal and vertical derivative. The green window in Figure 6.1 represents a sample 3×3 window that would be passed into

the Harris feature equations. The 3×3 window fades from black to white in both the right to left and top to bottom directions, which will produce a strong Harris feature value.

Once the image feature points have been identified, they need to be located in other frames. To accomplish this, a template matching correlation algorithm was implemented, which is the foundation of the most basic tracking methods available. In this work, a template window (8×8 pixels in this implementation) around each feature is compared to each possible location within a specified search window. The location in the searched image that best matches the original template is selected as the correlated location.

To provide increased tolerance to image noise, template matching was performed on a rank transformed image rather than on the original grayscale image. To compute the rank transform of an image, each pixel is ranked, according to brightness, against neighboring pixels within a given radius. For example, if the center pixel in a 3×3 window was the brightest out of all the pixels in the window, then that center pixel would be given a rank of 8 indicating that it was brighter than the other eight neighboring pixels. The rank transform produces a non-parametric and somewhat rotation invariant image format that increases the accuracy of template matching comparisons. The benefits of template matching using a rank transformed image can be found in [18]. An FPGA core to perform the rank transform was added to the camera core as shown in Figure 6.2.

6.3 System Architecture

With an FPGA platform that contained both reconfigurable FPGA logic and dual embedded PowerPC processors, various options were available to implement these algorithms. The guiding intent was to utilize the system in the best way possible to increase the performance of the required algorithms. Therefore, it was decided to utilize the reconfigurable FPGA fabric to perform as much of the computation as possible. The system was designed with two main components. They were: an FPGA feature detection core and an FPGA template matching correlation core. The architecture of each of these components will now be described.

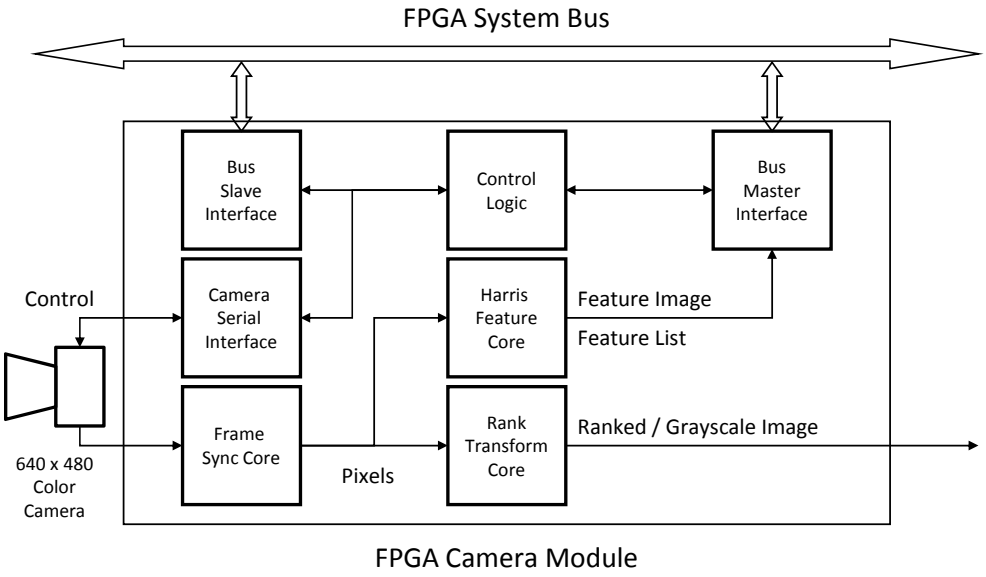

Figure 6.2: FPGA camera core

6.3.1 Harris Feature Detector Core

The Harris feature detector core was designed as a single component of a larger camera core as depicted in Figure 6.2. The camera core serves as the FPGA interface for an external digital camera. This core, shown in Figure 6.3, implements all the required functionality to perform the Harris feature detector algorithm on a grayscale image. The camera core consists of the logic required to communicate with both the camera and the FPGA system bus and other various image processing cores, among which the Harris feature core is one component.

The camera used in this implementation was a VGA image sensor, which was configured to output 8-bit grayscale pixel values. These pixel data values are then fed through a series of image processing cores. The Harris feature core calculates a feature value, as described in [17], for each pixel in an image. These feature values are then stored in main memory using two different methods. Each feature image can be stored in main memory as an 'image' (640×480 at 2 bytes per pixel), or they can be formed into a 'list' that stores the (x, y) location of the strong feature points in the image. An example of a Harris feature image can be found in Figure 6.5 along with the source image in Figure 6.4. The 'image' of

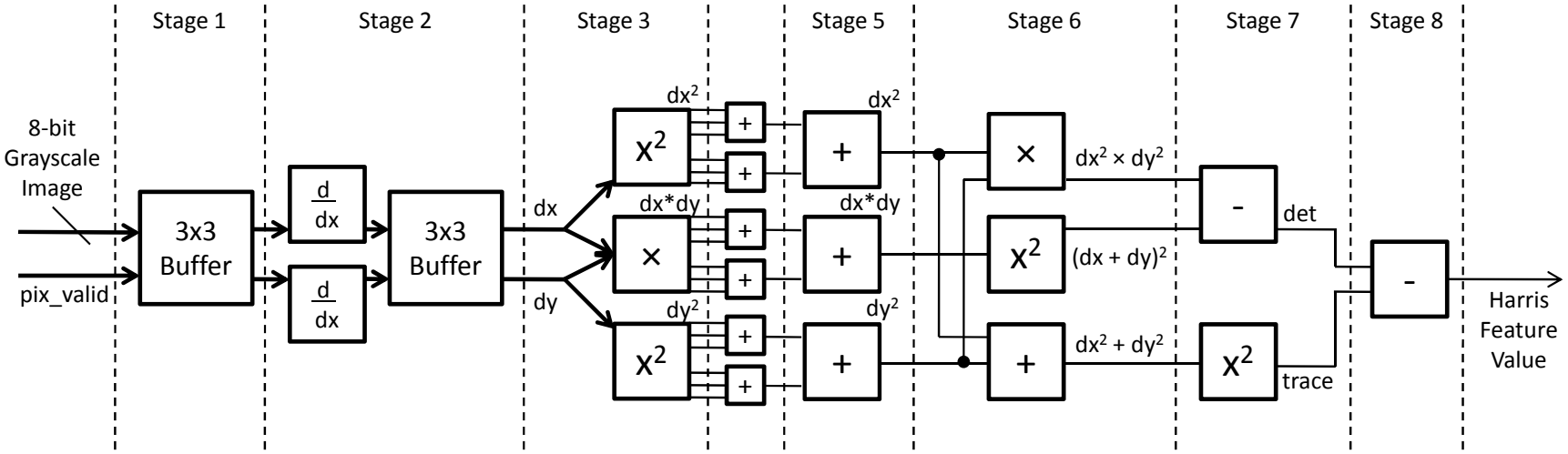

Figure 6.3: Harris feature detector core

the features is useful for user visualization, while the 'list' is more useful in feature dependent software algorithms.

