Parallelised Pose Estimation for Automated Landing

CS267: Project Report

Sunil Shah
sunil.shah@berkeley.edu

Nahush Bhanage
nahush@berkeley.edu

Hoang Nguyen
hoanghw@berkeley.edu

ABSTRACT

Current approaches for automated landing of unmanned aerial systems (UAS) are based on GPS localization, which we show is quite inaccurate. We optimise a previously implemented computer vision based pose estimation algorithm to run in real time on a low cost open source embedded computer using open source software.

1. INTRODUCTION

The rise of the hobbyist unmanned aerial system movement has been driven by three factors: 1) the increasing availability and popularity of open source hardware and software, 2) the maker movement and 3) the availability of low cost sensors. These have led to communities such as DIY-Drones, a message board where users developed the open source ArduPilot autopilot, initially based on the Arduino platform. In combination with such an autopilot, a user could feasibly 3D print (or purchase) their own multi-rotor frame, mount motors and wire to the autopilot radio control equipment used for ordinary radio control aircraft. For a total cost of approximately $1,000, a user would have a fully autonomous aerial system.

As the hobbyist UAS become more capable, an increasing number of researchers and startups are building products and systems using them. Based on this trend, a federal mandate set in place in 2012 a requirement for the FAA to integrate unmanned aerial systems (UAS) into the national airspace by 2015 for civilian and commercial use.

General purpose open source hardware boards such as the Raspberry Pi or BeagleBone Black typically use the ARM processor architecture, a reduced instruction set computing (RISC) design. This is the architecture most commonly used in smartphones and allows users to run full Linux based operating system such as Ubuntu or Android onboard, with use of with standard libraries and utilities. (This is unlikely prototyping boards such as the Arduino which have their own special purpose environment which is user friendly but limited.)

While increasing in computing power rapidly, RISC architectures lack the raw performance of more mature and advanced x86 processors which have the same or higher clock speeds but considerably more provisions for fast floating point operations, larger caches and increased processor level parallelism. However, the power, thermal efficiency and spatial gains from using a RISC architecture have made them compelling for power and payload constrained UAS.

In this project, we take a vision based pose estimation algorithm developed for a separate project and investigate parallel processing techniques to optimise its performance on a general purpose open source computer. Current methods of localisation for UAS rely upon GPS, which, unless augmented by additional ground stations transmitting from a known reference location, is inaccurate. In a test run of 10 automated landings where the UAS took off and attempted to land in the same location using GPS, we gained a mean accuracy of 195.33 cm [3].

Using a landing pad with a known design coupled with a webcam and computer on-board, it is possible to compute considerably accurate pose estimates using computer vision techniques [3]. These permit the UAS to localise itself much more precisely than using GPS alone which enables precision hovering and landing. However, these computer vision techniques are computationally intensive, relying upon several passes over an image to extract features. In our earlier work, we were unable to get this operating at more than 3 frames a second on the BeagleBone Black embedded computer.

The inner control loop of the ArduPilot autopilot software runs at 10 Hz - where it receives sensor data and outputs a control signal. This is deemed adequate for precise control of a UAS in all but the most extreme environmental conditions. Therefore, for pose estimation to be useful for UAS control, it is necessary to generate pose estimates at a rate of 10 Hz (or, 10 frames per second). Given that, in reality, certain frames are dropped due to motion blur, we strive for a rate of greater than 10 Hz.

We first survey prior work in section **??**, then describe our system architecture in section **??**. In section **??** we detail our approach to optimisation and in section **??** we show the results of this approach.

2. PRIOR WORK

Our naive implementation was the result of a previous project and a full description of prior work related to automated landing approaches are available in that paper [3]. The algorithm implemented is described by Sharp et al. in [11] and the overall approach is the same, although the exact implementation details may vary slightly. In their implementation, using highly optimised custom C code, they were able to reach a frame rate of 30 frames per second.

We consider other prior work related to high performance embedded computing. Several efforts have bene made to explore the effect of parallelising certain robotics appliations but these assume use of a desktop computer and often make use of general purpose computing on the GPU frameworks like CUDA or OpenCL. 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 [6]. 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 [8].

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. SYSTEM DESCRIPTION

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

Our overall system architecture is shown in figure 3.

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 [1]. 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 [10].

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, multicore HardKernel Odroid XU. The specification for each of the boards we used is outlined in table 1.

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 [12].

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 [9]. 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 1 shows how our code was modularised.

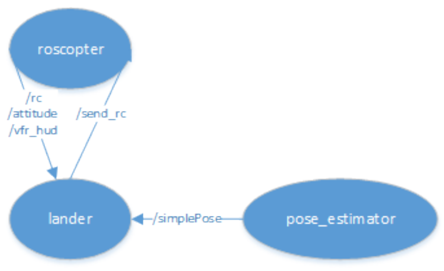

Figure 1: ROS nodes and topics for exchanging messages.

