Real-time Image Processing on Low Cost Embedded Computers

Sunil Shah

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-117
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-117.html

May 20, 2014

Acknowledgement

Without the generous support of 3DRobotics and advice from Brandon Basso, this project would not have been possible. Additionally, several other graduate students collaborated on and contributed extensively to this project: Constantin Berzan, Nahush Bhanage, Gita Dombrowski and Hoang Nguyen.

University of California, Berkeley College of Engineering

MASTER OF ENGINEERING - SPRING 2014

Electrical Engineering and Computer Science

Robotics & Embedded Software

REAL-TIME IMAGE PROCESSING ON LOW COST EMBEDDED COMPUTERS

SUNIL SHAH

This **Masters Project Paper** fulfills the Master of Engineering degree requirement.

Approved by:

1. Capstone Project Advisor:

Signature: _____ Date _____

Print Name/Department: RAJA SENGUPTA/CIVIL AND ENVIRONMENTAL ENGINEERING

2. Faculty Committee Member #2:

Signature: _____ Date _____

Print Name/Department: PIETER ABBEEL/ELECTRICAL ENGINEERING AND COMPUTER SCIENCES

Abstract

In 2012 a federal mandate was imposed that required the FAA to integrate unmanned aerial systems (UAS) into the national airspace (NAS) by 2015 for civilian and commercial use. A significant driver for the increasing popularity of these systems is the rise in open hardware and open software solutions which allow hobbyists to build small UAS at low cost and without specialist equipment. This paper describes our work building, evaluating and improving performance of a vision-based system running on an embedded computer onboard such a small UAS. This system utilises open source software and open hardware to automatically land a multi-rotor UAS with high accuracy. Using parallel computing techniques, our final implementation runs at the maximum possible rate of 30 frames per second. This demonstrates a valid approach for implementing other real-time vision based systems onboard UAS using low power, small and economical embedded computers.

1 Introduction

To most, the rapid deployment of unmanned aerial systems (UAS) is inevitable. UAS are now vital to military operations and, as the hobbyist multi-rotor movement takes off, the colloquialism *drone* has become a frequent reference in popular culture. Hobbyists can buy vehicles off-the-shelf for less than a thousand US dollars and fly them almost immediately out of the box.

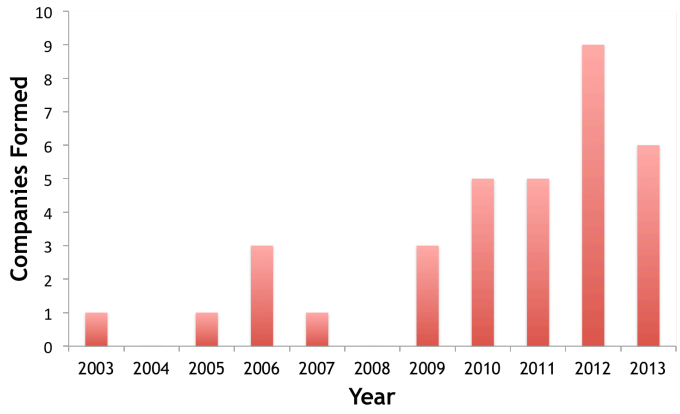

Figure 1: UAS companies incorporated by year (own data).

In 2012, President Obama imposed a federal mandate that the Federal Aviation Administration (FAA) must propose rule changes to integrate UAS into the National Airspace (NAS) by 2015 [15]; currently all non-military and non-government flight is prohibited. While the incumbent defense companies are prepared for this approaching rule change, many nascent businesses and established technology companies are looking to gain a foothold in this market. In particular, there has been a dramatic growth in the number of startups looking to provide UAS products and services (as shown in figure 1), as well as previously unprecedented acquisitions of UAS businesses by technology behemoths like Google and Facebook. Amazon recently made use of UAS for marketing purposes and has devoted an entire team to researching their potential integration into their business. This growing commercial interest in the use of UAS extends from applications such as precision agriculture [7] [21] to package delivery. Industry association AUVSI predicts that integration of UAS will cause $82 billion of economic activity [4].

The rise in hobbyist interest in UAS has been driven by several factors: 1) the appearance of affordable prototyping tools (such as the Arduino platform and 3D printers); 2) affordable sensors (such as the

accelerometers used in smartphones); and 3) growth in popularity of open source software. The growth in recent years of online communities centred around hardware and software which facilitate interaction with the physical world - for example, the Arduino platform, have made it possible for others to build robotics applications on top of relatively stable and well understood hardware. DIYDrones, the foundation for our industrial sponsor 3DRobotics' business, is one of these. As an online community formed around the creation of an Arduino-based open source autopilot called ArduPilot, DIYDrones has become a force majeure. For just a few hundred dollars, a motivated hobbyist is now able to buy the necessary *intelligence* to fly a model aircraft autonomously.

Unfortunately, the simple and economical design of these boards causes lacklustre performance. While the ArduPilot project has moved on from the Arduino platform, their next generation autopilot, the PixHawk, is still insufficient for complex processing tasks, running at a mere 168 MHz with just 256 kilobytes of memory [1]. While it is possible to write highly optimised code for this platform - it is time-consuming for users looking to quickly prototype a sensing application. Additionally, the limited memory available makes it infeasible to re-use open source libraries without significant manual manipulation during installation.

This Master's project, therefore, extends the concept of a 'co-computer' from the research of the Cyber-Physically Cloud Computing lab in the Civil Systems department at UC Berkeley. Their research typically involves integration of various subsystems to produce UAS suited for particular applications such as the tracking of pedestrians or collaborative sensing. Their systems offload heavy sensor data processing to a secondary *co-computer* [12], usually a much faster computer with a traditional x86 instruction set processor. This design allows the UAS to compute advanced computer vision and robotics algorithms onboard in real-time.

This approach is commonplace amongst the design of computational systems for complex robots. Stanford's DARPA Grand Challenge winner, the autonomous car Stanley, used six rack-mounted computers which individually handled processing of various subsystems that carried out localisation, sensor fusion, planning and other tasks [20]. It is almost certain that FAA certification of UAS for commercial flight will be predicated on having redundant subsystems that allow the UAS to maintain control if higher level functions fail.

Because of their specialist use cases, the computers traditionally used are often expensive and typically cost over a thousand dollars at current prices. Furthermore, using processors intended for desktop use requires larger and hence heavier cooling equipment, meaning that their weight and size renders them infeasible as payload on hobbyist sized multi-rotor UAS. Furthermore, their power draw is significant at between 60 to 80 watts.

This paper therefore concentrates on the increasingly prevalent open source ARM-based embedded computers. We focus on the popular BeagleBone Black, a single core ARM Cortex A8 board that costs just $45, and the Hardkernel Odroid XU, an octo-core ARM Cortex A15 board that costs $169 but is considerably more powerful. Each of these boards draws less than 20 watts of power and is easily carried on small multi-rotor aircraft.

We build and implement a vision-based automated landing system based on the work of Sharp, Shakernia, and Sastry in 2001 [18]. We follow a similar approach to theirs but optimise for re-use of popular open source projects such as OpenCV, a computer vision library and ROS, a middleware system to allow a modular software architecture. We then focus on improving the performance of this system - by first optimising the *hotspots* inherent in our algorithm, and then by utilising processor specific optimisations and parallel computing techniques to maximise throughput.

Automated landing is an interesting problem with a very clear commercial application. If UAS are to be used in urban environments for tasks such as package delivery, it is necessary for them to be able to land accurately. The current state of the art relies upon localisation using GPS. Our testing using the built-in *return-to-launch* mode yielded a mean accuracy of 195.33 cm, with a standard deviation of 110.73 cm over 10 launches. While this may be sufficient for landing in an uninhabited field, it is certainly not sufficient for landing in an inhabited area with spatial constraints.

The next section surveys the prior work in this area. Section 3 describes the design methodology and the optimisations implemented. The results of our work and a discussion of these are outlined in section 4. Finally, section 5 covers concluding remarks.

2 Prior Work

This project draws upon prior work in two different disciplines. Firstly, we consider accurate vision based landing of a UAS, for which there have been several published approaches.

For this project we work with vertical takeoff and landing (VTOL) multi-rotor UAS - since these are the most popular type of UAS used by hobbyist users and nascent commercial entities. Multi-rotors have become popular in the recent past; traditional approaches to automated landing of VTOL aircraft are modelled on the automated landing of helicopters.

The automated landing problem was investigated in the research literature of the early 2000s. Sharp, Shakernia, and Sastry [18] designed an approach for automatically landing an unmanned helicopter. Their landing target uses a simple monochromatic design made up of several squares. Onboard the helicopter, they use a pan-tilt-zoom camera and two embedded computers with Intel CPUs. They discuss the details

of their approach to pose estimation, but omit the details of the helicopter controller. Using a real-time OS and highly optimised custom code, they are able to get their vision system to operate at a rate of 30 Hz.

Saripalli, Montgomery, and Sukhatme [17] designed another approach for automatically landing an unmanned helicopter. They use a monochromatic H-shaped landing target. Their onboard vision system detects this landing target and outputs the helicopter's relative position with respect to it. This is sent wirelessly to a behavior-based controller running on a ground station, which then directs the helicopter to land on top of the target. They are able to run their controller at 10 Hz this way. They are also using a high-accuracy differential GPS system, and it is not clear how much their differential GPS and vision systems contribute to a successful landing.