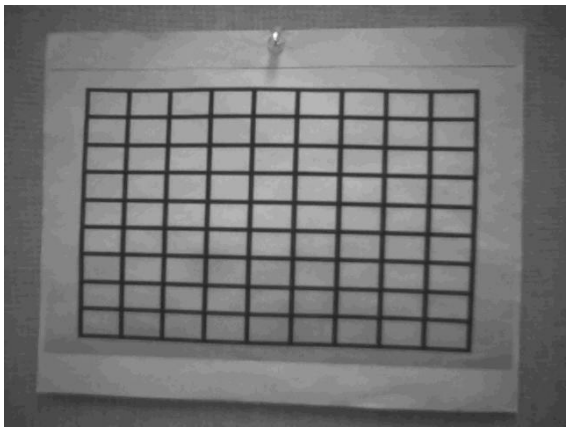

Figure 6.4: Harris source image

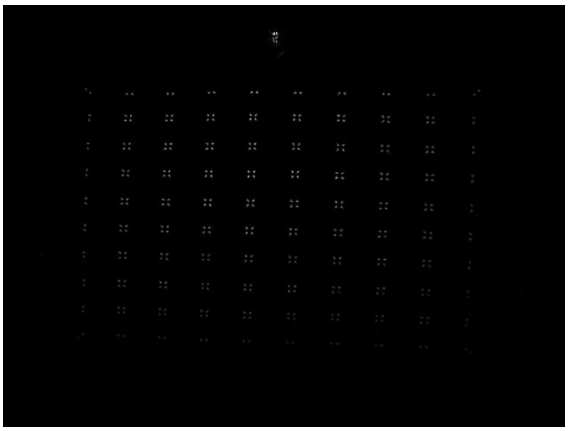

Figure 6.5: Harris feature image

An important characteristic of the data flow in this application is that the image is transferred from the camera one pixel at a time, which allows for easy implementation of pipelined image processing algorithms on the FPGA. This was utilized by implementing the Harris feature equations (6.1) and (6.2) with a 9-stage pipeline architecture using a 3×3 kernel window for both the gradient and matrix calculations. This inline implementation allows the Harris feature values to be computed as the image data is being streamed out of the camera. The image properties that were used were a 640×480 image size, a frame rate of 30 Hz, and a camera data bus speed of 27 MHz. At these rates, a single image is transferred out of the camera in 11 ms. The pipelined implementation adds only 9 clock cycles of latency to this time, which effectively allows the Harris feature values for an entire image to be instantly available after an image has finished being read out of the camera. For comparison, an OpenCV software implementation of the Harris feature detection algorithm requires 14 ms, which does not include image capture time, for a similarly sized image on a 2.1 GHz Core Duo desktop computer.

There are some significant differences between this FPGA hardware implementation and an OpenCV software implementation. Due to the resources required to implement

floating point calculations in FPGA hardware, this implementation uses only fixed point arithmetic in contrast to OpenCV's double precision software implementation. The resulting feature values of this implementation are also constrained to 16-bit integer quantities for each pixel, which lacks the sub-pixel accuracy of OpenCV. The limited output range of 16-bits is due to the use of bit-width minimization techniques where an FPGA bit-vector, representing a numeric quantity, is reduced in size and precision in an effort to conserve FPGA hardware resources and allow a faster processing time. Consequently, this implementation cannot be considered to be perfectly comparable to an OpenCV implementation, but it does run in real-time.

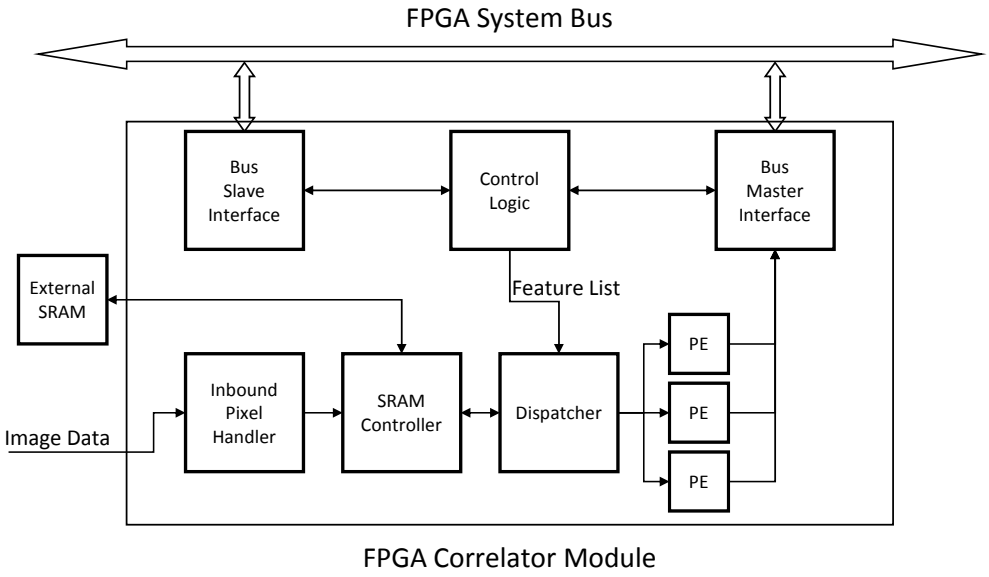

Figure 6.6: FPGA correlation core

6.3.2 Feature Correlation Core

The feature correlation core, as shown in Figure 6.6, was designed as a standalone core that would implement the template correlation computations of the vision system. This core does not automatically correlate the features that are detected in the Harris detector core. While this is physically possible, allowing the software routines running on the embed-

ded PowerPC to filter and select the features for correlation provides greater flexibility for different applications. The features chosen for correlation are passed to the correlation core as a list of points, which are then correlated, and returned to the software as a list. This list contains the original (x, y) feature location, the correlated location (x, y), and an error value that signifies the difference between the correlated location to the original location.