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)
Ports	USB 2.0 client USB 2.0 host Ethernet HDMI 2x 46 pin headers	USB 3.0 host x 1 USB 3.0 OTG x 1 USB 2.0 host x 4 HDMI Ethernet
Cost	$45	$169

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 [7].

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.

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. [11]. 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. Figure 4 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 5.

Corner Detection. As a first step, we detect the corners of the landing platform in an image, shown visually in figure 6:

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

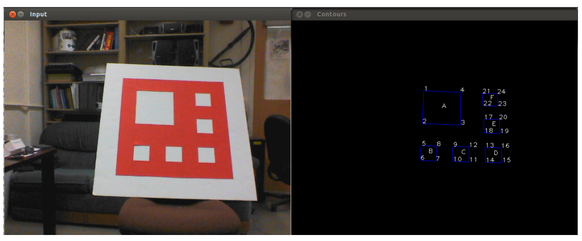

Figure 4: Left: Design of our landing platform. Right: Output of the corner detector (24 points, in order).

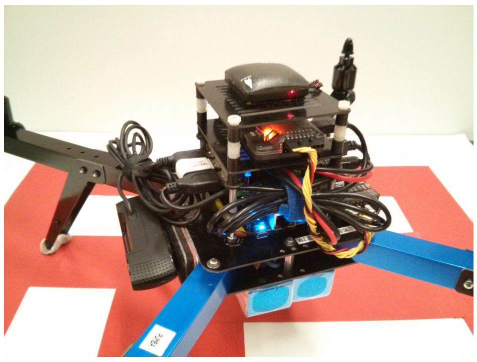
Figure 2: Our hardware stack fully assembled. Total weight excluding batteries is 1.35 kg.

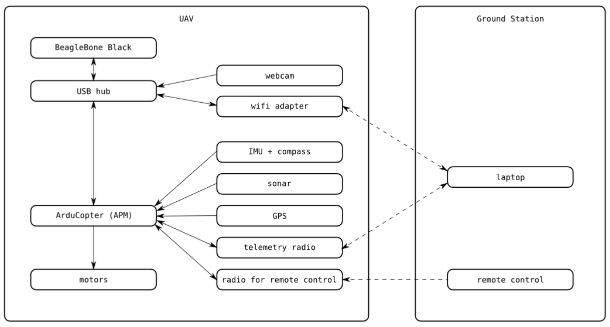

Figure 3: Architecture of our automated landing system. We use inexpensive off-the-shelf hardware. All the computation is performed onboard the UAS.

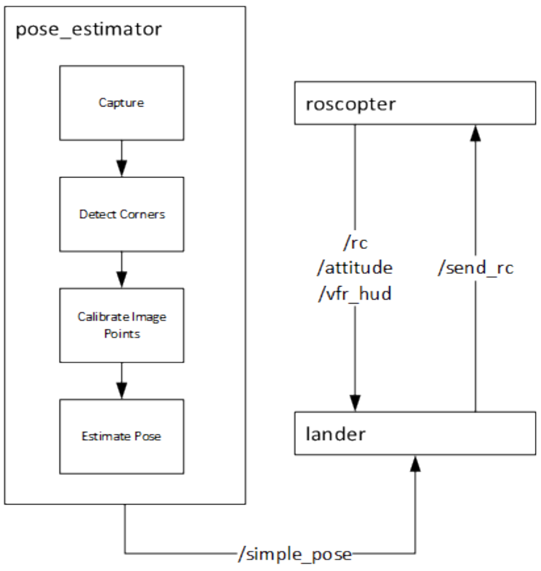

Figure 5: Flowchart of overall algorithm.

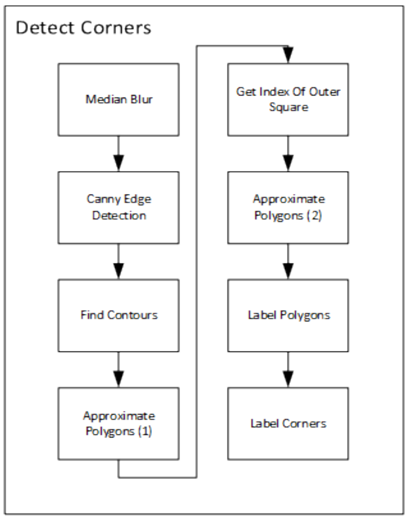

Figure 6: Flowchart of *Detect Corners*.

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. [11], whose details are omitted here for brevity. We use SVD to approximately solve a linear system of 48 equations with 6 degrees of freedom.

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. The state machine is more fully described in [3].

4. APPROACH

The naive implementation we began with operated at less than 3 frames per second on the BeagleBone Black. As mentioned previously, this is too slow for real-time control. This section describes the optimisations we made to our implementation to improve performance.