Garcia-Pardo, Sukhatme, and Montgomery [6] look at a more general problem, where there is no pre-specified landing target, and their helicopter has to search autonomously for a suitable clear area on which to land.

The second discipline we draw upon is that of high performance embedded computing on reduced instruction set computers, such as those implementing the ARM architecture. Several efforts have been made to explore the effect of parallelising certain robotics applications but these typically involve the use of general purpose computing on the GPU. This doesn't translate well to embedded computers due to the lack of vendor support for graphics chips that are provided. These chips often don't support heterogenous parallel programming languages, such as OpenCL or NVidia's CUDA.

However, there are several efforts looking at optimising performance for ARM-based processors [9]. This is driven by growing smartphone usage, nearly all of which use ARM processor designs. Qualcomm, in particular, provides an ARM-optimised computer vision library for Android called FastCV. While this is optimised for their own series of processors, it does have generic ARM optimisations that are manufacturer agnostic. Efforts have been made to explore OpenCV optimisation for real-time computer vision applications too [13].

Finally, it should be noted that this project isn't the first attempt to use an open source embedded computer on a UAS. Many hobbyists experiment with these boards, as noted in the discussion threads on DIYDrones.

A San Francisco based startup, Skycatch Inc., uses a BeagleBone Black to provide a custom runtime environment. This allows their users to write applications on top of their custom designed UAS in scripting language JavaScript. While the user friendliness of this approach is evident, it is also clear that using a high level interpreted application results in a tremendous loss of performance which makes it impossible to do all but the most basic of image processing in real-time. This implementation is also

closed source.

Other commercial entities, such as Cloud Cap Technologies, provide proprietary embedded computers running highly customised computer vision software. However, these cost many thousands of dollars and are difficult to *hack*, making them impractical for research and startup use.

Ultimately, this project's contribution is to demonstrate the tools and techniques that can be used to implement highly performant vision algorithms onboard a UAS using low-cost open source hardware and open source software.

3 Method & Materials

The intention with this project was to re-use readily available hardware and software as much as possible. Along those lines, we made the decision to use several open source libraries and, as previously mentioned, two open source embedded computers.

While specific implementation details may only be covered briefly in this paper, detailed instructions, notes and the full source code is available online under the GNU Lesser General Public License at `https://github.com/ssk2/drones-267`.

3.1 Hardware Architecture

3.1.1 Autopilot

Testing was carried out with version 2.6 of 3DRobotics' APM autopilot. This runs their open source ArduCopter autopilot software for VTOL UAS. A full specification is available online [2]. Using a USB cable, we are able to exact high level control of the aircraft using the MAVLink protocol, described further in section 3.2.1.

3.1.2 Embedded Computer

Our vision system was implemented on two popular open hardware boards which are both community designed and supported. Documentation is freely available and the boards themselves are produced by non-profit entities.

We considered boards that could run the entire Linux operating system - for ease of setup and flexibility in software installation. This immediately discounted the archetypical "Arduino" family of boards - since these are not general purpose computers and cannot run a full installation of Linux. The alternatives are primarily ARM processor based boards - for their low cost processors which have efficient power utilisation [16].

Our first choice was the Beagleboard's BeagleBone Black, a single core computer not dissimilar to the Raspberry Pi, albeit with a faster processor. This quickly proved to be underpowered and we then migrated to the faster, multi-core HardKernel Odroid XU. The specification for each of the boards we used is outlined in table 1.

3.1.3 Peripheral Hardware

A considerable amount of research went into selecting and integrating low-cost embedded hardware. In particular, it was necessary to trial several different wireless adaptors and USB hubs to find those that

Table 1: Summary of board specifications

Board	**BeagleBone Black**	**Hardkernel Odroid XU**
Processor	AM335x 1GHz ARM Cortex-A8	Exynos5 Octa Cortex-A15 1.6Ghz quad core and Cortex-A7 quad core CPUs & Zynq-7000 Series Dual-core ARM A9 CPU
Memory	512 MB	2 GB
Storage	2GB onboard & MicroSD	e-MMC onboard (configurable)
	USB 2.0 client	USB 3.0 host x 1
	USB 2.0 host	USB 3.0 OTG x 1
Ports	Ethernet	USB 2.0 host x 4
	HDMI	HDMI
	2x 46 pin headers	Ethernet
Cost	$45	$169

had good driver support in Linux and provided enough power to peripheral devices (such as the camera). Our architecture is shown in figure 2.

Due to hardware limitations, connecting a computer to the APM autopilot via USB turns off the wireless telemetry that is natively provided. Therefore, it was necessary to set up and configure the embedded computer as an ad-hoc wireless network to allow us to receive telemetry and debugging data while testing.

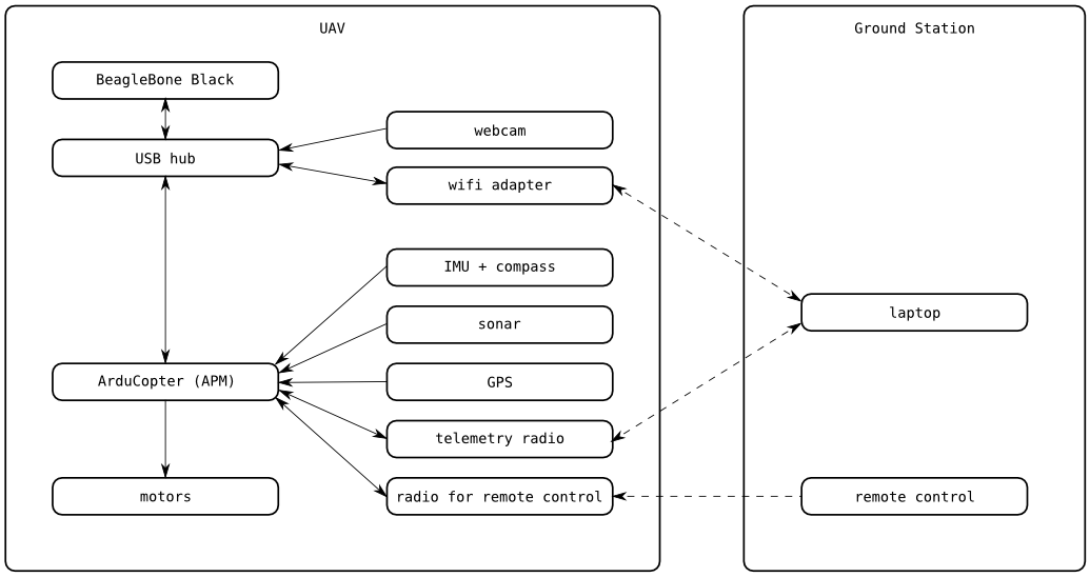

Figure 2: Architecture of our automated landing system. We use inexpensive off-the-shelf hardware. The laptop and remote control are for monitoring and emergency takeover by a human pilot. All the computation is performed onboard the UAS.

3.1.4 Camera

Our testing rig and vehicle was equipped with a Logitech C920 USB web camera. This is a high end consumer product that has higher than average optical acuity, good driver support on Linux and a global shutter. (Cheaper web cameras use rolling shutters which allow increased low light sensitivity at the cost of a slower shutter speed. This is acceptable when there is not significant motion in the frame but unsuitable for this application.)

3.1.5 Integrating Co-computer

It was necessary to design a special rack to integrate the co-computer onto our UAS. Figure 3 shows our design to allow for adequate ventilation and the requisite connections to power and the autopilot.

3.2 Software Design

3.2.1 Operating System and Library Setup

On each of these boards, we installed a supported (manufacturer supplied or recommended) variant of Ubuntu Linux, ROS Hydro and OpenCV from source. Where possible, we enabled support for the ARM specific NEON single instruction, multiple data extensions to increase performance [19].

In line with 3DRobotics' open source ethos, we re-used existing libraries where possible. Substantial progress has been made on the **Robotics Operating System** (ROS), a framework and set of libraries that allow for a highly modular architecture with a natively supported robust publish/subscribe messaging system [14]. ROS also provides simple scheduling mechanisms to let processing happen in an event driven manner or at a fixed interval (e.g. 10Hz).

Using ROS allowed us to separate components into separate *ROS nodes*. While this makes it easier to group similar code together, it will also make it trivial in the future to move individual nodes onto heterogeneous boards connected by TCP/IP. Figure 4 shows how our code was modularised.

OpenCV, a computer vision library, is extremely popular and has considerable functionality relevant to this project. It also supports Video4Linux, a project to support common video capture devices in Linux. While other computer vision libraries exist, OpenCV is the most popular and hence is best supported online [10].

Finally, we adapted **roscopter**, a compact ROS package that allows serial communication with devices supporting the MAVLink protocol (a standardised protocol used for communication to and from autonomous flight controllers or autopilots). This allows us to effect control over the autopilot from our co-computer.