The correlation core seeks to correlate image features through the use of template matching. The pixel data used in template matching correlation is received directly from the camera core as the pixels are received from the camera. The pixels are received by the inbound pixel handler, which coordinates the writing of images through the SRAM controller to the external SRAM on the Helios platform, which serves as a high-speed image cache. Once an image has been completely stored in the SRAM, features can then be correlated in that image.

The dispatcher is responsible for transferring the template area and then the search area out of the SRAM and into a processing element, which actually performs the template comparisons. When a processing element has finished the correlation comparisons, it outputs the (x, y) correlated location with the lowest Sum of Squared Differences (SSD) error. That (x, y) location along with the SSD template error and the original pixel location are then transferred back over the system bus to main memory.

Each of the major components in the correlation core will now be described.

Inbound Pixel Handler

As the pixels are received from the camera, the inbound pixel handler, shown in Figure 6.8, writes them into an SRAM, which serves as a local image cache. The SRAM has a 20-bit address space and a word size of 36-bits, which implies that the SRAM is not byte addressable. The upper 4 bits of the 36-bit word are not used in this implementation giving the SRAM a usable capacity of 4 MB. In this application that 4 MB space is divided into eight $\backsim$500 KB segments, each of which is large enough to contain one 640×480 image at one byte per pixel. The inbound pixel handler provides the functionality to keep track of the memory address of the current image that is being written, along with the rotated addresses of the previously acquired images.

To simplify the access of image data in the SRAM, the 20-bit address space was partitioned into three sections, as shown in Figure 6.7. The lower 8 bits are used to specify the x coordinate, while the middle 9 bits are the y coordinate. The upper 3 bits are used as a frame select among the eight possible images that can be stored in the SRAM. Through the use of this addressing scheme, individual pixels can be read out of the SRAM by simply concatenating the y and x coordinates into an address.

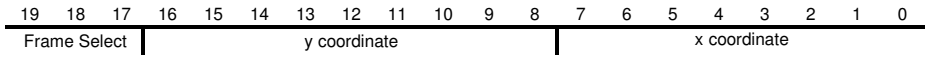

Figure 6.7: Partitioned SRAM address space

SRAM Controller

The SRAM controller or arbiter core, shown in Figure 6.8, serves as the interface to the external, single-ported SRAM on the Helios platform. In this application, a single read port and a single write port were required. To arbitrate between simultaneous requests on the two internal ports, a simple protocol was implemented to select the least recently serviced port. The SRAM has a 3-cycle operational latency with zero required wait states between requests. The 3-cycle request period is matched with a 3-stage pipeline in the SRAM controller. The pipeline operation allows a single read or write request to be issued on each cycle, which allows the SRAM to be utilized at its maximum available bandwidth. This is a significant achievement since the SRAM memory bandwidth is the bottleneck in this correlation core.

It is important to note that the SRAM is a 36-bit word-sized memory device. This implies that each read and write operation is limited to four bytes; the upper four data bits are simply ignored.

