International Micro Air Vehicle Conference and Flight Competition (IMAV2013)
17-20 September 2013, Toulouse, France

# Optimizing Image Processing on OMAP3 with driver-level frame buffering and color space conversion

Johannes Schellen, Christian Dernehl and Stefan Kowalewski

RWTH Aachen University, Aachen, Germany
`johannes.schellen@rwth-aachen.de`
`dernehl@embedded.rwth-aachen.de`
`kowalewski@embedded.rwth-aachen.de`

**Abstract**

Onboard image processing for unmanned aerial vehicles (UAVs) has become a popular method in the recent decades and the number of available hardware solutions has increased. Growing processing power and reduced weight and size of the embedded systems facilitate more computational power onboard the UAVs, making real-time image processing feasible. With respect to the software framework, the OpenCV library provides a set of useful functions to extract information from images. In this paper we first present a latency improvement over using OpenCV for camera input and show that the frame buffer optimization results in a latency reduction of up to one fifth compared to the OpenCV library. For the second part, we explain how utilizing direct driver access and hardware capabilities enables a faster color space conversion than OpenCV library functions. The color space conversion is tested with the L*a*b color space, which proves to be the right choice for our application, which is the detection of red objects in inhomogeneous light conditions. For our outdoor MAV application, the detection of six rectangular red objects takes no longer than about 50ms on average.

## 1 Introduction

An increasing computational power and decreasing prices have increased the applications for embedded systems and allow to run complex algorithms onboard. With the growing use of such systems, the research and development in the *UAV* (unmanned aerial vehicle) area has risen, so that a higher degree of automation becomes possible. Nowadays, the development in the UAV sector has advanced so far, that such vehicles can be bought in the retail store. However, the range of UAV types, which are sold off the shelf, is limited to either rotor-craft (tri-, quad-, hexa- and octacopter) or fixed wing vehicles and are controlled either with remote control or a fixed set of pre-programmed waypoints. More expensive systems include for example a first person view system, nevertheless, onboard video processing remains a niche in this sector, although in the research and development, there are more advanced systems available. One part of the video processing, which improves the degree of automation, is the segmentation, allowing to determine objects in the foreground from the background. Depending on the objects, which should be detected, the chosen method for the video segmentation is dependent on the application. For instance, objects can be identified by attributes, such as geometry, color or contrast. In this paper we present an optimization of a capture device for the Gumstix embedded on-chip processor[1], which is used in several research and development

[1]See http://www.gumstix.org .

projects[11][12][13]. Additionally our capture device allows to gather different image formats than RGB, that is L*a*b, for example. The L*a*b color space has been chosen for our application, which is to detect an arch, with six flags attached to its borders, which has been used in the IMAV 2012 competition[2]. Regarding the performance, the adapted implementation is compared against the *Open Computer Vision* (OpenCV) framework[3]. With respect to the target flight vehicle, the Embedded Systems Laboratory and the Institute for Flight Dynamics at the RWTH Aachen University have developed since 2011 a tilt-wing UAV, with hovering and fixed-wing capabilities[7][9], in which the video system is implemented.

The structure of the paper is given in the following. After the introduction into the topic has been given in this section, the related work on this field is briefly presented. Thereafter, an overview of the system is given and approaches to decrease the latency are explained. The results of the tests are evaluated in Section 4. Finally conclusions are drawn in the last section.

## 2 Related Work

In the 80's, Waxman et al.[17] developed a prototype for an vision system, capable of detecting roads from the bird perspective. About ten years later, Solka et al.[15] present similar approaches to detect landscape marks made by humans and vision-based landing becomes available[14].