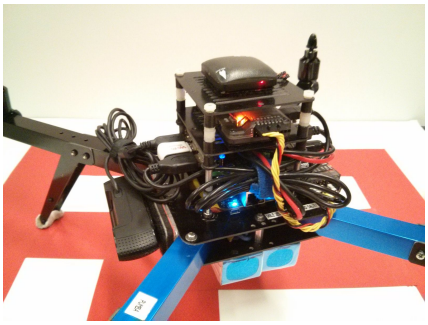

Figure 3: Our hardware stack fully assembled. Total weight excluding batteries is 1.35 kg.

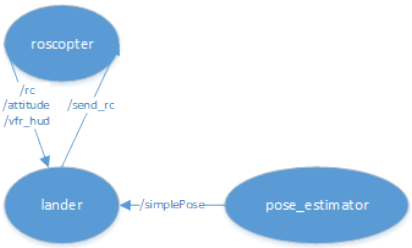

Figure 4: ROS nodes and topics for exchanging messages.

Programming Language Our choice of programming language was guided by framework support and the need for performance when running on our boards. ROS has the most limited official support, having bindings for just C++ and Python. Given the known poor performance of embedded Python on ARM processors, our modules were implemented in C++.

3.3 Automated Landing

The implemented automated landing system was based on the pose estimation algorithm described by Sharp et al. [18]. The following section describes our implementation of their approach and optimisations to it.

3.3.1 Landing Pad Design

This particular algorithm requires the landing pad used for pose estimation to have a known pattern. In this case, our design is a monochromatic design consisting of five squares within a sixth, larger, square. The proportion of these squares to each other is known. For this pattern to be visible at higher altitudes, it must be large. This causes issues at low altitude when the entire pattern cannot be captured within the field of view of the camera. We discuss workarounds to this in section 4.2.2. Figure 5 shows our landing pad and corner detection in action.

3.3.2 Vision Algorithm Overview

The overall structure of our vision algorithm is shown in figure 6.

Corner Detection As a first step, we detect the corners of the landing platform in an image, shown visually in figure 7:

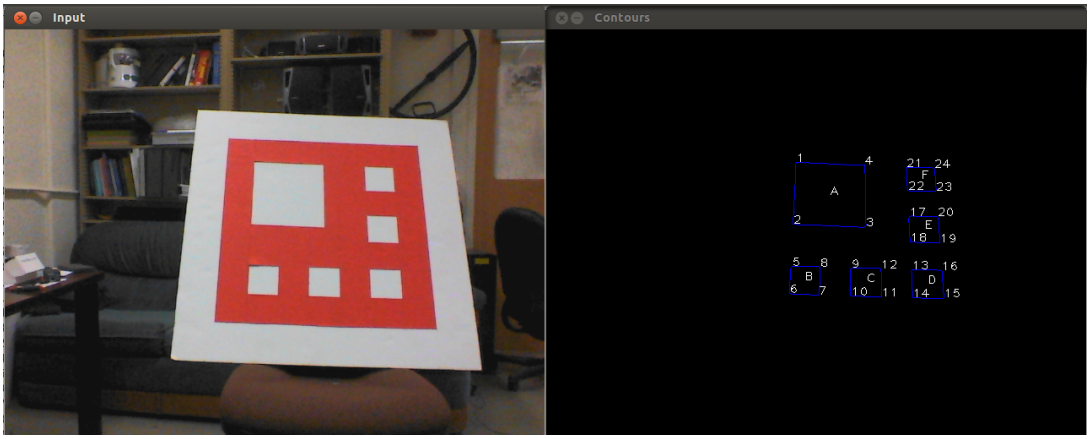

Figure 5: Left: Design of our landing platform. Right: Output of the corner detector (24 points, in order).

1. **Median Blur, Canny Edge Detection** Denoise the image using a 3x3 median filter, and pass it through the Canny edge detector.
2. **Find Contours** Identify contours and establish a tree-structured hierarchy among them.
3. **Approximate Polygons** Discard contours which are not four-sided convex polygons and which have an area less than an experimentally determined threshold value. We look for four-sided polygons and not specifically for squares, since they will not appear as squares under perspective projection.
4. **Get Index Of Outer Square** Using the contour hierarchy, determine a contour which contains 6 other contours. This contour represents the boundary of our landing platform. Store coordinates of the corners of these 6 inner contours.
5. **Label Polygons** Label the largest of the 6 polygons as 'A' and the farthest one from 'A' as 'D'. Label polygons as 'B', 'C', 'E' and 'F' based on their orientation and distance relative to the vector formed by joining centers of 'A' and 'D'.
6. **Label Corners** For each polygon, label corners in anti-clockwise order.

Pose Estimation We define the origin of the world coordinate frame to be the center of the landing platform, such that all points on the landing platform have a Z coordinate of zero. The corner detector gives us image coordinates for the 24 corners. Thus, we have a set of 24 point correspondences between world coordinates and image coordinates. Given this input, we want to compute the quadcopter's pose, i.e. the position and orientation of the camera in the world coordinate frame. To do this, we followed the approach of Sharp et al. [18], whose details are omitted here for brevity. We use SVD to approximately solve a linear system of 48 equations with 6 degrees of freedom.

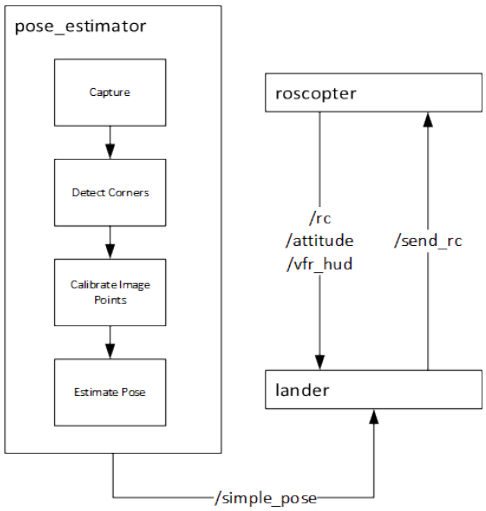

Figure 6

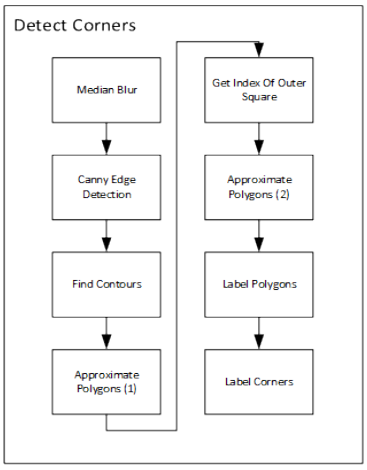

Figure 7

The output from the pose estimator is a translation vector $t = \begin{bmatrix} t_x & t_y & t_z \end{bmatrix}^\top$ and a 3x3 rotation matrix R. We compute the camera position in world coordinates as $C = -R^\top t$, and the yaw angle as $\alpha = \arctan(R_{21}/R_{11})$. (The roll and pitch angles can be computed similarly, but we do not require them in the control algorithm.)

The approach above assumes a calibrated pinhole camera. For the pose estimates to be meaningful, our camera had to be calibrated first. We calibrated our camera using the `camera_calibration` tool provided in the OpenCV tutorials, plus some manual tuning. We used the resulting calibration matrix to convert the raw pixel coordinates into coordinates for a calibrated pinhole camera model, which we then fed into the equations above.

3.3.3 Real-time Control

In order to actually land a vehicle using these pose estimates, it was necessary to implement a high-level controller which worked in conjunction with the autopilot's own stabilisation modes.

Our controller takes the form of a state machine, illustrated in Figure 8. The UAV starts out in the *FLYING* state. When landing is desired, it switches into the *SEEK_HOME* state. This uses the autopilot's *return-to-launch* mode to bring the UAV close to the original takeoff location, using GPS. When the landing platform becomes visible, the UAV switches into the *LAND_HIGH* state. Here we use our vision-based pose estimates with a simple proportional controller to guide the UAV towards the landing platform. (The error terms in our controller are given as x, y, z, and yaw deviations. The controller descends at a fixed rate, using the z deviation only as an estimate of the altitude.) When the UAV reaches a predefined altitude (where pose estimates are no longer possible, due to limited field of

view), our controller enters the *LAND_LOW* state, and descends slowly by dead reckoning. When the barometric pressure sensor indicates that the UAV has reached the ground, the controller switches into the *POWER_OFF* state.

Due to the interface provided by roscopter, our control input consists of the raw pulse width modulation (PWM) values that would typically be read from the human pilot's radio-control receiver. For instance, a value of 900 represents no throttle, whereas 1800 represents full throttle. By overriding these values, we can simulate human control input from our controller. This crude approach is obviously limited - a better approach would be to extend the MAVLink protocol with a new message type to allow for error values to be sent directly into the inner control loop of the autopilot.