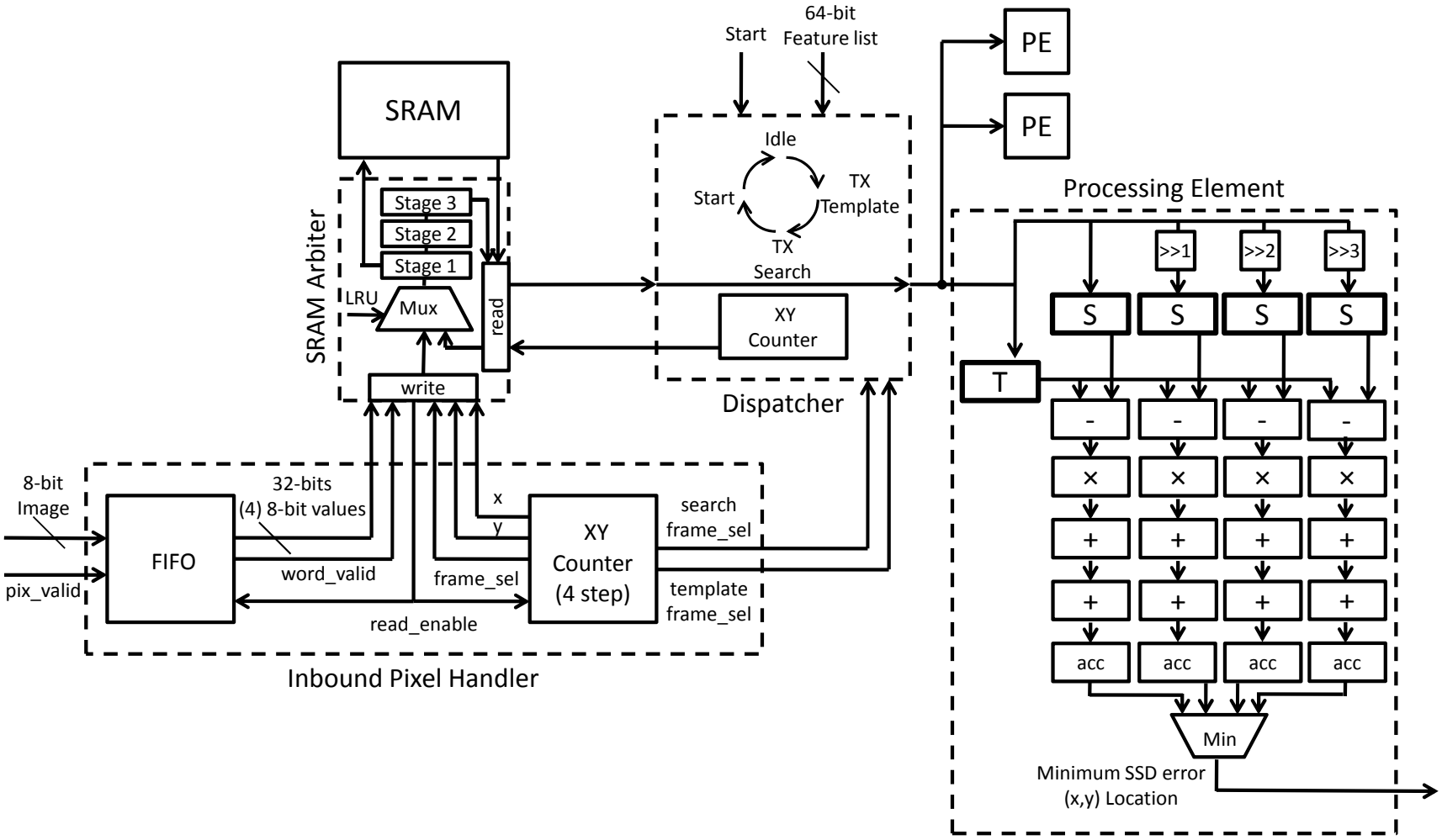

Figure 6.8: Correlator core detail

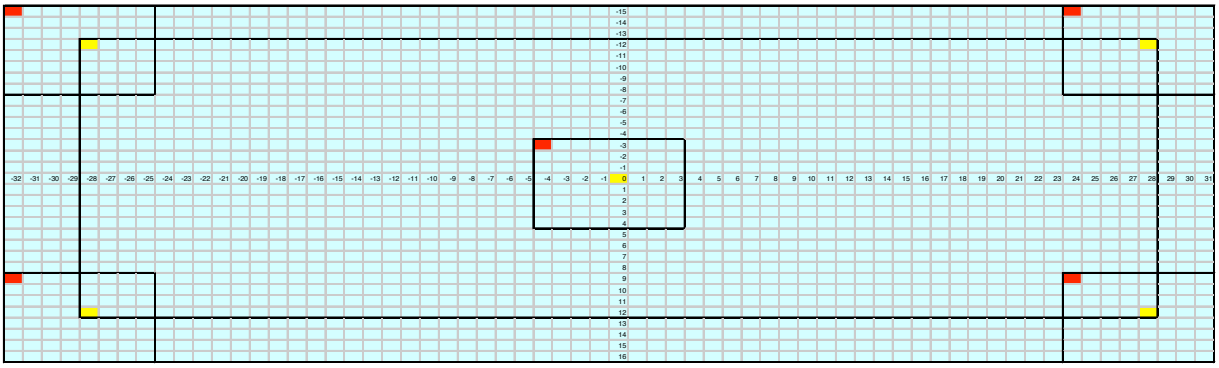

Figure 6.9: Template matching search area

Dispatcher

The main responsibility of the dispatcher is to read out the template window area from one image in the SRAM and the search window area from another image in the SRAM and transfer them to a processing element as shown in Figure 6.8. The features that have been selected to be correlated are stored in the dispatcher in an internal list. Each feature in the list consists of an (x, y) feature location, and an (x, y) search location, using 16-bit unsigned numbers. Once the dispatcher has received a list of features to correlate from the software algorithms running on the PowerPC, it is given a command to start processing those feature points. The processing elements are iterative computation engines and require a set amount of time to complete their computations. Consequently, the dispatcher cannot continuously dispatch data to one of them as it has to wait for one of the elements to complete and become idle. Once, the dispatcher has started execution, it will continue to dispatch features to the processing elements until all features have been correlated.

The (x, y) feature location is used to read out an 8×8 template window from the SRAM into an idle processing element. An example of the template window is depicted as the center box in Figure 6.9. As can be seen from the figure, the template window has a non-symmetric radius which extends 4 pixels to the left and down from the center feature location (yellow square) and 3 pixels up and to the right. A symmetric window radius such as 7×7 or 9×9 is preferable for template matching algorithms. But, as will be shown

later, an 8×8 window aligns better with the maximum read port widths of FPGA memory elements.

The (x, y) feature location defines the center of the template window that is stored in the SRAM cache. The window is read out of the SRAM row-by-row, starting with the top row. As the window width is larger than the SRAM read width, multiple memory reads are required for each row. If the x coordinate of the feature lies on a 4-byte memory boundary (meaning the last 2 bits of x are zeros), then the fastest way to read out the template area from the SRAM would be to read out the 4-byte word that is one address before the feature location and then read out the 4-byte word starting at the feature location. In this situation, a template area could be read out of the SRAM in 16 memory read operations. But if the x location is not on a 4-byte aligned boundary, then the template window area will spill over 3 sequential 4-byte words in the SRAM requiring at a minimum 24 memory reads from the SRAM to read out the template window. This is the first memory transfer challenge in the system.

There are two possible solutions to overcome this. The first is to use byte re-alignment engines that drop leading and trailing bytes from the reads as necessary to generate an 8-byte wide row from the 12-byte row read out of the SRAM. This is the optimal solution as it keeps the number of SRAM memory reads to the minimum number of 24 but requires complicated control logic that was deemed too expensive in FPGA reconfigurable logic. Therefore, a second solution was implemented, which was to read out the template window one byte at a time. This requires 64 memory read operations, which is more than double the minimum number of 24, but requires less complicated control logic. To put the template area choices in perspective, the additional 40 memory reads were considered insignificant when compared to the 512 memory reads that are required to read out the search area.

A search area that was as large as possible was desired to provide the ability to track a moving target over a larger area. The size of the search area was chosen to be the maximum area that could fit inside a single FPGA BRAM, which has a size of 2,048 bytes. These BRAMs serve as reconfigurable dual port memory elements inside the FPGA that can be used for any application. The BRAM was configured to have dual, 4-byte wide read / write ports, which is the realistic maximum width for these ports. As a 4-byte word memory, the

BRAM had an address space of 9 bits. The BRAM address space was broken down in a similar manner to the SRAM address space. The upper 5 address bits were used to indicate a y coordinate in the BRAM and the lower 4 bits were used to indicate the 4-byte aligned x coordinate. Partitioning the BRAM in this way allows for a 64×32 block of the search image to be stored in a single BRAM. It would be preferable if this search area were a square and not a rectangle, but due to the size limits of the FPGA BRAM, this is the largest area that can be stored in a single BRAM.

The search area (x, y) location in the feature list defines the center of the 64×32 search window. The search window can be at any location in the second image and is not required to be centered at the feature location as shown in Figure 6.9. This allows the search window to be biased according to an estimated location computed in software according to some prediction algorithm. By implementing a biased search window, a feature can be correlated to any location in the image regardless of the distance from the original location.