4.1 Profiling

We began by systematically profiling our code to identify hotspots where a disproportionate amount of time was being spent. A full outline of our benchmarking technique is described in section 5.1. At each stage of optimisation, we ran the pose estimation implementation through the same set of tests. Figure 7 shows hotspots inherent in our overall process and figure 8 shows hotspots within the "Detect Corners" subroutine.

4.2 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.

4.3 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 [2].

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 [5] and has ARM optimised code paths. Using this, it is possible to capture images at 30 frames per second on the BeagleBone Black [4].

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 avaiable on desktop computers (e.g. Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX)).

4.4 Single-threaded Optimisations

We considered two approaches to optimise our single-threaded implementation. Since the BeagleBone Black uses a single core processor, multi-threading is unlikely to have resulted in significant performance gains and so it was important to optimise the single-threaded version as much as possible to extract better performance from this board.

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. We describe the nature of these changes in the next section.

A second optimisation was to try and increase processor level parallelism (as opposed to the operating system parallelism that threading would provide). 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 [6]. Our initial attempt at rewriting computer vision functions using intrinsics were actually slower and time constraints prevented us from investigating this. Therefore, results from our own SIMD implementation are omitted.

4.5 Multi-threaded Optimisations

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 implementation 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 9. 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 $\backslash simplePose$ topic in order of image acquisition. This approach is essentially **pipelining**, each thread can be considered a pipeline, allowing concurrent processing of an image while the master

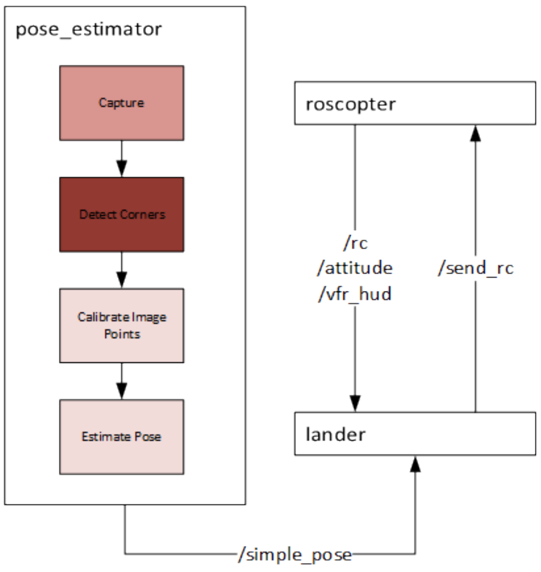

Figure 7: Flowchart of overall algorithm with hotspots highlighted.

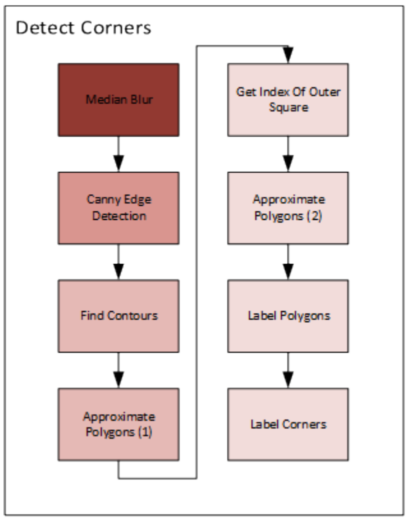

Figure 8: Flowchart of *Detect Corners* with hotspots highlighted.

thread captures the next image. Figure 10 shows how frames are processed for a single threaded process above that for a pipelined multi-threaded process.

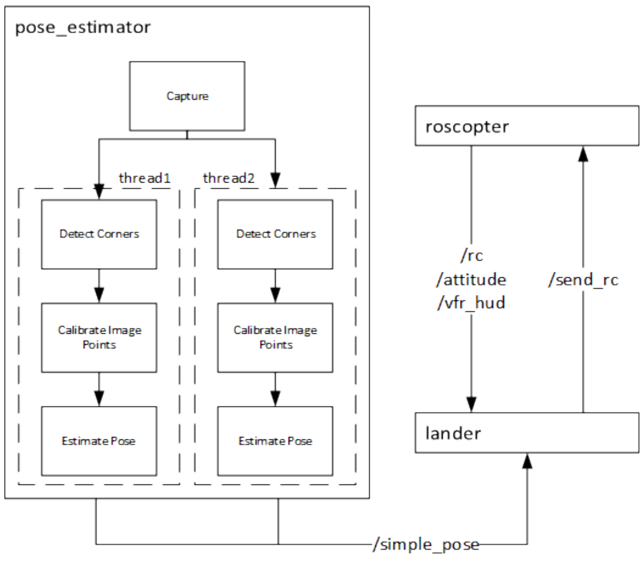

Figure 9: Refactored ROS nodes to allow multi-threading.

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

5. RESULTS

5.1 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.

5.2 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 2: Baseline FPS for the BeagleBone and Odroid boards.