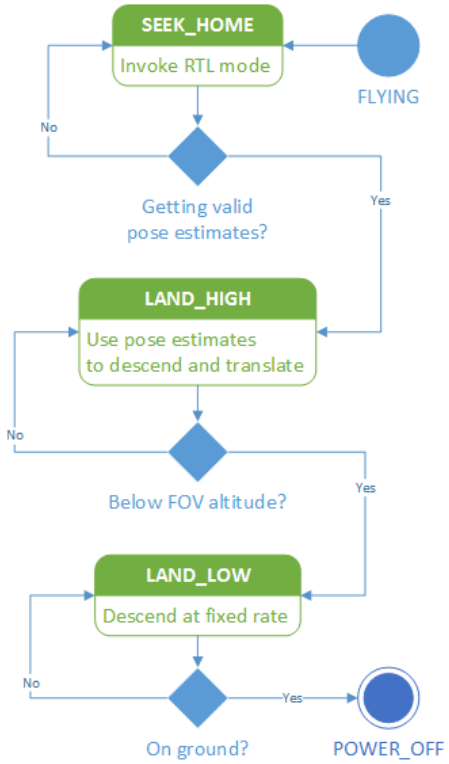

Figure 8: State diagram of our landing controller.

3.3.4 Optimisation Techniques

Our initial implementation operated at less than 3 Hz (i.e. it calculated pose estimates less than 3 times a second). This is far too poor for real-time control. The control loop of our autopilot operates at 10 Hz, taking sensor input from the GPS sensor at a rate of 5 Hz. In order to precisely control the UAS, it is necessary to optimise our solution significantly such that it provides us with estimates at least at 5 Hz. The following optimisation methods described are nearly all applicable to other vision based systems.

Identifying Hotspots A full outline of our benchmarking technique is described in section 4.3.1. At each stage of optimisation, we ran the pose estimation implementation through the same set of tests to identify hotspots - areas which took the majority of processing time. Figure 9 shows hotspots inherent in our overall process and figure 10 shows hotspots within the "Detect Corners" subroutine.

Removing Redundancy An obvious optimisation to our initial approach was to identify function calls and substeps that were unnecessary. Through benchmarking, we attempted to remove or reduce the impact of calls which were taking a significant amount of processing time. At each stage, we ensured that robustness to image quality was retained by testing against approximately 6,000 images captured in the lab and in the air.

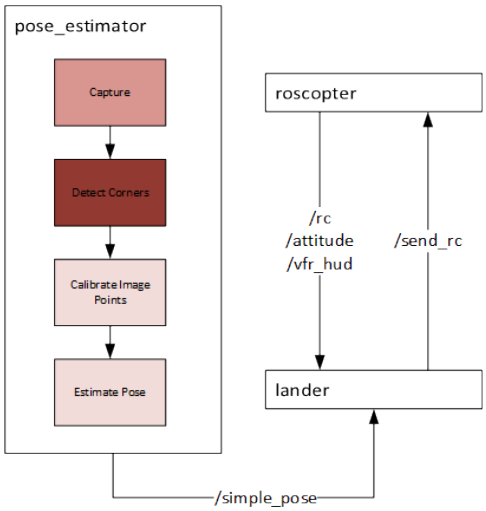

Figure 9

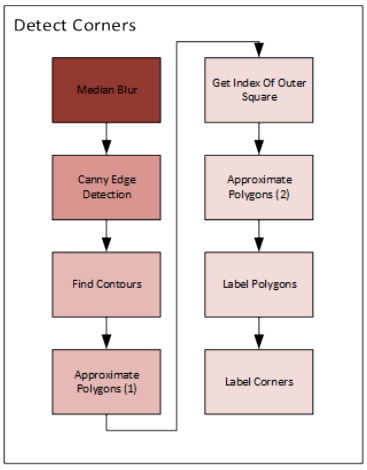

Figure 10

Compiler Optimisations gcc and other compilers offer a myriad of optional performance optimisations. For example, they offer user specified flags that cause the generated executable file to be optimised for a certain instruction set. When compiling libraries such as OpenCV for an embedded computer, it is typical to cross-compile; compilation on embedded computers typically takes many times longer. Cross-compilation is when compilation happens on a more powerful compilation computer that has available to it a compiler for the target architecture. In our case, compilation was on a quad-core x86 computer for an ARM target architecture. At this stage, it is possible to pass certain parameters to the compiler that permit it to use ARM NEON instructions in the generated binary code.

NEON is an "advanced SIMD instruction set" introduced in the ARMv7 architecture - the basis for all modern ARM processors. Single instruction, multiple data (SIMD) instructions are data parallel instructions that allow a single operation to be performed in parallel on two more operands. While the compiler has to be conservative in how it utilises these in the generated binary code so that correctness is guaranteed, these allow some performance gain.

Additionally, library providers may implement optional code paths that are enabled when explicitly compiling for an ARM target architecture. For instance, the OpenCV maintainers have several functions that have optional NEON instructions implemented. This also results in a performance boost.

Library Optimisations OpenCV utilises other libraries to perform fundamental functions such as image format parsing and multi-threading. For core functions, such as parsing of image formats, a standard library is included and used. For advanced functions, support is optional and so these are disabled by default.

We experimented with enabling multithreading functionality by compiling OpenCV with Intel's Thread Building Blocks library. This is a library that provides a multi-threading abstraction and for which support is available in select OpenCV functions [3].

Secondly, we re-compiled OpenCV replacing the default JPEG parsing library, libjpeg, with a performance optimised variant called libjpeg-turbo. This claims to be 2.1 to 5.3x faster than the standard library [8] and has ARM optimised code paths. Using this, it is possible to capture images at 30 frames per second on the BeagleBone Black [5].

Note also that we were unable to use the Intel Integrated Performance Primitives (IPP) library. IPP is the primary method of compiling a high performance version of OpenCV. However, it accomplishes this by utilising significant customisation for x86 processors that implement instruction sets only available on desktop computers (e.g. Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX)).

Disabling CPU Scaling Modern embedded computers typically use some sort of aggressive CPU scaling in order to minimise power consumption through an 'ondemand' governor [11]. This is beneficial for consumer applications where battery life is a significant concern and commercial feature.

However, for a real-time application such as this, CPU scaling is undesirable since it introduces a very slight latency as load increases and the CPU frequency is increased by the governor. This is more desirable still in architectures such as the big.LITTLE architecture used on the Odroid XU which actually switches automatically between a low-power optimised processor and a performance optimised processor as load increases.

This can be mitigated by manually setting the governor to the 'performance' setting. This effectively disables frequency scaling and forces the board to run at maximum clock speed.

Use of SIMD Instructions As mentioned previously, modern ARM processors implement the NEON SIMD instruction set. Compilers such as gcc make instruction set specific SIMD instructions available to the programmer through additional instructions called intrinsics. Intrinsics essentially wrap calls to the underlying SIMD instruction. Using intrinsics, it is possible to exploit data parallelism in our own code, essentially rewriting higher level, non-parallel, function calls with low level function calls that operate on multiple data simultaneously. This approach is laborious and is therefore used sparingly. It can, however, yield significant performance improvements [9].

Use of Multi-threading Multi-threading is a promising approach. Certain library functions may inherently exploit multi-threading and there is a slight benefit to a single-threaded process to having multiple cores (the operating system will balance processes across cores). However, our single-threaded imple-

mentation still gains little performance from having multiple cores available - many of which were not occupied with useful work. Our multi-threaded implementation separates out the **Capture** step from the latter **Detect Corners**, **Calibrate Image Points** and **Estimate Pose** steps as shown in figure 11. This is accomplished by creating a pool of worker threads (the exact quantity of threads is configurable - generally 1 for every available core). Each time an image is captured, it is dispatched to the next free thread in a round-robin fashion. Work is distributed evenly across threads and, since each thread finishes computation of images in a similar time, pose estimates are published to the *\simplePose* topic in order of image acquisition. This approach is essentially what is commonly known as **pipelining**, each thread can be considered a pipeline, allowing concurrent processing of an image while the master thread captures the next image. Figure 12 shows how frames are processed for a single threaded process above that for a pipelined multi-threaded process.

We use the POSIX thread (pthread) libraries to assist with thread creation and management.