With a search area size of 2,048 bytes partitioned into 512 words of 4 bytes, the transfer from the SRAM would require a minimum of 512 read operations if the x coordinate were on a 4-byte address boundary. The same byte alignment issues exist for the search area as for the template area. To overcome these issues, a varying, non-symmetric search area radius was implemented so that the start of the search area always begins on a 4-byte address boundary. This was accomplished by first computing the starting 1-byte aligned memory address of the search area (the red block in the upper left corner in Figure 6.9) using the search area (x, y) location from the feature list and then truncating the last two bits off the address to generate a 4-byte aligned memory address. Using this method of non-symmetrically varying the search area radius results in four possible search area configurations as described in Table 6.1.

Table 6.1: Four possible search area configurations

Search area x address	Left x radius	Right x radius
4 byte aligned + 0	-32	+31
4 byte aligned + 1	-33	+30
4 byte aligned + 2	-34	+29
4 byte aligned + 3	-35	+28

The varying search radius only applies to the left and right boundaries of the search area, as the upper and lower boundaries are controlled by the y coordinate of the search area location, which is not affected by the x coordinate 4-byte memory address alignment. As shown in Table 6.1, the x radius of the search window varies from $(-32 \,..\, +31)$ to $(-35 \,..\, +28)$.

Processing Elements

The processing elements (PE) serve as the main computation engines for all template matching comparisons. The template and search windows of the images are transferred into the PE's as previously described and are stored internally in FPGA BRAMs. The main task of the PE is to compare the 8×8 template window to each possible 8×8 window location in the search area. An SSD comparison between the template and the search area windows was used to produce an error quantity representing the similarity between the two windows. The sum of squared differences was computed by first subtracting each pixel in the 8×8 template from the same corresponding location in the 8×8 search window resulting in a 8×8 difference window. Each of the 64 values in the 8×8 difference window was then squared so the difference value between each pixel location is positive. The 64 values in the squared difference window were then summed together to produce a sum of squared differences. The 8×8 window in the search area that had the lowest SSD error was selected as the match for the correlated feature location.

The number of possible window locations is less than the actual size of the search area due to template window overlap. The number of possible search locations can be seen in Figure 6.9 as the interior black rectangle that ranges $(-12 \,..\, +12)$ in y and from $(-28 \,..\, +28)$ to $(-31 \,..\, +26)$ in x depending on the search area (x, y) location byte alignment. This results in 1425 possible search locations.

To meet the design objective of processing speed, the template comparisons need to be computed quickly. The main bottleneck of this process is the memory access bandwidth of the FPGA BRAMs. They support a maximum of 2 read or write ports at 4 bytes wide each, resulting in a maximum read width of 8 bytes per clock cycle. This maximum BRAM read width corresponds to a single row of an 8×8 template window, which is the reason

why an 8×8 template window was chosen rather than a 7×7 or a 9×9 window size which both have a symmetric radius width of 3 or 4 respectively. With a read width of 8 bytes, a template window can be compared to a possible search location in 8 clock cycles, one cycle for each row in the template window.

With memory read ports configured to 4 bytes, this configuration will only work well on template comparisons that operate on 4-byte aligned memory reads. This is the same difficulty experienced when reading the template window out of the SRAM. In that situation, the template area was able to be read out of the SRAM one byte at a time due to only requiring 64 memory reads for the entire operation. However, in this situation comparing the template window to the search area sub-window one byte at a time would drastically reduce performance (requiring 64 cycles to compare one window instead of 8) due to the large number of comparisons that need to be computed.

Therefore, to overcome this challenge, 4 copies of the entire search area were created using 4 BRAMs and three byte re-alignment engines. These re-alignment engines were implemented to drop the first one, two, or three bytes of the search area data as it is transferred from the SRAM to the PE. By dropping zero, one, two, or three leading bytes of the search area, each column of the original search area now lies on a 4-byte boundary in one of the four BRAMs. By creating multiple copies of the search area, four template comparisons can be performed in parallel. With this implementation, four template comparisons can be computed in 8 clock cycles resulting in an average number of 2 clock cycles per template comparison. With 1425 locations to search, a PE can correlate a single feature in about 3000 clock cycles, which on an FPGA with a 100 MHz clock requires 30 µs.

6.4 Results

The feature detection and correlation cores were implemented as described on the Helios robotic vision platform. In this implementation three PE's were used, as shown in Figure 6.8, which provided concurrent processing. More PE's would have been implemented, but the FX60 FPGA lacked enough DSP slices, which contain a hard-core, high speed multiplier for the SSD comparisons. The camera used was a Micron MT9V111 VGA CMOS image sensor that runs at a maximum of 30 Hz, even though this implementation is capable of

higher frame rates. With a system frequency of 100 MHz, the entire vision system consumed 3.4 watts measured at the input terminals to the Helios platform resulting in a system that meets the low-power needs of a variety of applications. The resources consumed by the major components of this system are listed in Table 6.2.

Table 6.2: FPGA resource consumption of major system components

Component	FPGA Slices		BRAMs		DSP Slices	
Harris Feature Detector Core	605	2%	4	1%	30	12%
Camera Core	2736	10%	18	7%	30	12%
SRAM Controller	99	1%	0	0%	0	0%
Dispatcher	151	1%	0	0%	0	0%
Single Processing Element	932	3%	5	2%	32	25%
Correlation Core	3877	15%	20	8%	96	75%
Total System Utilization	10171	42%	168	72%	126	98%
Total System Capacity	25280	100%	232	100%	128	100%

Once the design had been implemented in FPGA hardware, exact execution times of the individual components were measured. The measurements can be found in Tables 6.3 and 6.4. These measurements did not show much variation and can be considered stable execution times. While Table 6.4 lists the maximum number of correlations that could be computed between frames captured from the camera, it would be impractical to attempt to continuously correlate image locations between frames. The reasoning is that the goal of the vision system is not to just correlate image feature locations but rather to use the correlated feature points in some other algorithm such as video stabilization or an obstacle avoidance algorithm, which implies there needs to be some time left over after correlation to perform other useful computations. Therefore, the number of correlations that would be attempted would be dependent upon the amount of time needed by other processes that use those correlations. If half of the available time was utilized for feature correlation, which was a rule-of-thumb the author used as a starting point, the system would be capable of computing over $1,300$ image feature correlations per image frame given a 30 Hz video rate.