With respect to our target computer system, the Gumstix series, which have been developed and extended in the last decade, the number of onboard processing applications has increased. Salazar et al.[13] port the GPS open source Toolkit (GPSTk) to the Gumstix series, making further GPS development easier and faster. Thinking of a solution with a Kalman-filter, AggieNav developed by Clavin Coopmans, offers an integrated system where the Kalman-filter is executed on the Gumstix hardware, which is connected to other subsystems, such as the inertial measurement unit[6]. Another Kalman filter approach to estimate the position via optical flow is explained in the paper from Kendoul et al.[10]. Eynard et al.[8] demonstrate a solution to approximate the altitude from a camera pointing downwards, while the complete position is calculated by Wang et al.[16] and Phang et al.[12], for instance. Finally, Lange et al. demonstrate how vision-based landing can be achieved[11].

## 3 System Design and Implementation

After the work of other research projects has been acknowledged, the system is described and the optimization procedure is developed. First the relevant hardware and software components are illustrated. Subsequently, the latency reduction is explained. Finally, the color space conversion to L*a*b* is presented.

### 3.1 The Hardware

The system on which the computer vision application is implemented consists of a *Gumstix Overo Computer-on-Module* with a 620MHz ARM CPU and 512MB of RAM, and a *Gumstix Caspa VL* camera. The camera uses an Aptina MT9V032 wide-VGA CMOS sensor. It is connected to the *Texas Instruments OMAP3 system-on-a-chip* (TI OMAP3 SoC) on the Gumstix Overo COM via a 28 pin flex cable which carries the parallel data signal and the $I^2C$ configuration channel.

[2]See http://www.imav2012.org .
[3]See http://www.opencv.org .