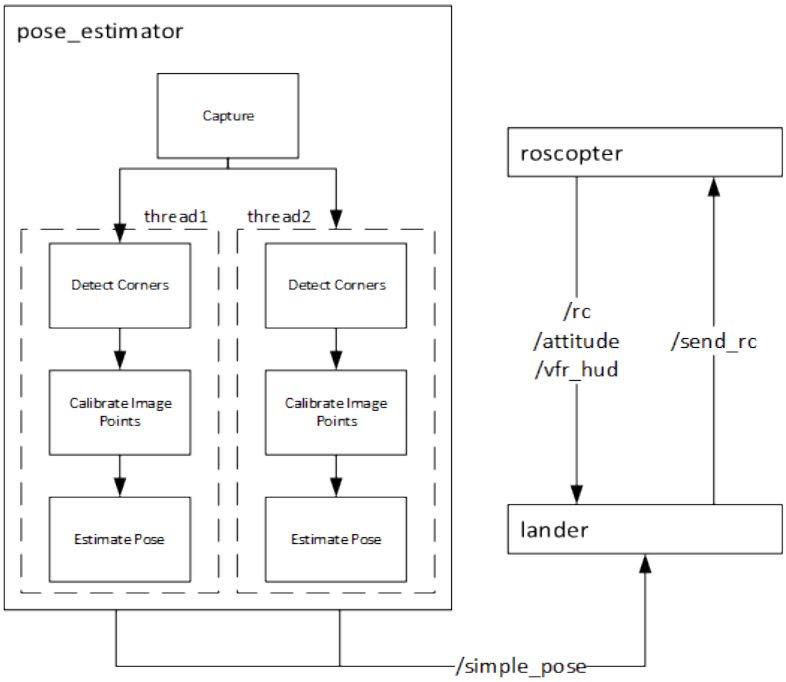

Figure 11

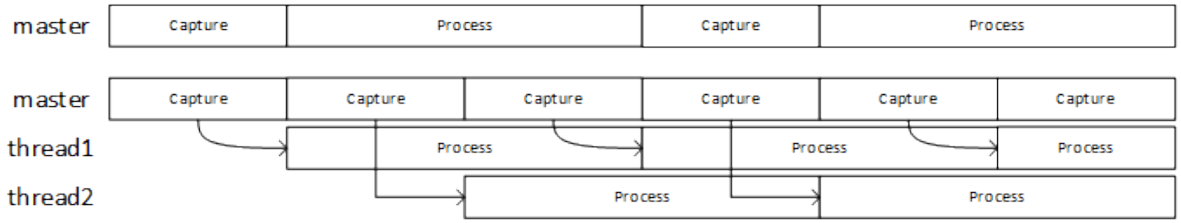

Figure 12

4 Discussion & Results

4.1 Accuracy

As mentioned previously, we experimentally determined the mean accuracy of GPS-based landing to be 195.33 cm away from the takeoff position. We verified the accuracy of our pose estimation implementation using a test rig in our lab. The camera was held above the centre of the landing pad and the true measured x, y and z were compared to those reported by the pose estimator.

Additionally, we also measured the standard deviation over multiple measurements at the same scene. Table 2 shows these results. There is a small (1.4% relative) systematic overestimation of the true height. This can be corrected by performing a more precise calibration.

The height estimate is very stable at these heights (small standard deviation). The x and y estimates are also fairly accurate, although the error in these estimates is an order of magnitude greater than the lower bound. This is consistent with the fact that there is some noise in the corner detection, and the pose estimator finds an approximate solution. A possible solution around excessive noise in the pose estimates would be to incorporate a Kalman filter.

true height	z mean	z std	x bound	x std	y bound	y std	yaw std
88 cm	89.3 cm	0.05 cm	0.19 cm	0.43 cm	0.14 cm	0.39 cm	0.12 °
120 cm	121.1 cm	0.08 cm	0.26 cm	1.16 cm	0.19 cm	1.06 cm	0.12 °
170 cm	172.0 cm	0.18 cm	0.37 cm	2.74 cm	0.27 cm	2.17 cm	0.07 °
226 cm	229.0 cm	0.54 cm	0.49 cm	6.51 cm	0.36 cm	6.05 cm	0.34 °

Table 2: Accuracy of our pose estimates. “x bound” and “y bound” are lower bounds for the error on x and y, given by the limited image resolution.

4.2 Other Challenges

4.2.1 Motion Blur

Despite choosing a high end web camera, the sensor was still sensitive to low-light situations. Under lower-light conditions, such as a cloudy day, the captured images would suffer from motion blur, as shown in figure 13. Blurry images would cause pose estimation to fail.

A potential way to mitigate this is to use a lit landing pattern along with a fixed exposure time. While we have built a landing pattern using red LEDs, the specific camera used did not support a programmatic way to manipulate the shutter speed and so this approach did not work.

4.2.2 Field Of View

We measured our camera's field of view to 69° on the long side and 42° on the short side. This means that at a height of one meter, we see an area of 1.37 m by 0.77 m on the ground. Our landing platform is about 0.76 m by 0.76 m. This means that at a height of one meter, the UAV has to be exactly above the landing pad in order to see all of it; pose estimates can be calculated only if the entire landing pad is visible. Since the UAV is rarely exactly above the landing platform, the limited field of view is also a problem at heights above one meter.

Figure 13: An image where pose estimation fails.

The controller should switch to the *LAND_LOW* state when below a height at which it was able to calculate pose estimates. However, this was suboptimal: 1) ground effects made the motion of the aircraft very unstable when hovering close to the ground and it would drift significantly as it lowered; and 2) the drift in altitude estimates provided by the barometric pressure sensor made it difficult to tell when to *POWER_OFF*.

In order to mitigate these issues, we modified our original code and built a smaller landing pad. While this requires the UAS to be closer (at a maximum height of 3 metres) to start receiving pose estimates, it means the camera can see the entire pad until it is 50 centimeters from the ground. (This is considerably better than the approximately 2 meter altitude where pose estimates would cut off for the larger pad.)

4.2.3 Control

While this project was not explicitly focussed on the control aspects, our control algorithm was necessary to demonstrate that this approach is technically viable. Unfortunately, windy conditions when testing combined with slow performance meant that the UAS was unable to maintain a steady position and instead drifted signficantly as gusts of wind presented themselves.

Extending our proportional controller to a full PID controller would provide more aggressive control and would perform better under these conditions. However, this approach would still rely on passing raw PWM values to the autopilot via roscopter.

The most optimal approach would be to extend the MAVLink protocol such that only error values are sent to the inner control loop of the autopilot. These error values would then be presented to the core PID loop maintaining stability and as such, it would reduce the need for an extra control loop and hence

make redundant the extra level of indirection and corresponding tuning.

4.3 Performance

A considerable amount of effort went into optimising performance of this vision system when run on an embedded ARM computer. The following section describes the methods used to benchmark various implementations, what the maximum performance attainable is and what results we gained.

4.3.1 Benchmarking Methodology

Environment The large fixed size landing pad was used a fixed distance of approximately 1.5 meters from the web camera. All tests were performed indoors in a room without windows and with a constant lighting level produced by a fluorescent tube light. No other user space processes were running on each computer aside from *roscore* and the pose estimator process itself.

Timing Data Each implementation was benchmarked using calls to C++'s *gettimeofday* function. For each implementation we profiled the amount of time taken for various calls within our pose estimation routine over 20 frames. For each test, we discarded the first 20 frame chunk since this is when the camera automatically adjusts exposure and focus settings. This data was collected for 10 arbitrary 20 frame chunks and averaged to provide overall figures.

4.3.2 Maximum Performance

The Logitech C920 web camera we are using captures frames at a 640 x 480 pixel resolution at a maximum of 30 frames per second. It would be impossible to operate any quicker than the camera is capable of delivering frames and therefore the upper bound for performance is 30 Hz.

Board	**BeagleBone**	**Odroid**
No computation	29.57	29.65
Basic decoding using OpenCV	18.69	24.60

Table 3: Baseline FPS for the BeagleBone and Odroid boards.

Using the framegrabber.c application provided in [5], it was possible to evaluate what the maximum single threaded performance could be, assuming basic decoding of image frame using OpenCV and no other computation. Table 3 shows the results of this benchmark. We see that the BeagleBone performs at 75% of the speed of the Odroid - which is intuitively correct, given the slower clock speed of the BeagleBone's processor and the fact that it has just a single core - some amount of its load will be used by operating system tasks that can be scheduled on other cores on the Odroid.

4.3.3 Performance of Naive Implementation

To begin with, we benchmarked our naive implementation with no optimisations on both boards and a desktop computer with a quad-core Intel Core i5 processor. This showed an obvious discrepancy in performance, with average FPS for each shown in table 4.

Board	Pad visible	
	no	yes
BeagleBone	2.94	3.01
Desktop	30.04	29.97
Odroid	8.80	8.93

Table 4: Naive implementation: average FPS for the BeagleBone, Odroid and a desktop computer.

The desktop computer has no issue running at the maximum 30 FPS but both the BeagleBone and Odroid fail to give the necessary 10 Hz required for real-time control (as described in section 3.3.4).

There is a slight but obvious discrepancy between performance when the pad is in view and when the pad is not which only appears to manifest itself on the two ARM boards (the BeagleBone and Odroid). By profiling the steps of detect corners, the first four of which are shown for the Odroid in table 5, we can begin to see why. Canny edge detection and finding contours takes very slightly longer on both boards when there is no pad in view. Intuitively this is because when the pad is in view, it occludes a significant part of the image. The pad is constructed of very simple quadrilaterals that are atypical of the many small heterogeneous shapes that a normal frame is composed of. For the remaining tests the figures represent performance when the pad is in view.

Pad	**Median Blur**	**Canny**	**Find Contours**	**Approximate Polygons (1)**
No	0.054	0.039	0.003	0.001
Yes	0.054	0.038	0.002	0.001

Table 5: Naive implementation: breakdown of first four steps of *Detect Corners* with and without a frame in view for Odroid (similar results for BeagleBone).

Figure 14 show a breakdown of the naive implementation of the overall algorithm on all three of our boards. It is clear that the majority of time is spent in the *Capture* and *Detect Corners* stages. Video capture is handled by an OpenCV function so we are unable to easily profile that. However, figure 15 shows a breakdown of time spent in the *Detect Corners* subroutine. Again, here, it is clear that two calls to *Median Blur* and *Canny* contribute the majority of processing time.