Using the framegrabber.c application provided in [4], 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 2 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

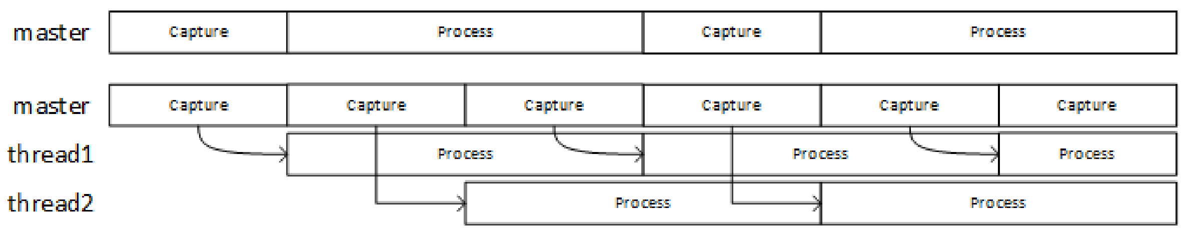

Figure 10

- some amount of its load will be used by operating system tasks that can be scheduled on other cores on the Odroid.

5.3 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 3.

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

Table 3: 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 **??**).

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 4, 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.

Figure 11 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 12 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.

5.4 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 **??**:

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 5 shows the improvement in average FPS for each adapation. 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.

Figure 13 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.

5.5 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 14 shows how the time taken by each stage of *Detect Corners* before and after this change.

Table 6 shows the performance improvement of approximately 60% when run on the BeagleBone and 120% when run on the Odroid.

5.6 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 15 and 16 shows this for the BeagleBone and Odroid

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

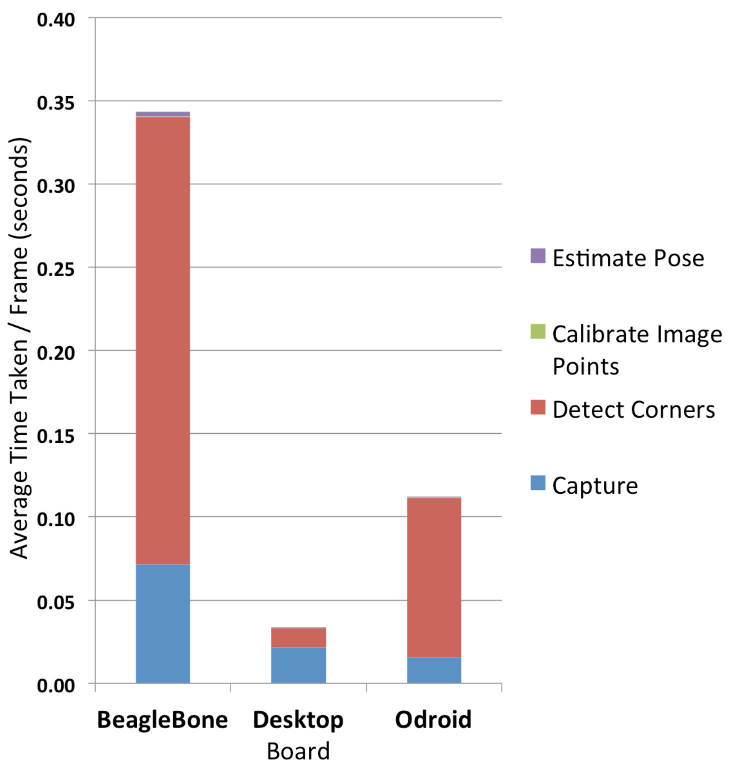

Figure 11: Naive implementation:
breakdown of overall algorithm

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

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

Table 6: Single-thread optimised implementation: average FPS for the BeagleBone and Odroid