Table 6.3: Performance of implemented components

Component	Time (ns)	Clock Cycles
Transfer data out of SRAM into PE	6700	670
PE Correlation	30050	3005
Average time to correlate one feature	12493	1249

Table 6.4: Maximum number of correlated features per time period

Time Period	No. Correlated Features
1 ms	80
1 frame @ 30 fps	2636
1 frame @ 60 fps	1318

In review, the purpose of this second application was not to perform any flight tests or trials but rather to design an image processing system that could be used as a foundation for other algorithms. Image feature detection and correlation were chosen as the algorithms for this second application based on their general applicability to a variety of algorithms such as image stabilization, target tracking, obstacle avoidance, or motion estimation. Since there was no flight test or trial event to verify the success of this implementation, only empirical methods based on visual observation could be utilized. Visual confirmation of correct operation was achieved by placing various printed, grid-pattern templates in front of the camera and observing the results.

An image is included in Figure 6.10 that illustrates the feature detection and correlation algorithms as implemented in this vision system. The red crosses identify where strong Harris features were detected in the current frame, and the green crosses identify where the features were correlated to in the previous frame. As can be seen in the image, the points that were tracked are near the corner edges of each of the checkerboard blocks. This demonstrates that the Harris feature detector correctly identified strong feature points to track and did not pick random pixel locations throughout the image. In this image, the camera was panning down to illustrate that the correlated points (green) in the previous image follow the location of the feature in the current image (red). As can be seen, the green crosses are

generally correlated directly below the red crosses, which is expected given that the camera was panning down.

After reviewing the results found in Figure 6.10, it can be concluded that this implementation fulfills the objectives of this application by implementing a vision system that finds and correlates a significant number of image features at camera frame rates while consuming just 3.4 watts.

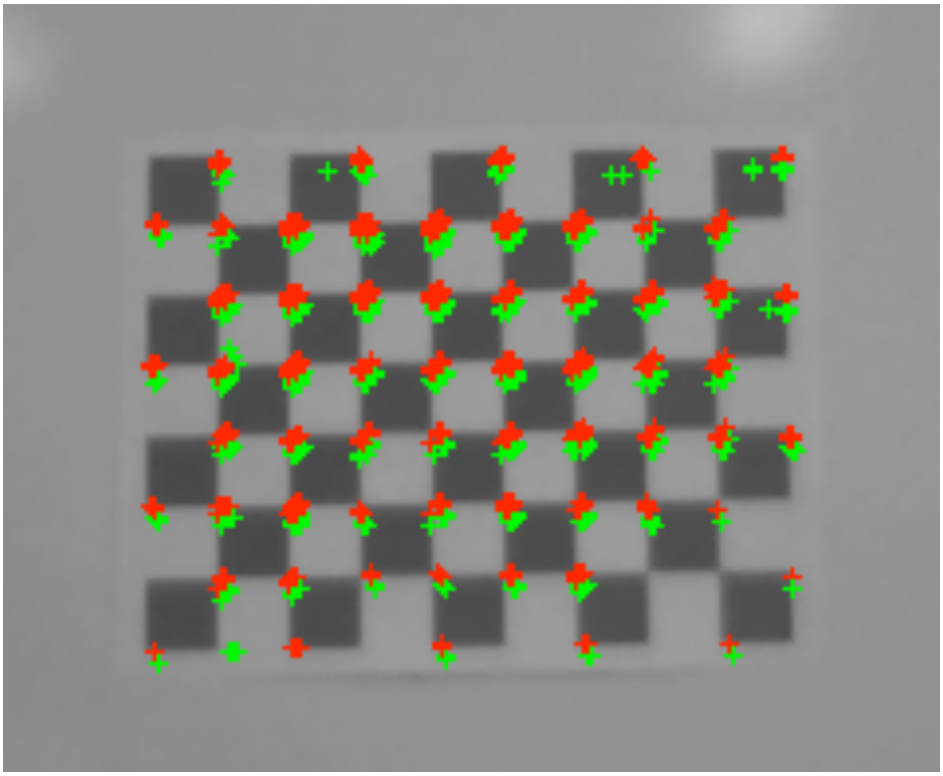

Figure 6.10: Calibration checkerboard pattern demonstrating the feature detection and correlation algorithms implemented by the vision system implementation. The red crosses identify where features are in the current frame while green crosses identify where the features were correlated to in the previous frame. In this image, the camera was panning down to illustrate that the correlated points (green) in the previous image follow the location of the feature in the current image (red).

Chapter 7

Conclusion

7.1 Summary

The purpose of this work was to present a vision system suitable for onboard aerial vehicle guidance. The applications targeted for such a system were outlined to include smart sensors or delivery vehicles. The problems associated with existing methods, primarily excessive latency, were described as unacceptable for any real-time vision system and that an onboard vision system was required.

To guide the reader through the development process, first a systematic evaluation of potential computing platforms was presented, which concluded an FPGA-based platform was the preferred solution. An embedded image processing primer was presented that covered the basic properties of image sensors that would have impact on a mobile vehicle image system. Included also in this primer was an example-based explanation of the techniques available to implement image processing algorithms on FPGA-based platforms.

Two applications were presented to fulfill the intended objectives of this work. The first application was the development of a vision system to facilitate vision guided landing. In this application, color segmentation and connected component algorithms were utilized to locate and track a single color landing site marker. Multiple landing attempts were shown to be successful including the landing of the UAV in the back of a moving pickup truck.

The second application presented was a feature tracker sub-system. The intent of this system was to provide a base level system that could be used in a variety of applications. The hardware components of this system were primarily comprised of a Harris feature detector and a correlator core that performed template matching to track locations between successive image frames. While this sub-system was not used in a flight test situation, it was shown to be capable of successfully identifying and tracking multiple features.

7.2 Contributions

The physical Helios computing platform (i.e., the main circuit board and daughter cards) was previously developed in the Robotic Vision Lab at Brigham Young University. However, at the time that the work on this thesis began, there was not much developed as far as an embedded operating system or customized FPGA logic cores beyond some low level device interfaces (Frame Sync, Camera Serial, USB chip, GPIO). Consequently, most of the effort described in this thesis was to create a foundation level embedded image processing platform. One goal of this work, in addition to the applications described herein, was to create a base platform that could be easily adapted by future users to their particular needs. In fact, at the time of this writing, this has already occurred.

The contributions that the author made can be broken down into three areas: Hardware, Embedded OS, and Algorithms. A description of each of the individual contributions under these categories is set forth below.

7.2.1 Hardware

The 'Hardware' contributions refer to FPGA logic cores developed as a part of this work. As described previously, these may perform all or only a portion of the total algorithm.