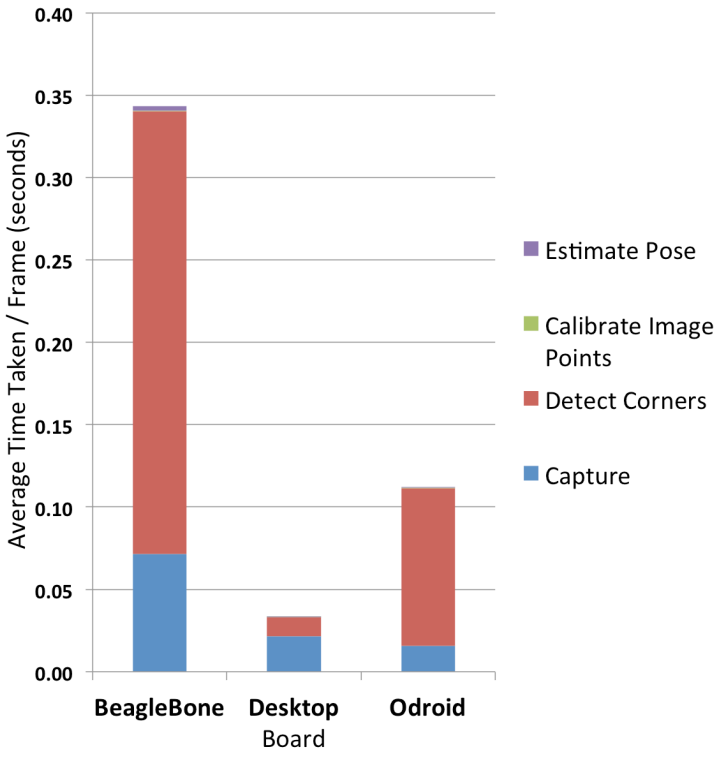

Figure 14: Naive implementation: breakdown of overall algorithm

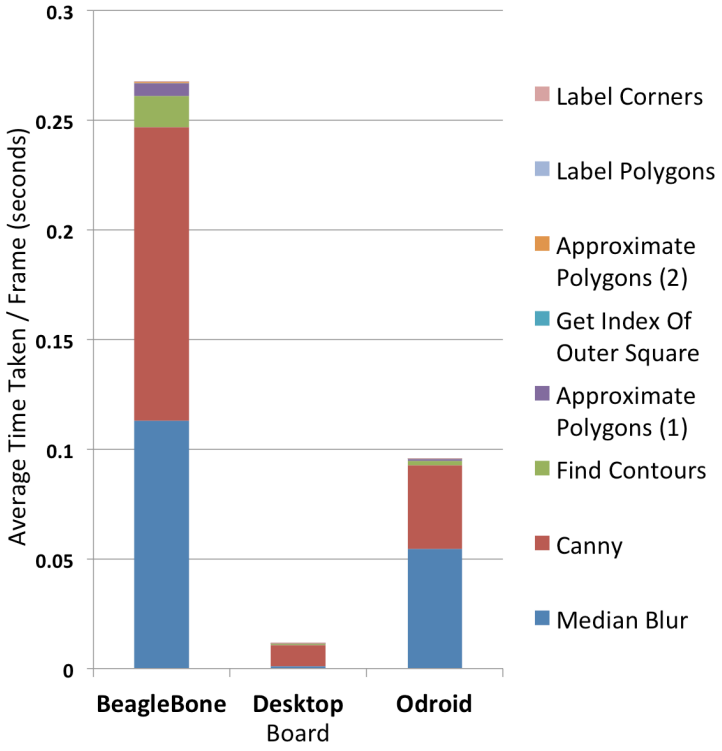

Figure 15: Naive implementation: breakdown of *Detect Corners*

4.3.4 Performance of Naive Implementation with Optimised Libraries

The next stage of optimisation was to adapt the libraries used onboard. OpenCV was the primary focus and we did three things, as described in 3.3.4:

1. Cross compile with NEON code generation enabled
2. Cross compile with NEON code generation and Thread Building Blocks (TBB) support enabled
3. Cross compile with NEON code generation, TBB support and libjpeg-turbo support enabled

Table 6 shows the improvement in average FPS for each adaptation. We get overall a 10% gain in performance over the standard library. Note that the addition of Thread Building Blocks does very little for performance because the OpenCV functions used were not optimised to use it.

Board	Standard	NEON	NEON+TBB	NEON+TBB+libjpeg-turbo
BeagleBone	2.91	2.84	2.98	3.20
Odroid	8.93	9.60	9.60	9.90

Table 6: Naive implementation with optimised libraries: average FPS for the BeagleBone and Odroid with various library optimisations.

Figure 16 shows how the time taken for steps in the overall algorithm changes with each adaptation. Compiling for the NEON architecture causes consistent gains across each procedure while using libjpeg-turbo specifically optimises the *Capture* step. This can be explained by the fact that the *Capture* is frame data encoded in Motion-JPEG from the web camera into the **Mat** object OpenCV uses internally.

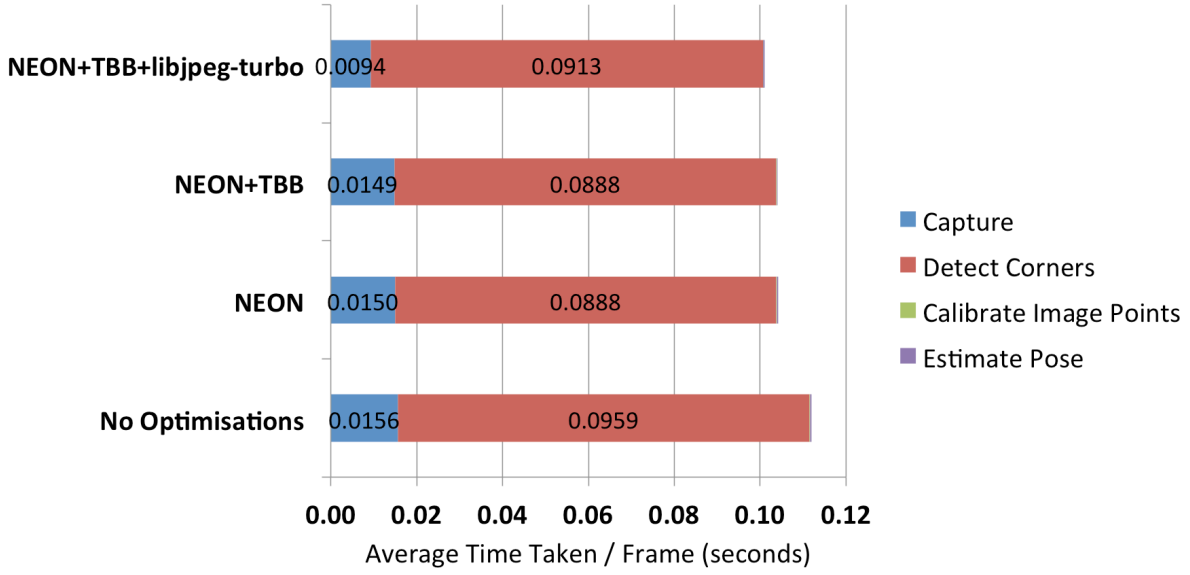

Figure 16: Naive implementation with optimised libraries: breakdown of overall algorithm for Odroid (similar results for BeagleBone).

4.3.5 Performance of Single-threaded Optimised Implementation

Our first optimisation to the code itself was to remove redundant calls and to minimise computation whereever possible. It turns out that the *Median Blur* step within the *Detect Corners* subroutine was unnecessary. Furthermore, some of our loops used to detect polygons would operate on all contours detected and not just those of interest. This led to extra unnecessary computation.

Figure 17 shows how the time taken by each stage of *Detect Corners* before and after this change.

Table 7 shows the performance improvement of approximately 60% when run on the BeagleBone and 120% when run on the Odroid.

Board	Optimised Libraries	Optimised Single Threaded
BeagleBone	3.20	5.08
Odroid	9.90	21.58

Table 7: Single-thread optimised implementation: average FPS for the BeagleBone and Odroid

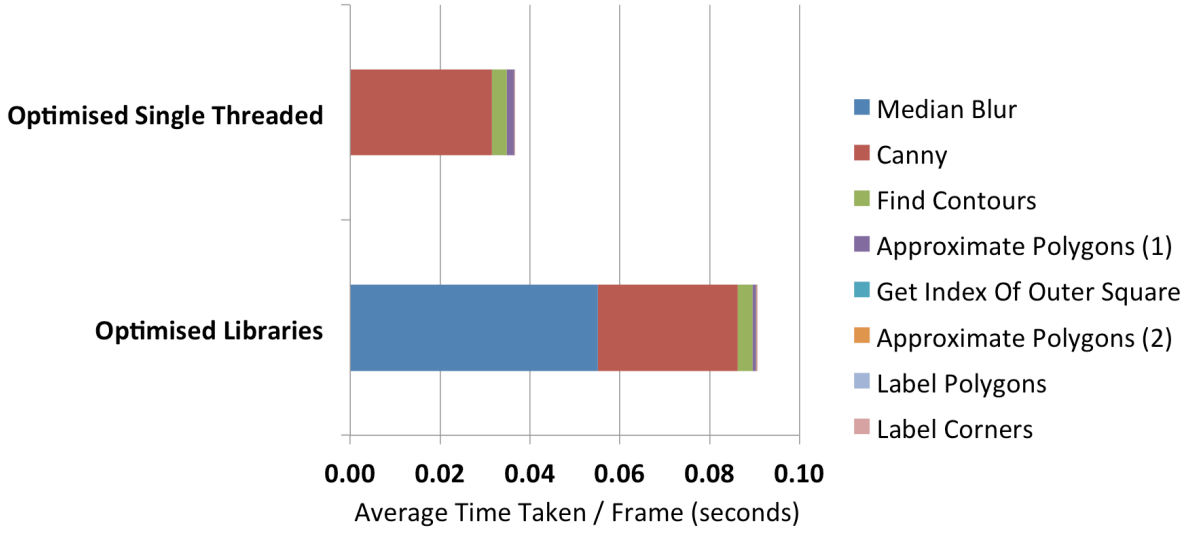

Figure 17: Single-thread optimised implementation: breakdown of *Detect Corners* for Odroid (similar results for BeagleBone).