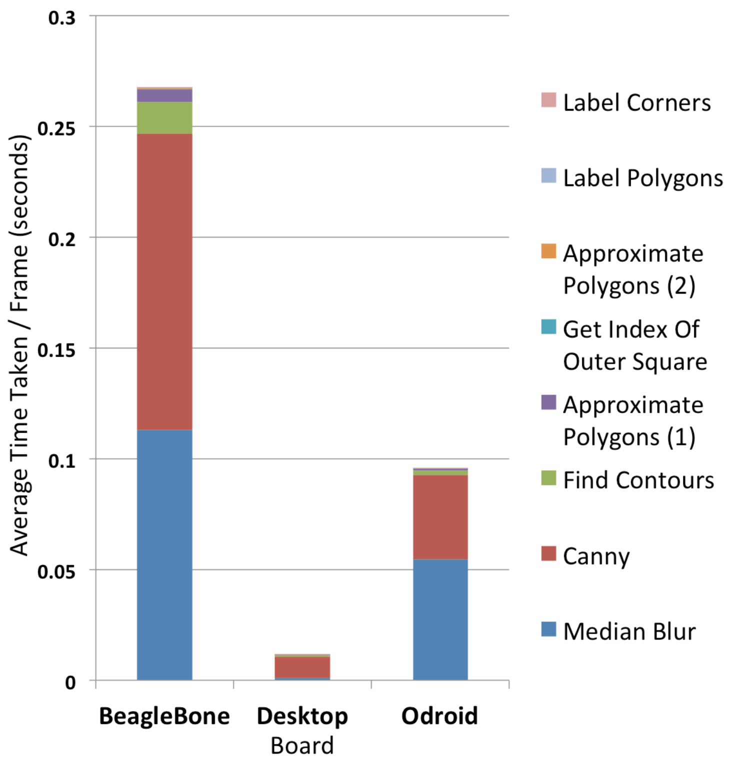

Figure 12: Naive implementation: breakdown of *Detect Corners*

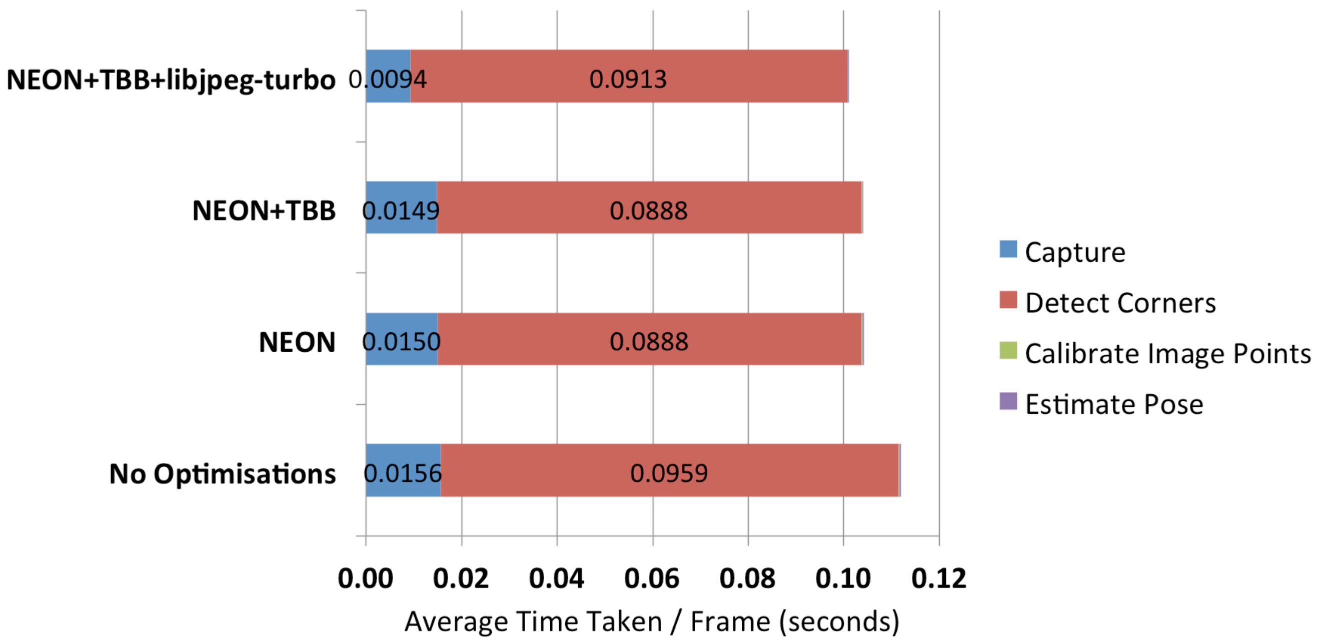

Figure 13: Naive implementation with optimised libraries: breakdown of overall algorithm for Odroid (similar results for BeagleBone).