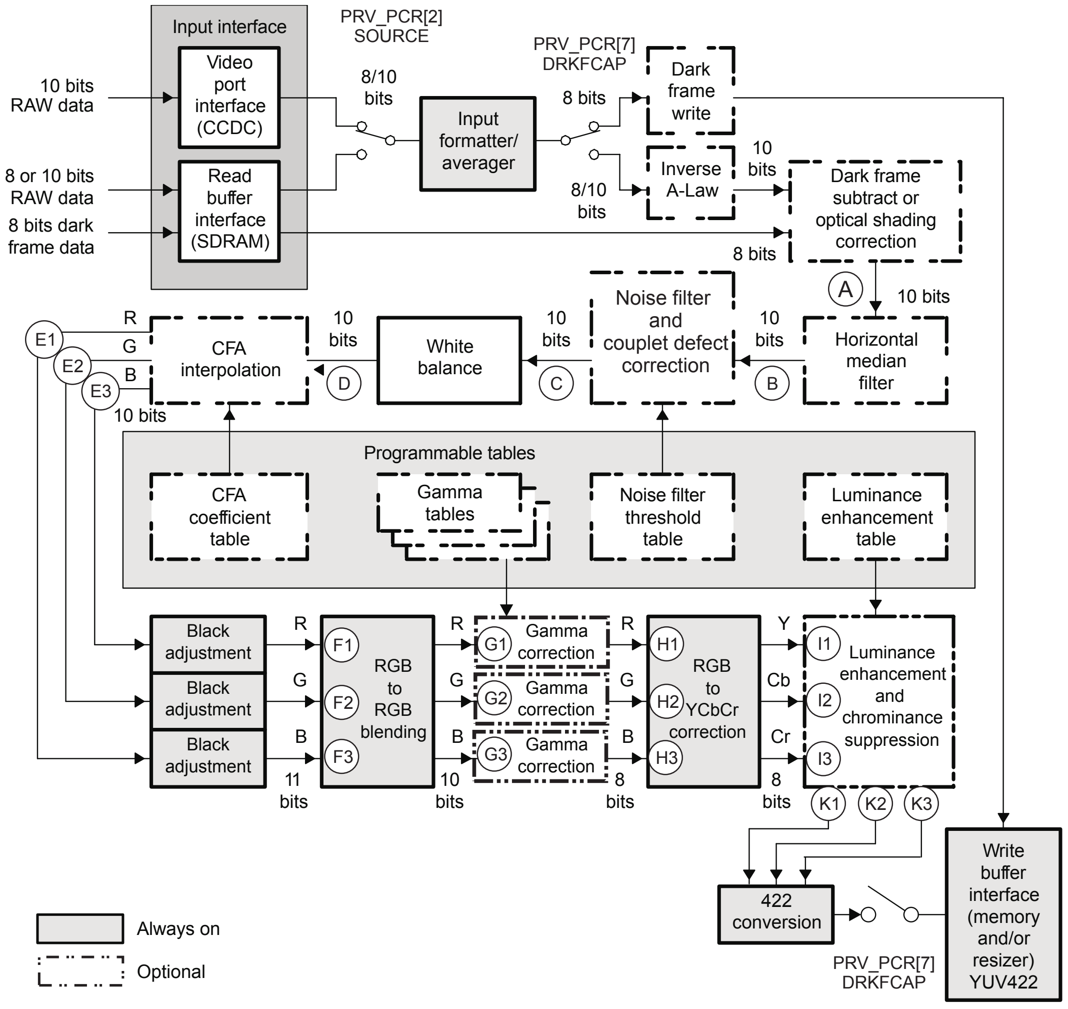

Figure 1: OMAP3 ISP Preview Engine Block Diagramm (taken from [1])

The TI OMAP3 SoC combines an ARM Cortex A8 CPU with a multitude of peripherals, such as a dedicated camera interface called *image signal processor* (OMAP3ISP). This peripheral receives sensor data from the Caspa VL camera, processes the pixels as they are read from the camera and uses DMA to write the images to main memory.

A central component of the OMAP3ISP is the *preview engine*, depicted in Figure 1, which takes the Bayer pattern sensor data, applies image enhancements and white balance scaling, interpolates the subpixel data and transforms the RGB pixels into YCbCr pixels. In Section 3.4 the reprogramming of the OMAP3ISP preview engine to output image data in the L*a*b* color space instead and the retrieving the data unmodified and with low latency is shown. First however, the implementation provided by the Open Computer Vision library is evaluated.

## 3.2 OpenCV

The OpenCV library provides many image processing and computer vision algorithms as well as data structures and I/O functions. Applications can read camera images by using the OpenCV *VideoCapture* class, which takes one argument, being either the index of the camera or the name of a video file in case a recording is to be used instead of a live camera. Image data is converted from the native pixel format of the camera into the BGR pixel format which is used throughout OpenCV.

During the evaluation of the OpenCV VideoCapture class for the given application, a high latency between events in front of the camera and their appearance in images which are read through the VideoCapture class has been measured. The cause of this latency is in the OpenCV source code file `modules/highgui/src/cap_libv4l.cpp`. According to the comments at the beginning of the file, it is derived from the example code in the Video4Linux2 API documentation. Like the example code, `cap_libv4l.cpp` uses a FIFO queue of buffers. Four buffers are reserved for the queue by default, and when an image is transferred to the application, it is copied into yet another buffer.

## 3.3 Latency Reduction

A FIFO queue is useful for avoiding dropped frames if the application, on average, processes frame buffers faster than they are captured. On the other hand, a FIFO queue fills up and overflows if the processing time per frame is even just slightly longer on average than the time between two frame captures. In Figure 2, the effect of a full queue on the latency is depicted. The top half shows the utilization of four buffers in a FIFO configuration where the processing takes three and a half times as long as the frame time. The bars below visualize the latency between the time when an image is captured and the time when it is fully processed. The thick part of the bar shows the latency building up while the frame is in the queue, and the thin part represents the latency due to the actual processing by the application.

To avoid the high latency and the conversion into the BGR color space inflicted by the OpenCV VideoCapture class, a capture module which uses the Video4Linux2 API directly has been developed. This new module is based on the example code from the Video4Linux2 API documentation as well, but uses a *double buffering* strategy instead of a queue.