4.3.6 Performance of Multi-threaded Optimised Implementation

The next optimisation implemented was pipelining by using multiple threads. Profiling data was collected for 1, 2, 4 and 8 threads and compared to the baseline single-threaded optimised version.

Figures 18 and 19 shows this for the BeagleBone and Odroid respectively. Performance suffers significantly on the BeagleBone, a single core board, as context switching causes extra overhead. This context switching is visible when using a single thread on the Odroid (because of context switching between the master thread and the single worker thread) but the performance benefit becomes clear as additional threads are introduced.

2 threads appears sufficient to get us to almost 30 frames per second - with the occasional frame not getting processed due to fully loaded worker threads. With 3 or more worker threads, no frame gets lost.

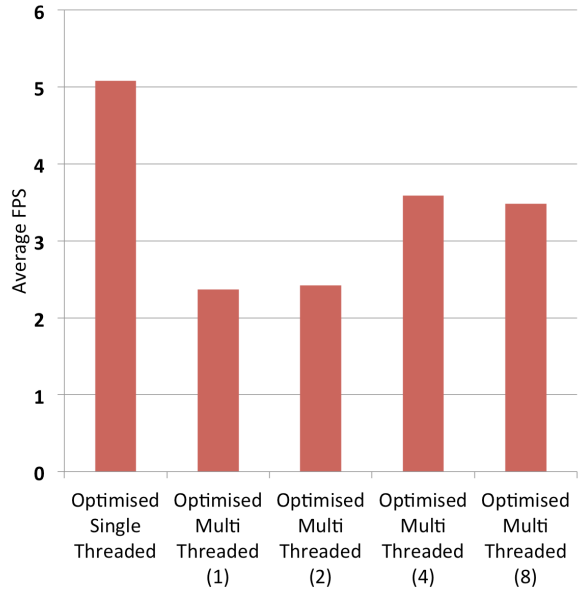

Figure 18: Multi-thread optimised implementation: average FPS for BeagleBone for different numbers of threads

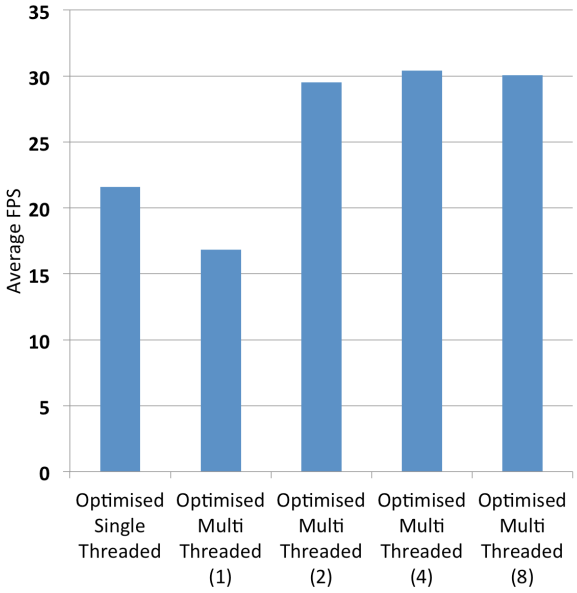

Figure 19: Multi-thread optimised implementation: average FPS for Odroid for different numbers of threads

Table 8 shows the 1 minute average system load (also known as the number of waiting processes) reported by Linux as additional threads are added when running on the Odroid. This does not increase significantly beyond 2 threads, validating the earlier point that 2 threads is sufficient to carry out nearly all of the computation.

Threads	System Load
1	0.59
2	1.10
4	1.17
8	1.20

Table 8: Multi-thread optimised implementation: 1 minute average system load for Odroid for different numbers of threads

4.3.7 Performance of Single-threaded Optimised Implementation (2)

When it became clear that our initial single-threaded optimised implementation was not working quickly enough on the BeagleBone and when multi-threading failed to work, we re-visited the single-threaded approach and looked for alternative approaches to computation.

By replacing *Canny Edge Detection* by adaptive thresholding, we were able to improve the performance of our single-threaded implementation further to almost 6 FPS on the BeagleBone and to the maximum 30 FPS on the Odroid. The pose estimation results remain correct.

Table 9 shows the improvement in FPS over our original single-threaded optimised implementation.

Board	Optimised Single Threaded	Optimised Single Threaded (2)
BeagleBone	5.08	5.97
Odroid	21.58	30.19

Table 9: Single-thread optimised implementation (2): average FPS for the BeagleBone and Odroid

Figure 20 shows that the majority of the speed increase came from the reduction in time taken by the *Canny Edge Detection* step. There was also a corresponding time saving during the *Find Contours* step - presumably because thresholding is actually a more selective preprocessing method than Canny edge detection.

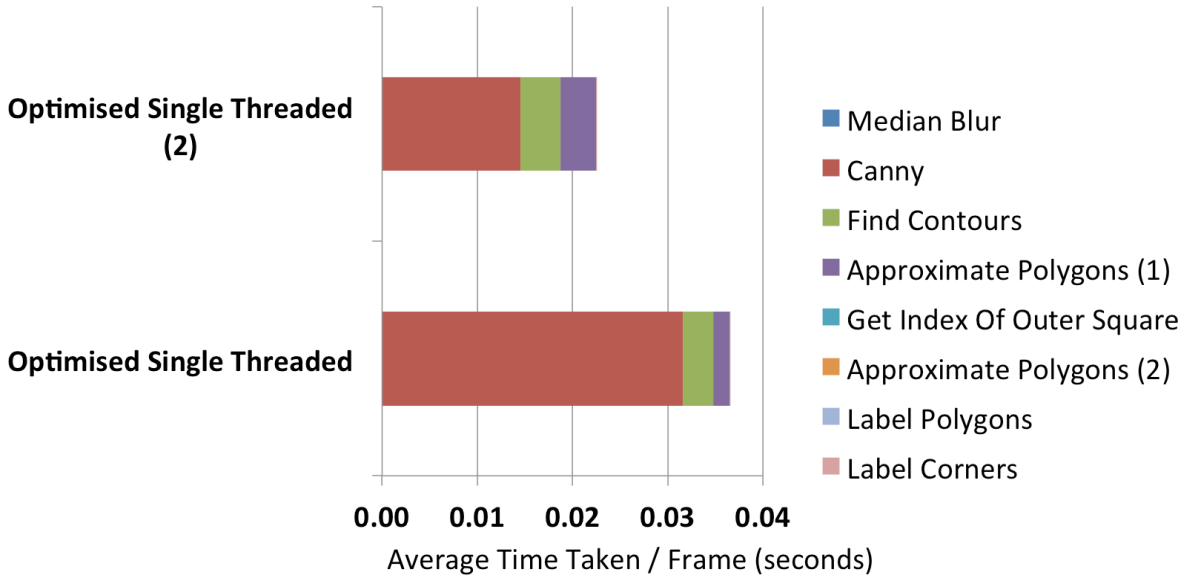

Figure 20: Single-thread optimised implementation 2: breakdown of *Detect Corners* for Odroid (similar results for BeagleBone).

4.3.8 Performance Summary

Figures 21 and 22 show the increase in average FPS for each of the implementations as various optimisations were added. Our best performance was just under 6 FPS for the BeagleBone and at the full 30 FPS for the Odroid. Figure 23 shows the difference in processing time for each frame in *Detect Corners* from the naive implementation to the optimised single threaded version. Notice that our optimisations made a proportionally greater difference on the Odroid than the BeagleBone.