PLB Camera Core

The purpose of the camera core is to capture incoming images from a camera, run any inline or cached inline image processing algorithms, and store them in memory. To store the images in memory, the core must communicate over a system bus in the FPGA. At the beginning of this work, the only camera core that existed used a simple and low performance OPB interface, which stands for On-chip Peripheral Bus. This bus was sufficient for transferring small images (e.g., 320×240) at low frame rates, but it did not have enough bandwidth to carry larger images (e.g., 640×480) at high frame rates. Therefore, the author developed a camera core that interfaced with the PLB (Processor Local Bus) which could sustain the required data rates.

HSV Color Conversion Core

It was discovered through experience by the author that varying lighting conditions can greatly disturb image processing algorithms that run on RGB color space images. Therefore, the first hardware core that the author developed as a part of this work was a RGB to HSV color conversion core. This core was absolutely essential for the color segmentation work described in Chapter 5. The H, S, and V values computed by this core had an error margin of ± 2, which has minimal impact to the machine vision algorithms described in this work.

HSV Segmentation Core

As described in Chapter 5, performing the HSV segmentation in hardware provided a significant performance gain to the application. This core was a simple core to develop and only compares incoming pixel values to multiple color threshold values that can be set at runtime. The core can segment up to 8 different colors simultaneously.

Connected Components Streak Finder Core

While multiple approaches to compute connected components exist, the one most suited for FPGA implementation is broken down into two steps: streak finding and then streak connecting. Streak finding attempts to find continuous horizontal streaks in a binary image, which are described by a row, column start location, column end location, and the color of the streak, each of which are 2-byte values. These values were then transferred to main memory for the software portion of the algorithm to connect the streaks. This was perfectly suited for FPGA implementation due to the inline nature of the algorithm.

Harris Feature Detector Core

The Harris feature detector core implements the Harris feature algorithm described in Equations 6.1 and 6.2. This is a cached inline implementation due to the 3×3 matrix needed for the computations. Included in the feature core is a feature filter that will suppress any features that are below a runtime settable threshold.

Correlator Core

The purpose of the correlator core is to perform a template matching search process on a list of interested points. The interested points are those found by the Harris feature detector. This core consisted of various sub-components that were significant units in and of themselves. These sub-components were a muti-port SRAM controller, dispatcher, and individual processing elements that performed the template matching. One interesting note about this core is that when activated it increased the power consumption of the Helios board 33% from 3 to 4 Watts. It was also designed, implemented, and functioning correctly within one week.

USB Core Improvements

When this work began there already existed a USB core that was capable of transferring single values at a time over the USB bus. While this was sufficient for small diagnostic data, it was too slow to transfer image data. Therefore the author added 'burst' functionality to allow the USB core to directly transfer image data using DMA from Helios main memory to the USB bus. This enabled a 640×480 video rate of over 50 frames-per-second to be transmitted over the USB cable to a host computer. The USB core was also improved to allow USB hot swapping as the prior implementation required the board to be shut down and restarted every time the cable was attached or detached.

Technique Framework

When this work began the author was not well versed in image processing implementation techniques on FPGAs. Consequently, hardware development was an iterative process as new design approaches were discovered. To provide a framework with which to categorize implementation techniques and speed up the design phase the author defined a classification of the methods available to implement image processing algorithms in FPGA hardware. These three defined categories were: inline, offline, and cached inline.

7.2.2 Embedded Software Framework

Since the Helios platform utilizes an FPGA with one or two embedded PowerPC CPUs, those CPUs are used as the control center that initializes and manages the hardware cores as well as performs some of the image processing algorithms. At the beginning of this work, the existing software framework created for the Helios only possessed minimal support infrastructure. Much effort was made to expand the 'round-robin with interrupts' [19] style system with frameworks and libraries to support the image processing tasks that were performed on it. These are explained in greater detail below.

Primitive Structures

One of the most basic frameworks that is required in any CPU system is one that manages allocatable memory. In an embedded environment this required a Buffer Framework that would allocate static amounts of main memory, which was in limited supply, and use a checkout-checkin style policy to allow onboard algorithms the ability to utilize varying amounts of memory during runtime.

In a 'round-robin with interrupts' software approach, the main loop does not cycle at a constant rate. Therefore a Loop Scheduler Framework was implemented to allow specific functions to be scheduled to occur at specific time intervals.

Communication Libraries

An essential but not specifically identified requirement of the embedded image processing platform is the ability to communicate with the user and other devices. To accomplish this, a Communication Framework was implemented featuring packet-based transmission and reception of data that is verified using an Adler-32 checksum.

Frame Table

Much of image processing is centered around performing multiple algorithms or decisions on a single image frame. To facilitate the 'process' of image processing a Frame Table Library was created. This library provided a single place to store raw images, color segmented connected components, Harris feature lists, and other bookkeeping information.

Matrix Library

While developing an implementation of a RANSAC (Random Sample and Consensus) homography estimation algorithm in the feature tracking application described in Chapter 6, it became evident that a general purpose Matrix computation library was needed. The Matrix library implemented by the author was modeled after that found in the Open Computer Vision Library (OpenCV).

HIO GUI

The GUI that was developed to interface with the Helios platform was similar to some of the other framework contributions in that it doesn't receive much attention in the application chapters, but required significant effort to build and constantly adapt to support new features. This GUI is a multithreaded windows application developed using Microsoft Visual Studio© and is currently in its 4^{th} generation. In addition to interfacing with the Helios, a simulator was also built into the GUI to provide a more rapid environment to test feature tracking and homography estimation algorithms.

7.2.3 Software Algorithms

The software algorithms developed as part of this work for the embedded PowerPC were minor when compared to the infrastructure development efforts described in Section 7.2.2. However, they were a required part of the image processing applications described in Chapters 5 and 6. These are described below.

Connected Components

While most of the repetitive computations of the connected components algorithm were handled by a hardware core, the streaks in the image were searched and connected in a software. This algorithm performed all the searching and connecting operations 'in place' without requiring any additional memory utilization beyond that to store the streak structs.

RANSAC Homography Estimation

In addition to the simple tracking of feature points between images described in Chapter 6, a RANSAC algorithm was implemented to estimate the translational homography between two sequential image frames. This was intended to be used as part of a See-And-Avoid algorithm for a UAV, but was left for future work.

7.3 Future Work

The recommendations for future development are primarily improvements to the Helios platform that now are within reach. These are described below.