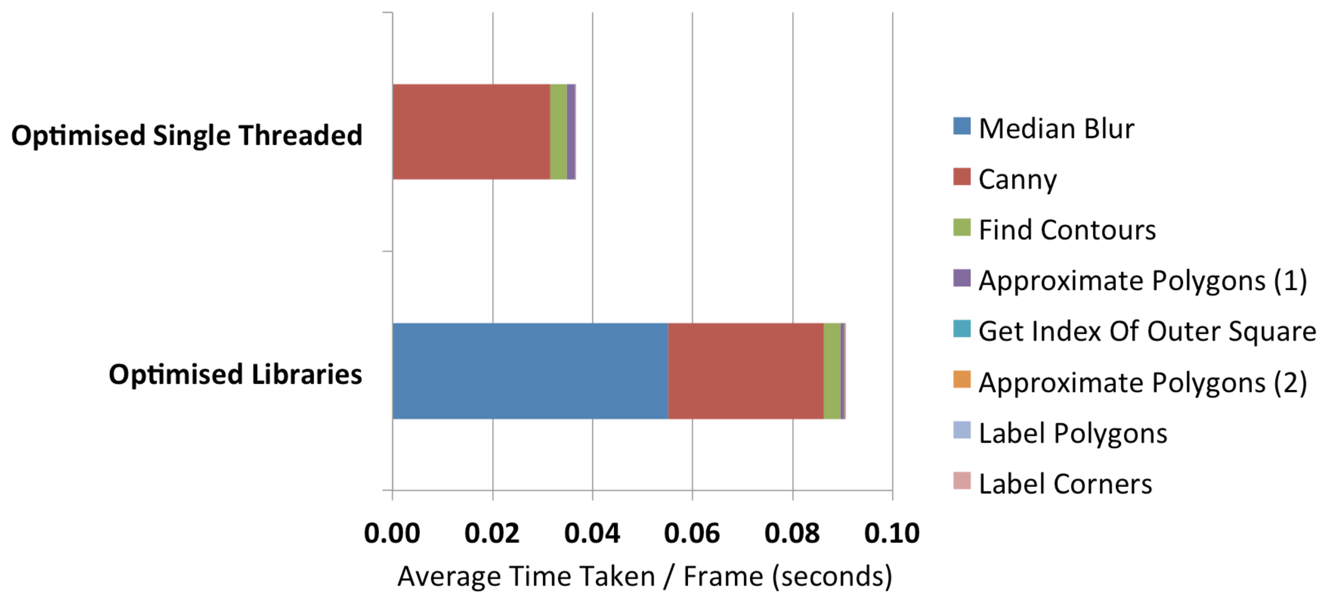

Figure 14: Single-thread optimised implementation: breakdown of *Detect Corners* for Odroid (similar results for BeagleBone).

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.

Table 7 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 7: Multi-thread optimised implementation: 1 minute average system load for Odroid for different numbers of threads

5.7 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 8 shows the improvement in FPS over our original single-threaded optimised implementation.

Figure 17 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.

5.8 Performance Summary

Figures 18 and 19 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 20 shows the difference in processing time for each frame in *Detect Corners* frmo the naive implementation to the optimised single threaded version. Notice that our optimisations made a proportionally greater difference on the Odroid than the BeagleBone.

6. CONCLUSION

In 2011, Sharp et al [11] 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

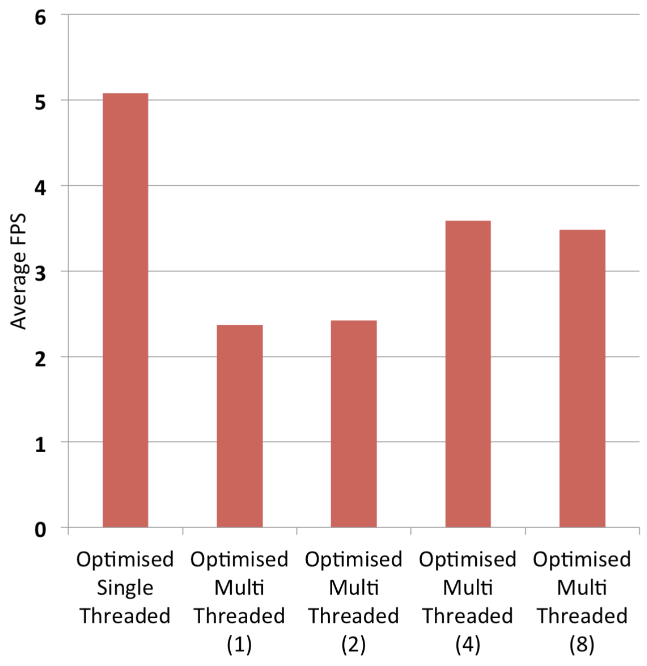

Figure 15: Multi-thread optimised implementation: average FPS for BeagleBone for different numbers of threads

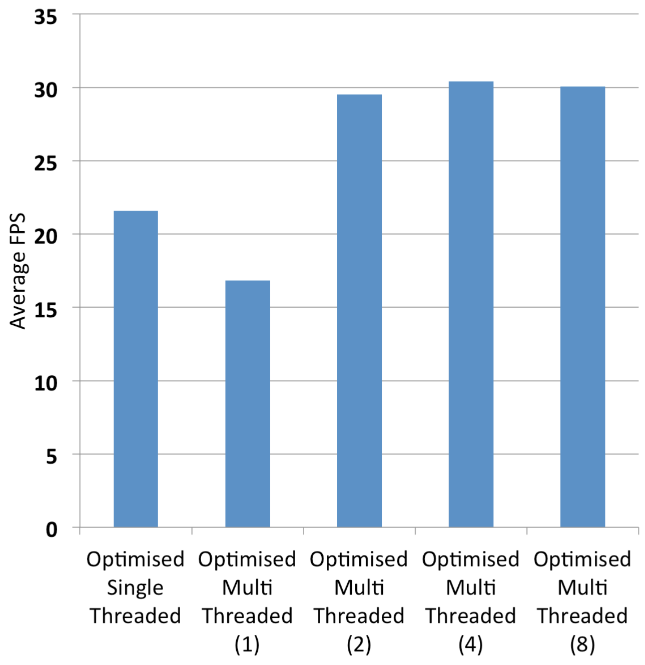

Figure 16: Multi-thread optimised implementation: average FPS for Odroid for different numbers of threads