The V4L2 API uses two queues of memory mapped buffers: One is the *input queue* of empty buffers which are fed into the driver and the other is the *output queue* of buffers which the driver has filled with images from the camera. The capture module appends buffers to the input queue with the `VIDIOC_QBUF` IOCTL and dequeues buffers from the output queue with the `VIDIOC_DQBUF` IOCTL. A separate thread is utilized to dequeue buffers as soon as they become available. If the application hasn't retrieved a buffer from the capture module by the time the next buffer becomes available, this thread immediately moves the older buffer to the input queue. In this way the latest completed frame is always available to be retrieved by the application, while a second framebuffer is being filled by the camera driver. The lower half of Figure 2 visualizes the concept and shows the improved timing compared to a four buffer queue. Like the queue, double buffering achieves maximum throughput by avoiding delays. But unlike the queue, double buffering keeps the latency in the capture module below one frame time.

For the implementation of this concept, a total of four buffers are used: One filled buffer awaits retrieval by the application. The driver writes to the second buffer, and the application processes the image in the third buffer. Finally, the fourth buffer waits in the input queue to avoid underruns during the time when the next buffer has been filled by the driver and the separate thread has not yet returned the older completed buffer to the input queue. Buffers swap roles as necessary so that no image data needs to be copied in the process.

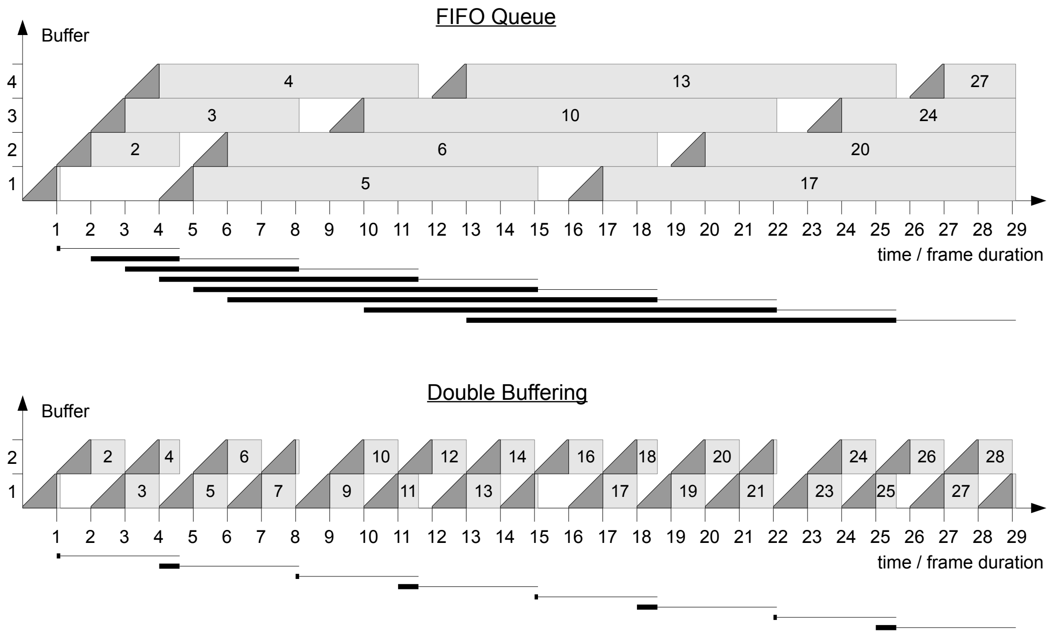

Figure 2: Latency of image capturing with a FIFO queue compared to double buffering

## 3.4 L*a*b* Color Space Conversion

With a low latency method for retrieving image data from the camera interface in place, focus is set on getting the data in the desired color space. The *CIE L* a* b** [4] color space is defined by a conversion from the *CIE XYZ* [2] color space. In addition to this conversion, more processing steps are necessary if the original data is in a color space other than XYZ. In order to convert from linear RGB data to L*a*b* data, the data is first converted to XYZ by a linear transformation. If the original data is non-linear RGB, then an inverse gamma correction is applied before the conversion to XYZ. Starting with YUV data prepends another matrix multiplication to get RGB. Reading camera images in the usual YUV format and then performing the necessary transformation steps on the CPU creates a high overhead. This overhead can be avoided by changing the parameters of the OMAP3ISP preview engine to produce L*a*b* image data instead of YCbCr data. This offloads the calculations from the CPU and frees processing time for later stages of the given computer vision application. Direct use of the Video4Linux2 API enables the reading of the output data of the OMAP3ISP preview engine without further conversions.

