Parallelised Pose Estimation for Automated Landing

Nahush Bhanage, Hoang Nguyen, Sunil Shah
CS267 Final Project

Automated Landing

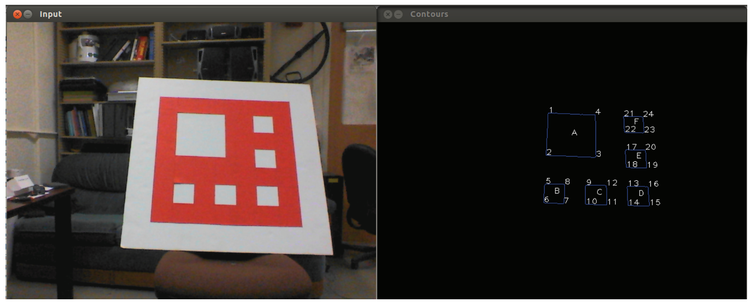
The rising popularity of unmanned aerial systems (colloquially known as ‘drones’) has been driven by the availability of low cost open source hardware, open source software and online communities that have sprung up around these products.

As these vehicles become increasingly capable and as there is pressure on the Federal Aviation Administration to create regulations and rules that permit their use for commercial ventures.

A key shortfall of the technology and one that predicates whether it is useful for much touted applications such as package delivery is that current autopilots use GPS to localise the vehicle when landing. This means that the accuracy of landing tracks the accuracy of GPS. We found, over 10 test runs, that the mean landing error (defined as the distance between takeoff and landing) was 195.33cm, with a standard deviation of 110.73cm.

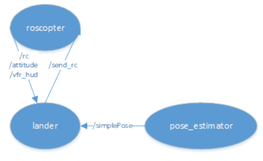
By using a known landing pattern, it is possible to use computer vision techniques to very accurately estimate the pose of the vehicle to within a few centimetres of its actual position. This approach was described by Sharp, Shakernia, and Sastry in “A Vision System for Landing an Unmanned Aerial Vehicle” in 2001.

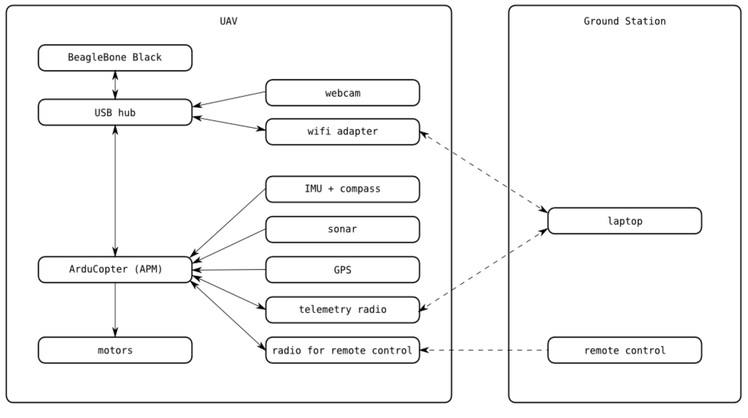
This class project aimed to improve performance of a pre-existing but poorly performing implementation of this algorithm such that it could operate fast enough to allow real-time control of the vehicle - i.e. at a rate of at least 5 Hz.



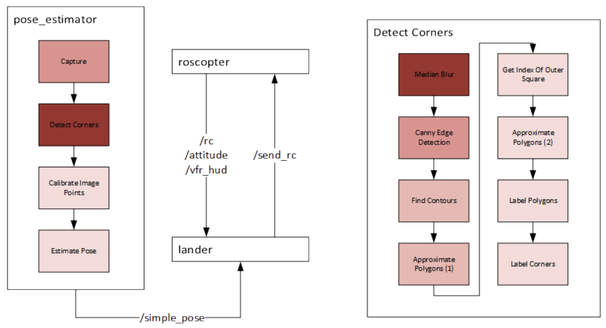
Architecture

Our system is built on open source hardware and software. We use the $45 BeagleBone Black (with a single 1 GHz core) and the $169 Hardkernel Odroid XU (with four 1.6 GHz cores) embedded computers coupled with ROS (a messaging and service framework intended for robotics systems) and OpenCV (a computer vision library). Our application is written in C++ and runs on Ubuntu Linux.





Approach



After profiling the code, we were able to highlight the hotspots in our algorithm - where it was spending the most time for each iteration. These are illustrated above, the figure on the left shows the overall algorithm, while the figure on the right shows the hotspots within just the Detect Corners subroutine. Based on this analysis, we explored a number of possible approaches to increasing the performance of this application.

1) Architecture Optimised Code Generation

gcc provides flags that turn on code generation optimised for the NEON instruction set supported by the ARM processors that both of our embedded boards utilise. OpenCV also provides NEON optimised code paths for certain methods within the library.