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

Table 8: Single-thread optimised implementation (2): average FPS for the BeagleBone and Odroid

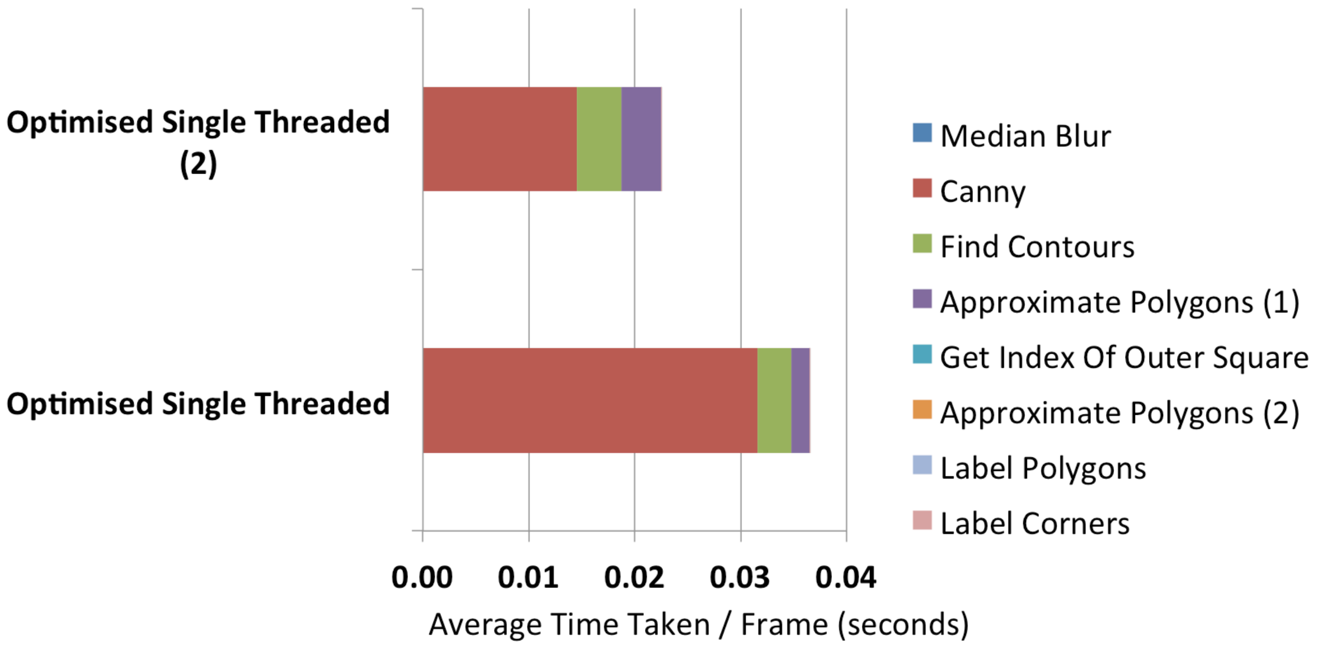

Figure 17: Single-thread optimised implementation 2: breakdown of *Detect Corners* for Odroid (similar results for BeagleBone).

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.

References

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

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

[3] Constantin Berzan, Nahush Bhanage, and Sunil Shah. "Accurate Vision-Based Landing For Multicopter UAVs". In: ().

[4] 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`.

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

[6] 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.

[7] 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`.

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

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

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

[11] 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.

[12] 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.