Figure 1 shows the hardware function blocks on which the conversion is performed: After the color filter array interpolation and black level adjustment, the linear RGB pixels are processed by the *RGB to RGB blending*, *Gamma correction* and *RGB to YCbCr conversion* function blocks. The gamma correction is implemented as a lookup into three tables of 1024 unsigned 8-bit values each. A 3x3 matrix multiplication and the addition of an offset vector are executed for each of the *RGB to RGB blending* and *RGB to YCbCr conversion* steps. These calculations are performed in fixed point arithmetic, and the matrix and vector components are limited to different ranges and resolutions. The detailed specification can be found in the literature [1].

The starting point for the conversion is linear RGB, and two matrix multiplications with a gamma conversion in between can be used to perform the necessary calculations. With this in mind, we take a closer look at the mathematical background.

Converting from linear RGB to L*a*b* is a two step procedure: First linear RGB is transformed to CIE XYZ and afterwards converted to CIE L*a*b*. For linear RGB data with the primary colors and the white point of the sRGB [5] color space, the conversion to CIE XYZ is performed by applying the linear transformation in Equation (1). The conversion to L*a*b* is defined by Equation (2). The tristimulus values $X_n, Y_n$ and $Z_n$ for the white point D65 [3] are given in Equation (3).

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412391 & 0.357584 & 0.180481 \\ 0.212639 & 0.715169 & 0.072192 \\ 0.019331 & 0.119195 & 0.950532 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{1}$$

$$\begin{aligned} L &= 116\ f(Y/Y_n) - 16 \\ a &= 500[f(X/X_n) - f(Y/Y_n)] \\ b &= 200[f(Y/Y_n) - f(Z/Z_n)] \\ f(x) &= \begin{cases} x^{1/3} & \text{if } x > (6/29)^3 \\ (841/108)x + 4/29 & \text{if } x \leq (6/29)^3 \end{cases} \end{aligned} \tag{2}$$

$$\begin{bmatrix} X_n \\ Y_n \\ Z_n \end{bmatrix} = \begin{bmatrix} 0,950456 \\ 1 \\ 1,089058 \end{bmatrix} \tag{3}$$

To map this conversion to the hardware, the conversion is decomposed into two matrix multiplications and one gamma correction as expressed in Equation (4), where $F$ is the component-wise application of $f$ from Equation (2).

$$\begin{aligned} \begin{bmatrix} L \\ a \\ b \end{bmatrix} &= \begin{bmatrix} -16 \\ 0 \\ 0 \end{bmatrix} + M_{Lab} F \left( M_{XYZ} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \right) \\ M_{Lab} &= \begin{bmatrix} 0 & 116 & 0 \\ 500 & -500 & 0 \\ 0 & 200 & -200 \end{bmatrix} \\ M_{XYZ} &= \begin{bmatrix} 0.412391/0.950456 & 0.357584/0.950456 & 0.180481/0.950456 \\ 0.212639 & 0.715169 & 0.072192 \\ 0.019331/1.089058 & 0.119195/1.089058 & 0.950532/1.089058 \end{bmatrix} \end{aligned} \tag{4}$$

The L* component ranges from 0 to 100 and the a* and b* components are in the -127 to 127 interval. While the L*a*b* conversion is specified for an RGB input range of 0 to 1, here the input values are in the range from 0 to 255. Accordingly, a scaling factor of $\frac{1}{255}$ is applied to the $M_{Lab}$ matrix.