2) Library Tuning

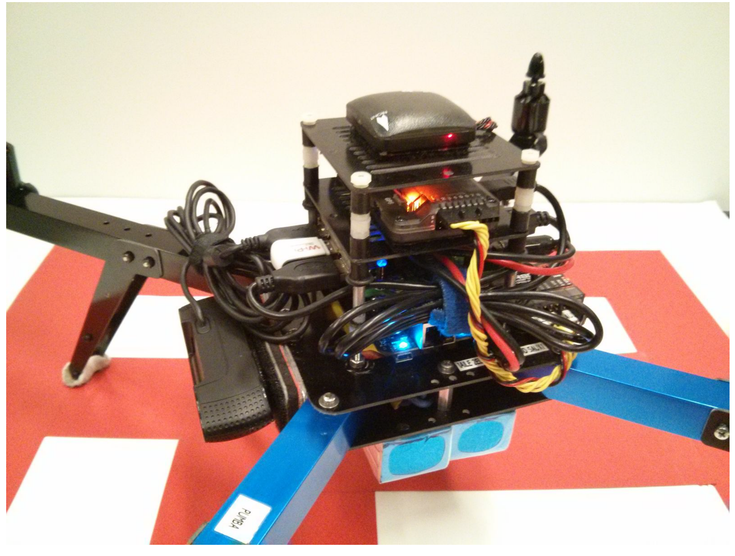
OpenCV is configurable with numerous tertiary libraries to provide faster image format parsing and multi-threading support. We enabled compilation with Intel’s Thread Building Blocks (TBB) library and the ARM optimised libjpeg-turbo.

3) Removing Redundancy

We found that we were able to retain correctness of the pose estimation algorithm and gain significant performance by removing or replacing certain steps within the Detect Corners subroutine.

4) Pipelining

Using pthreads, we were able to implement pipelining by creating a fixed size thread pool upon initialisation and by firing off successive frames to individual threads for processing.

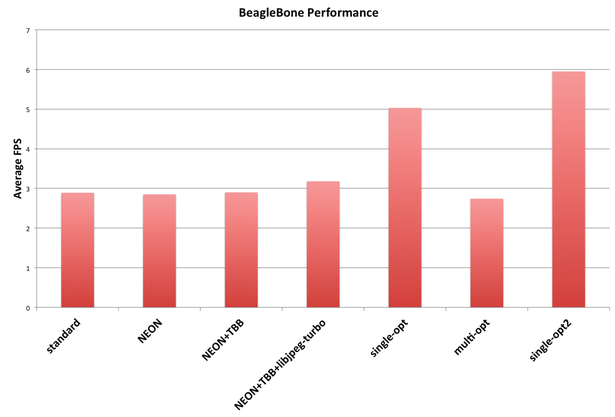


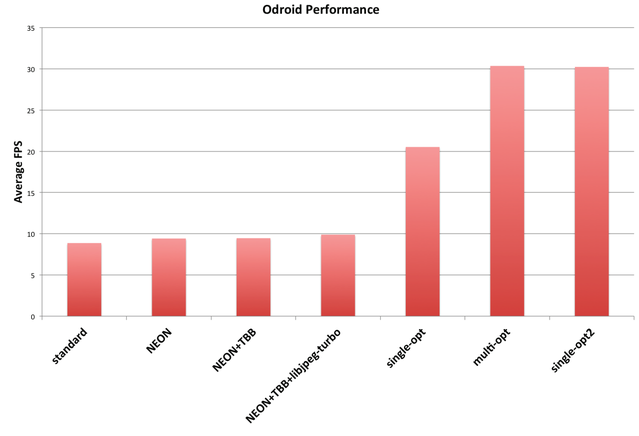
Results

Real-time control of an unmanned flying vehicle requires a rate of at least 5 Hz. The inner control loop of the popular open source ArduPilot autopilot operates at 10 Hz when reading from the onboard IMU and barometer.

Our initial implementation on the single core BeagleBone operated at just under 3 frames per second. We were able to improve this to almost double, at just under 6 frames per second. This, however, is not sufficient for real-time control, since frames can become blurred due vibration induced while flying. Since the BeagleBone has just a single core, multi-threading resulted in a performance loss due to context switching overhead. It should be possible to gain further performance by utilising NEON intrinsics to permit SIMD operation in our code. Given the complexity of implementing fast computer vision functionality, we were unable to successful get this approach to work.

The Odroid board, a four core board that operates at a faster clock speed, began at 8 frames per second with our naive implementation. Library improvements resulted in a modest boost to 10 frames per second but by removing redundant calls, we were able to get the performance to double to 20 frames per second. The addition of multi-threading allowed us to reach 30 frames per second - the maximum speed possible and more than sufficient for real-time control.





Cal UNMANNED