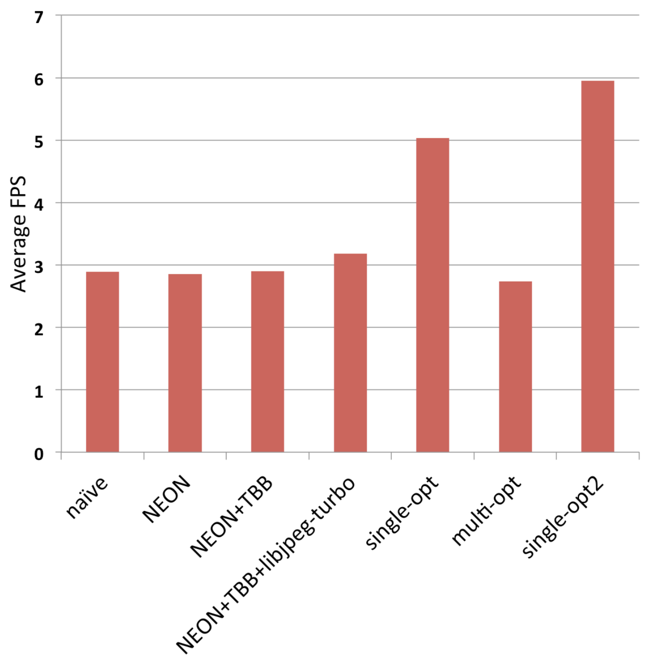

Figure 18: All implementations:
average FPS for BeagleBone

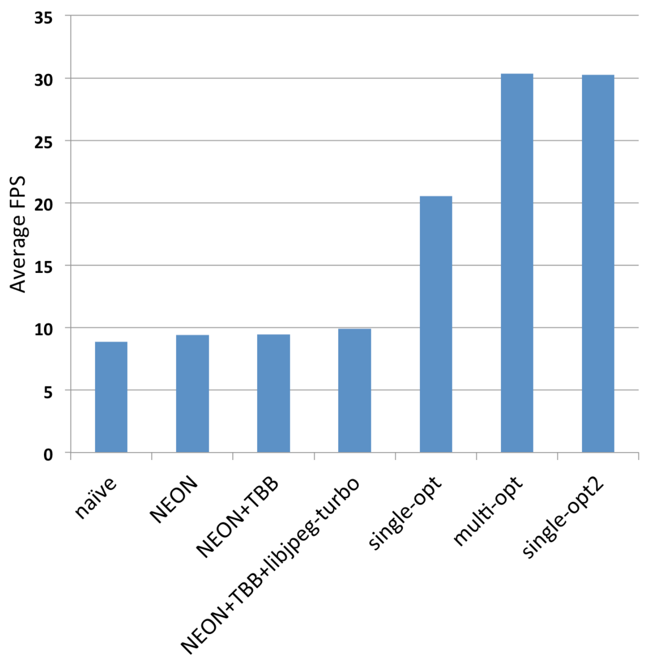

Figure 19: All implementations:
average FPS for Odroid

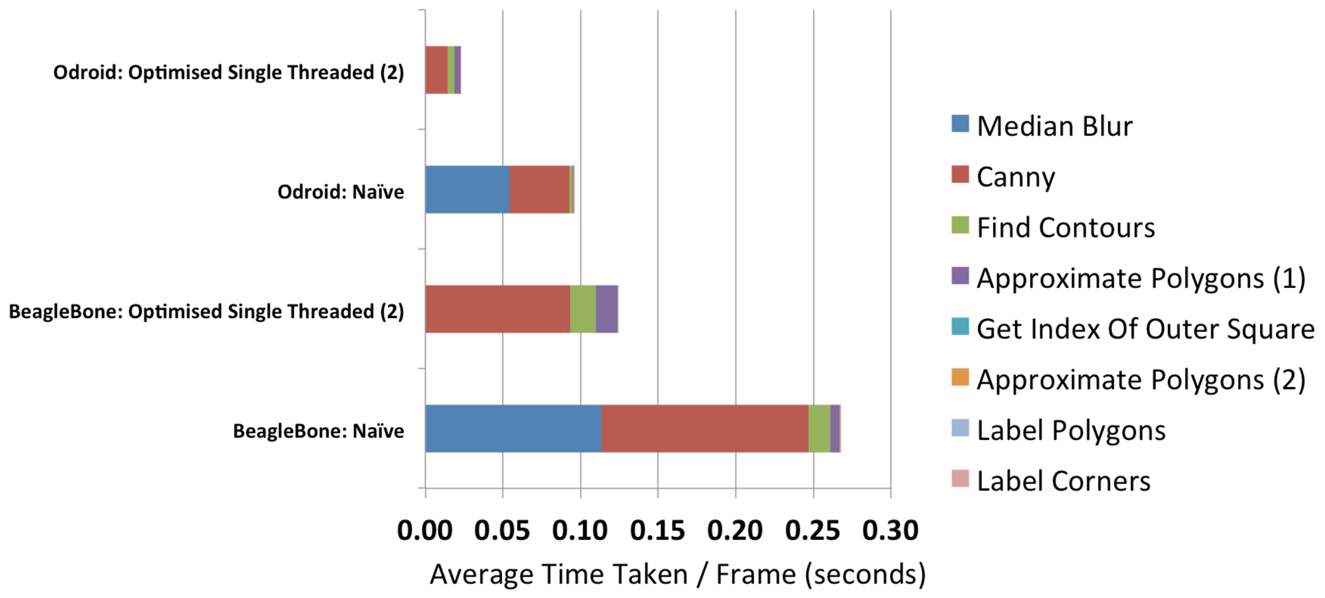

Figure 20: Naive to optimised implementation: breakdown of *Detect Corners* for BeagleBone and Odroid.