The output consists of 8-bit unsigned integers. In order to use the available resolution, the L*a*b* values are mapped to the full 0 to 255 interval. For the a* and b* components in the Cb and Cr channels, the required addition of 127 is implicitly performed by the OMAP3ISP preview engine, so no offsets need to be programmed for the color channels. The lightness channel however needs further attention. The output range is expanded from the 0 to 100 interval, and since the OMAP3ISP preview engine does not support negative offsets in the final processing step, the L* component is approximated according to Equation (5).

$$L = \frac{255}{100} \cdot \frac{116}{255} \cdot f(Y) - 16 \approx \frac{255}{100} \cdot \frac{100}{255} \cdot f(Y) = f(Y) \tag{5}$$

Consequently, the matrix to be programmed into the *RGB to YCbCr conversion* step of the OMAP3ISP preview engine is given in Equation (6).

$$M'_{Lab} = \begin{bmatrix} 0 & 1 & 0 \\ 1.960784 & -1.960784 & 0 \\ 0 & 0.784313 & -0.784313 \end{bmatrix} \tag{6}$$

The *RGB to RGB blending* step is programmed with the $M_{XYZ}$ matrix, and the gamma tables are computed according to the $f$ function from Equation (2), but scaled to an input range of 0 to 1023 and an output range of 0 to 255.

When these configuration parameters are programmed using the `VIDIOC_OMAP3ISP_PRV_CFG` private IOCTL, the preview engine outputs 4:2:2 sampled L*a*b* data with a rescaled and offset L* channel. The kernel V4L2 driver is unaware of the modified configuration and continues to treat the data like 4:2:2 sampled YCbCr data, so that is the format which the application has to request in order to receive the L*a*b* data.

## 4 Evaluation

For evaluation, a comparison to the OpenCV framework is presented regarding two aspects: In the first part, the throughput performance is shown with respect to the target platform, which has been introduced in Section 3.1. Thereafter, the latency reduction is measured using another ARM based system.

In the target application, the autonomous navigation of a MAV is aided by location information computed from visually recognizing an arch which is marked with six red flags. The algorithm for which the processing time is measured and plotted in Figure 3 consists of thresholding, correcting lens distortion, computing the center of gravity and size of each connected component, searching for a set of connected components which represent the six flags and computing the pose based on four point correspondences. Peaks within the plot result from a high number of different red objects within the scene or the entire absence of the arch. Both are situations in which many combinations of connected components have to be considered and rejected. Nevertheless, in many situations the entire algorithm completes in 40 milliseconds or less. In comparison, the conversion of a video frame from BGR to L*a*b* using the OpenCV `cvtColor` function takes about 40ms on the same hardware, on top of the overhead created by the YUV to BGR conversion before the frame is delivered to the application.

For the latency measurements, a Seagate Dockstar with a Logitech C270 USB camera has been selected as test environment. In the Dockstar, the Marvell Kirkwood 88F6281 SoC contains an ARMv5TE core at 1.2GHz. The system is equipped with 128MB of RAM. A two-color front panel LED is attached to two GPIOs. For the latency test, the camera, capturing 30 frames per second, is pointed at the LED. The test program turns the LED on and off at random times and measures the time until the state change is detected in a camera image by reading a single pixel from the image and comparing it to a threshold value. To simulate processing, the test program inserts a defined delay between the capturing of the image and the reading of the LED state. Figure 4 shows the measured latencies of test runs with OpenCV on one hand and our capture module on the other hand, each with 100ms and 500ms processing delays.

The measured latency matches the expected latency from theoretical considerations. Note that our implementation still achieves almost the same average latency with 500ms processing time in the application as the OpenCV VideoCapture class when only 100ms of processing time are available to the application per frame.