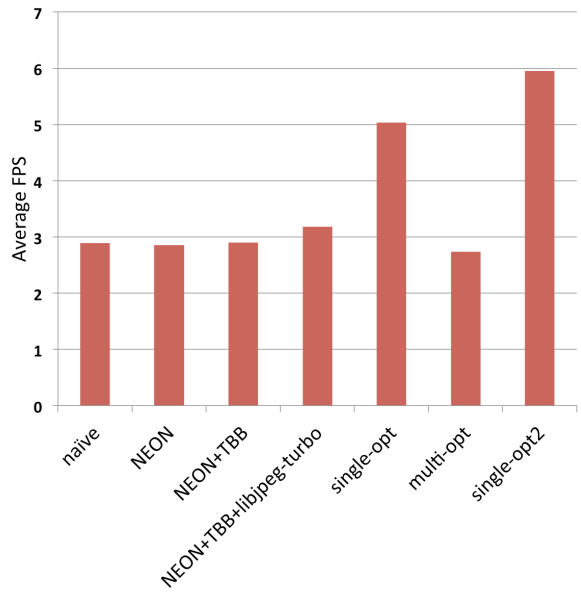

Figure 21: All implementations: average FPS for BeagleBone

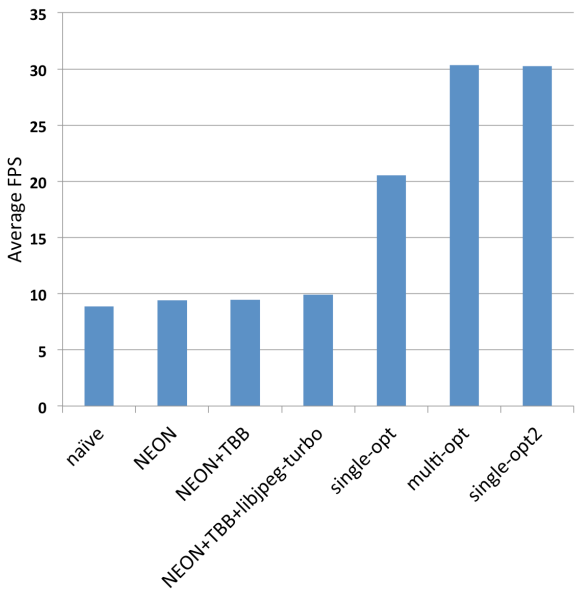

Figure 22: All implementations: average FPS for Odroid

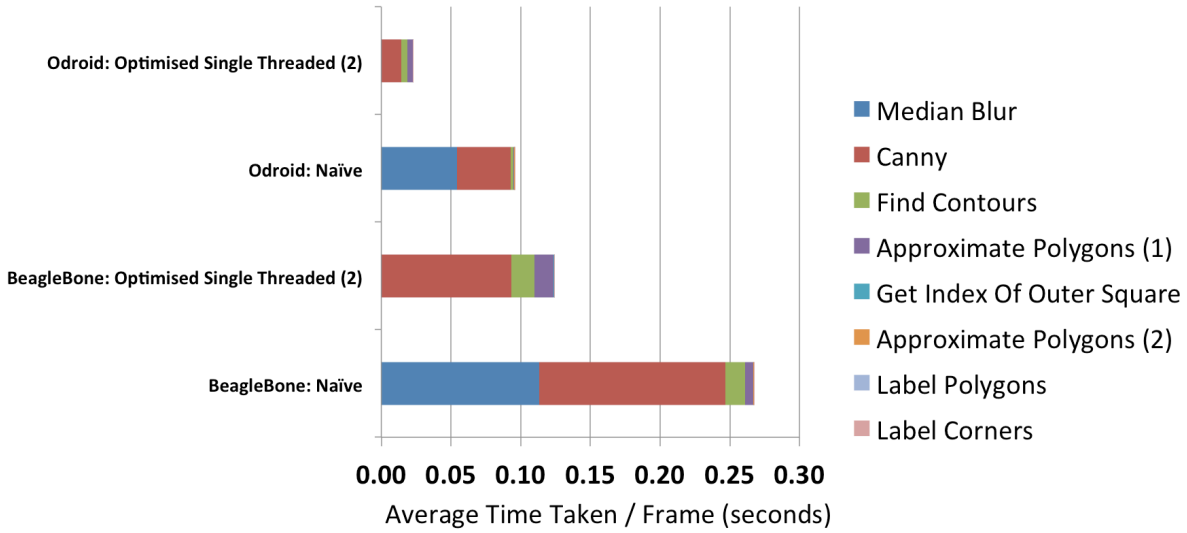

Figure 23: Naive to optimised implementation: breakdown of *Detect Corners* for BeagleBone and Odroid.

5 Conclusions & Future Work

In 2011, Sharp et al [18] reported results of 30 FPS while running this pose estimation algorithm. They used highly optimised custom C code running on a much slower board and took significantly more time to implement their algorithm. They made several optimisations which we have yet to explore, involving running computationally intense steps of their pose estimation algorithm on lower resolution images to provide approximate solutions for the larger resolution image.

While we were unable to achieve this result on the lower cost BeagleBone, we were able to get performance that may be sufficient for some basic applications in situations which are less dynamic and do not necessarily require an update rate of 10Hz. Using the Odroid we were able to prototype and implement their approach at the maximum possible frame rate with plenty of system capacity to spare.

This suggests that it is now feasible for advanced robotics applications to run onboard using a multi-core embedded computer. Additionally, our development time was significantly quicker - we were able to implement their approach in months as opposed to years. This is a promising result.

Thus a hybrid approach is suggested. One can prototype a new application quickly using open source libraries but some manual optimisation is required to gain peak performance. Since multi-core systems are quickly becoming the norm, implementing multi-threading should provide significant speedups for most vision processing tasks.

6 Acknowledgements

Without the generous support of 3DRobotics and advice from Brandon Basso, this project would not have been possible. Additionally, several other graduate students collaborated on and contributed extensively to this project: Constantin Berzan, Nahush Bhanage, Gita Dombrowski and Hoang Nguyen.

References

[1] 3DRobotics. *3DR Pixhawk*. [Online]. 2014. URL: https://store.3drobotics.com/products/3dr-pixhawk.

[2] 3DRobotics. *APM 2.6 Set*. [Online]. 2013. URL: http://store.3drobotics.com/products/apm-2-6-kit-1.

[3] OpenCV Adventure. *Parallelizing Loops with Intel Thread Building Blocks*. [Online]. 2011. URL: http://experienceopencv.blogspot.com/2011/07/parallelizing-loops-with-intel-thread.html.

[4] AUVSI. *The Economic Impact of Unmanned Aircraft Systems Integration in the United States*. [Online]. 2013. URL: http://www.auvsi.org/econreport.

[5] Michael Darling. *How to Achieve 30 fps with BeagleBone Black, OpenCV, and Logitech C920 Webcam*. [Online]. 2013. URL: http://blog.lemoneerlabs.com/3rdParty/Darling_BBB_30fps_DRAFT.html.

[6] Pedro J Garcia-Pardo, Gaurav S Sukhatme, and James F Montgomery. "Towards vision-based safe landing for an autonomous helicopter". In: *Robotics and Autonomous Systems* 38.1 (2002), pp. 19–29.

[7] Stanley R Herwitz et al. "Precision agriculture as a commercial application for solar-powered unmanned aerial vehicles". In: *AIAA 1st Technical Conference and Workshop on Unmanned Aerospace Vehicles*. 2002.

[8] libjpeg-turbo. *Performance*. [Online]. 2013. URL: http://www.libjpeg-turbo.org/About/Performance.

[9] Gaurav Mitra et al. "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 1107–1116.

[10] Stack Overflow. *What is the best library for computer vision in C/C++?* [Online]. 2009. URL: http://stackoverflow.com/questions/66722/what-is-the-best-library-for-computer-vision-in-c-c.

[11] Venkatesh Pallipadi and Alexey Starikovskiy. "The ondemand governor". In: *Proceedings of the Linux Symposium*. Vol. 2. sn. 2006, pp. 215–230.

[12] E. Pereira, R. Sengupta, and K. Hedrick. "The C3UV Testbed for Collaborative Control and Information Acquisition Using UAVs". In: *American Control Conference*. AACC. 2013.

[13] Kari Pulli et al. “Real-time computer vision with OpenCV”. In: *Communications of the ACM* 55.6 (2012), pp. 61–69.

[14] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. 2009.

[15] John L. Rep. Mica et al. “FAA Modernization and Reform Act of 2012”. In: (2012).

[16] Katie Roberts-Hoffman and Pawankumar Hegde. “ARM cortex-a8 vs. intel atom: Architectural and benchmark comparisons”. In: *Dallas: University of Texas at Dallas* (2009).

[17] Srikanth Saripalli, James F Montgomery, and Gaurav S Sukhatme. “Vision-based autonomous landing of an unmanned aerial vehicle”. In: *IEEE International Conference on Robotics and Automation*. Vol. 3. IEEE. 2002, pp. 2799–2804.

[18] Courtney S. Sharp, Omid Shakernia, and Shankar Sastry. “A Vision System for Landing an Unmanned Aerial Vehicle.” In: *IEEE International Conference on Robotics and Automation*. IEEE, 2001, pp. 1720–1727.

[19] Eric Stotzer et al. “OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip”. In: *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 114–127.

[20] Sebastian Thrun et al. “Stanley: The robot that won the DARPA Grand Challenge”. In: *Journal of field Robotics* 23.9 (2006), pp. 661–692.

[21] Chunhua Zhang and John M Kovacs. “The application of small unmanned aerial systems for precision agriculture: a review”. In: *Precision agriculture* 13.6 (2012), pp. 693–712.