- Onboard GPS. Currently Helios requires the use of the global positioning sensor on the autopilot for geo-localization. Adding an interface to a GPS sensor and the necessary support software frameworks would provide a useful tool for UAV applications.

- Onboard Gyroscope. Similar to the need for GPS data, an onboard Gyroscope would allow Helios to estimate positional attitude, which would allow image coordinates to be translated to ground-based coordinates.

- Gimbaled Camera. Currently the camera can only be mounted in a fixed position. By integrating the control of a gimbal in which the camera could be mounted, the image processing algorithms could search, find, and track objects in flight regardless of the flight path of the UAV.

- LVDS Camera. The current video image sensor is connected using a short and rigid 22-pin parallel data cable, which limits its placement to a fixed location within only a few inches of Helios. A LVDS (Low Voltage Differential Signaling) cable would allow the placement of the camera at any remote location on the UAV. For example, one camera could be placed on each wing tip and used for stereo vision applications.

- HD Camera. The current image sensor used on the Helios is a standard definition 640×480 camera. An HD camera would provide higher detailed images, which may improve the accuracy of image processing algorithms.

- Higher Frame Rate Camera. While in flight, even small movements can cause a large shift in the camera's field of view. A higher frame rate camera, implies less time between image captures, which reduces the possible change that must be accounted for in tracking algorithms.

- Camera Simulation Device. At present, runtime testing is crudely performed by placing various items in front of the camera and visually inspecting the output. If there were a device that connected to the camera input of the Helios board and a desktop computer, a predefined set of images could be fed directly into the Helios, which would allow repeatable testing to be performed after changes were made to the image processing cores or software algorithms.

- Bit-Level Accurate Simulator. Hardware development is slow and embedded systems are difficult to debug. A bit-level accurate simulator running on a desktop would allow more rapid algorithm development.

- Inflight Communication. At present the only way to communicate with the Helios while in flight is through the Autopilot. A 2-way communication device would allow the ground-based user to instruct the Helios to track some visually identifiable object.

- Wireless Video Transmission. At present, while in flight there is no way for the user on the ground to 'see' what the UAV is 'seeing.' If the Helios could convert raw or processed images into an interlaced NTSC stream, it could be transmitted using commercially available wireless video transmitters. The most significant impact of this is not the ability to transmit the raw camera images, but the processed images such as the color segmented or Harris feature images.

Bibliography

[1] V. Chitrakaran, D.Dawson, J. Chen, and M. Feemster, "Vision assisted autonomous landing of an unmanned aerial vehicle," *Proceedings of the IEEE Conference on Decision and Control*, pp. 1465–1470, December 2005. 9, 45

[2] J. Hintze, D. Christian, C. Theodore, M. Tischler, and T. McLain, "Automated landing of a rotorcraft UAV in a non-cooperative environment," in *Proceedings of the 60th Annual Forum of the American Helicopter Society*, Baltimore, Maryland, June 2004. 9

[3] S. Saripalli, J. F. Montgomery, and G. S. Sukhatme, "Visually guided landing of an unmanned aerial vehicle," *IEEE Transactions on Robotics and Automation*, vol. 19, no. 3, pp. 371–380, June 2003. 9, 10

[4] O. Shakernia, Y. Ma, T. J. Koo, J. Hespanha, and S. S. Sastry, "Vision guided landing of an unmanned air vehicle," *Proceedings of the IEEE Conference on Decision and Control*, pp. 4143–4148, December 1999. 9

[5] C. S. Sharp, O. Shakernia, and S. S. Sastry, "A vision system for landing an unmanned aerial vehicle," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 1720–1727, May 2001. 9, 10

[6] B. B. Edwards, W. S. Fife, J. K. Archibald, D.-J. Lee, and D. K. Wilde, "A design approach for small vision-based autonomous vehicles," *Proceedings of the SPIE International Conference on Intelligent Robots and Computer Vision*, vol. 6384, October 2006. 10, 11

[7] S. Saripalli, J. F. Montgomery, and G. S. Sukhatme, "Vision-based autonomous landing of an unmanned aerial vehicle," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2799–2804, May 2002. 10

[8] O. Shakernia, R. Vidal, C. S. Sharp, Y. Ma, and S. Sastry, "Multiple view motion estimation and control for landing an unmanned aerial vehicle," in *Proceedings of the 2002 IEEE International Conference on Robotics & Automation*, Washington DC, May 2002, pp. 2793–2798. 10

[9] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to programmable active memories," in *DEC Paris Research Laboratory, Tech. Rep.3*, 1989. 10

[10] J. Cho and et al., "A real-time object tracking system using a particle filter," in *Proc. of IEEE/RSJ Conf. on Intelligent Robots and Systems*, October 2006. 10

[11] D. Masrani and W. MacLean, "A real-time large disparity range stereo-system using FPGAs," in *Proc. of IEEE Conf. on Computer Vision Systems*, January 2006. 10

[12] J. Woodfill and B. V. Herzen, "Real-time stereo vision on the PARTS reconfigurable computer," in *Proc. of IEEE Symp. on FPGAs for Custom Computing Machines*, April 1997. 10, 11

[13] Y. Jia, X. Zhang, M. Li, and L. An, "A miniature stereo vision machine (MSVM-III) for dense disparity mapping," in *Proc. of the IEEE Int. Conf. on Pattern Recognition*, August 2004, pp. 728–731. 11

[14] W. S. Fife and J. K. Archibald, "Reconfigurable on-board vision processing for small autonomous vehicles," *EURASIP Journal on Embedded Systems*, 2007, article ID 80141. 11, 19, 64

[15] S. Fowers, D.-J. Lee, B. Tippetts, K. D. Lillywhite, A. Dennis, and J. Archibald, "Vision aided stabilization and the development of a quad-rotor micro UAV," in *IEEE International Symposium on Computational Intelligence in Robotics and Automation*, 2007. 11

[16] B. Edwards, J. Archibald, and W. Fife, "A vision system for precision MAV targeted landing," in *IEEE International Symposium on Computational Intelligence in Robotics and Automation*, June 2007. 11

[17] C. Harris and M. Stephens, "A combined corner and edge detector," in *4th Alvey Vision Conference*, 1988, pp. 147–151. 65, 67

[18] R. Zabih and J. Woodfill, "Non-parametric local transforms for computing visual correspondence," in *3rd European Conference on Computer Vision*, 1994, pp. 151–158. 66

[19] D. E. Simon, *An Embedded Software Primer*. Addison-Wesley, 1999. 87