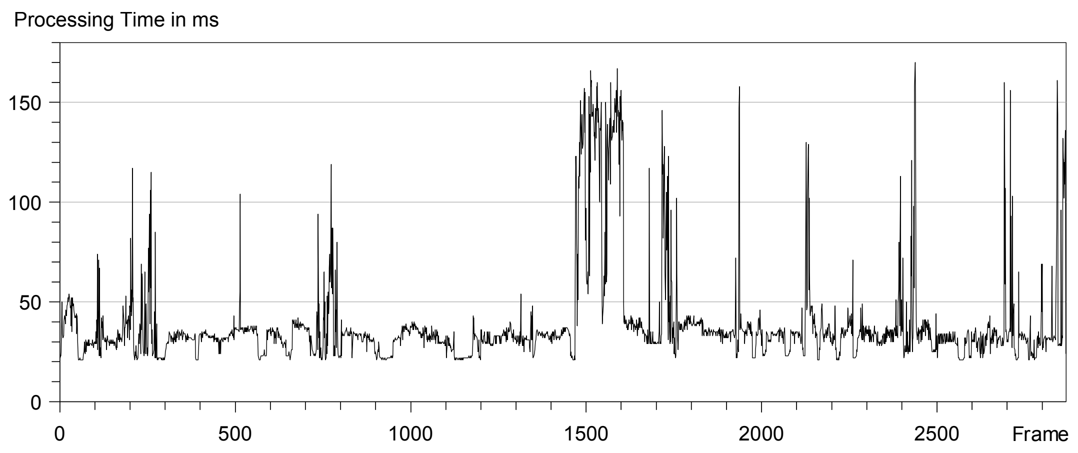

Figure 3: Measured processing time of the entire image recognition algorithm

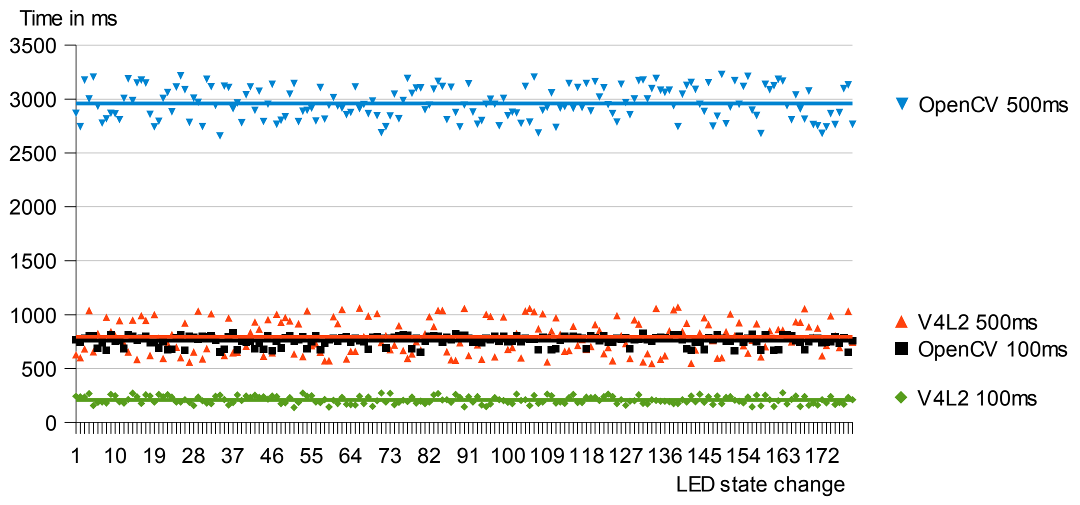

Figure 4: Measured latency from an LED state change to the detection in the camera image, with processing delays of 100ms and 500ms

## 5 Conclusion

In this paper we present an approach to improve the latency in comparison to the OpenCV implementation. Using double buffering instead of a FIFO queue allows the developer to implement more time critical applications. This improvement does not depend on properties of the Gumstix system and works on other hardware platforms as well.

Furthermore, we have demonstrated that utilizing the V4L2 API directly avoids the forced conversion to BGR and thereby opens up the possibility of moving the color space conversion to the OMAP3ISP preview engine. Compared to a software-only implementation with the OpenCV `cvtColor` library function, offloading the conversion to the dedicated hardware peripheral reduces the processing time by more than 40ms per frame. This step allowed us to double the frame rate of our application in many environments.

## References

[1] *OMAP35x Applications Processor Technical Reference Manual*, 2012.

[2] CIE. Commission internationale de l'eclairage proceedings, 1931.

[3] CIE. 15-2004, colorimetry, 3rd edition, 2004.

[4] CIE. Iso 11664-4:2008 (cie s 014-4/e:2007), colorimetry, part 4: Cie 1976 l*a*b* colour space, 2008.

[5] International Electrotechnical Commission. Multimedia systems and equipment, colour measurement and management, part 2-1: Colour management, default rgb colour space srgb, 1999.

[6] Calvin Coopmans. Aggienav: A small, well integrated navigation sensor system for small unmanned aerial vehicles. In *ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2009.

[7] Christian Dernehl, Dominik Franke, Hilal Diab, and Stefan Kowalewski. An architecture with integrated image processing for autonomous micro aerial vehicles. In *International Micro Air Vehicle Conference (IMAV)*, pages 138 – 145. IMAV, 2011.

[8] D. Eynard, P. Vasseur, C. Demonceaux, and V. Fremont. Uav altitude estimation by mixed stereoscopic vision. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 646–651, 2010.

[9] Felix Gathmann, Christian Dernehl, Dominik Franke, and Stefan Kowalewski. An integrated vision aided gps/ins navigation system for ultra-low-cost mavs. In *International Micro Air Vehicle Conference (IMAV)*, pages 1 – 8. IMAV, 2012.

[10] Farid Kendoul, Isabelle Fantoni, and Kenzo Nonami. Optic flow-based vision system for autonomous 3d localization and control of small aerial vehicles. *Robotics and Autonomous Systems*, 57(67):591 – 602, 2009.

[11] S. Lange, N. Sunderhauf, and P. Protzel. A vision based onboard approach for landing and position control of an autonomous multirotor uav in gps-denied environments. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*, pages 1–6, 2009.

[12] Swee King Phang, Jun Jie Ong, R.T.C. Yeo, B.M. Chen, and T.H. Lee. Autonomous mini-uav for indoor flight with embedded on-board vision processing as navigation system. In *Computational Technologies in Electrical and Electronics Engineering (SIBIRCON), 2010 IEEE Region 8 International Conference on*, pages 722–727, 2010.

[13] D. Salazar, M. Hernandez-Pajares, J.M. Juan, and J. Sanz. Rapid Open Source GPS software development for modern embedded systems: Using the GPSTk with the Gumstix. . Technical report, Grupo de Astronomia y Geomatica (gAGE), Universitat Politecnica de Catalunya, 2003.

[14] O. Shakernia, Yi Ma, T.J. Koo, J. Hespanha, and S.S. Sastry. Vision guided landing of an unmanned air vehicle. In *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, volume 4, pages 4143–4148 vol.4, 1999.

[15] J.L. Solka, D.J. Marchette, B. C. Wallet, V. L. Irwin, and G.W. Rogers. Identification of man-made regions in unmanned aerial vehicle imagery and videos. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):852–857, 1998.

[16] Fei Wang, Tao Wang, B.M. Chen, and T.H. Lee. An indoor unmanned coaxial rotorcraft system with vision positioning. In *Control and Automation (ICCA), 2010 8th IEEE International Conference on*, pages 291–296, 2010.

[17] A.M. Waxman, J. Le Moigne, L.S. Davis, Eli Liang, and T. Siddalingaiah. A visual navigation system. In *Robotics and Automation. Proceedings. 1986 IEEE International Conference on*, volume 3, pages 1600–1606, 1986.