FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Android-based implementation of Eulerian Video Magnification for vital signs monitoring

Pedro Boloto Chambino

WORKING VERSION

U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Luís Teixeira

Supervisor at Fraunhofer Portugal: Luís Rosado

June 7, 2013

Android-based implementation of Eulerian Video Magnification for vital signs monitoring

Pedro Boloto Chambino

Mestrado Integrado em Engenharia Informática e Computação

June 7, 2013

Abstract

Eulerian Video Magnification is a recently presented method capable of revealing temporal variations in videos that are impossible to see with the naked eye. Using this method, it is possible to visualize the flow of blood as it fills the face. From its result, a person's heart rate is possible to be extracted.

This research work is an internal project of *Fraunhofer Portugal* and its goal is to test the feasibility of the implementation of the Eulerian Video Magnification method on smartphones by developing an *Android* application for monitoring vital signs based on the Eulerian Video Magnification method.

There has been some successful effort on the assessment of vital signs, such as, heart rate, and breathing rate, in a contact-free way using a webcamera and even a smartphone. However, since the Eulerian Video Magnification method was recently proposed, its implementation has not been tested in smartphones yet.

The application will include features, such as, detection of a person's cardiac pulse, dealing with artifacts' motion, and real-time display of the magnified blood flow. Then, the application performance will be evaluated through tests with several individuals and the assessed heart rate compared to the one detected by the *Philips* application, and to the measurement of an heart rate monitor or a pulse oximeter.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

EVM	Eulerian Video Magnification
ICA	Independent Component Analysis
PPG	Photo-plethysmography
FFT	Fast Fourier transform
FPS	Frames per second
JNI	Java Native Interface
JVM	Java Virtual Machine
OpenCV	Open Source Computer Vision Library
IIR	Infinite impulse response

Chapter 1

Introduction

This chapter introduces this work, by first presenting its context, motivation, and project's objectives, on sections 1.1, 1.2, and 1.3, respectively.

Finally, section 1.4 describes the document outline.

1.1 Context

Eulerian Video Magnification is a method, recently presented at *SIGGRAPH*[1] 2012, capable of revealing temporal variations in videos that are impossible to see with the naked eye. Using this method, it is possible to visualize the flow of blood as it fills the face [WRS+12]. Which provides enough information to assess the heart rate in a contact-free way using a camera [WRS+12, PMP10, PMP11].

The main field of this research work is *image processing and computer vision*, whose main purpose is to translate dimensional data from the real world in the form of images into numerical or symbolical information.

Other fields include *medical applications*, *software development for mobile devices*, *digital signal processing*.

This research work is an internal project of *Fraunhofer Portugal*[2] supervised by Luís Rosado. Fraunhofer Portugal a is non-profit private association founded by Fraunhofer-Gesellschaft[3] [Por13] and

> "aims on the creation of scientific knowledge capable of generating added value to its clients and partners, exploring technology innovations oriented towards economic growth, the social well-being and the improvement of the quality of life of its end-users." [Por13]

[1] http://www.siggraph.org/
[2] http://www.fraunhofer.pt/
[3] http://www.fraunhofer.de/en/about-fraunhofer/

1.2 Motivation

Due to being recently proposed, the Eulerian Video Magnification method implementation has not been tested in smartphones yet.

There has been some successful effort on the assessment of vital signs, such as, heart rate, and breathing rate, in a contact-free way using a webcamera [WRS+12, PMP10, PMP11], and even a smartphone [Tec13, Phi13].

Other similar products, which require specialist hardware and are thus expensive, include *laser Doppler* [UT93], *microwave Doppler radar* [Gre97], and *thermal imaging* [GSMP07].

Since it is a cheaper method of assessing vital signs in a contact-free way than the above products, this research work has potential for advancing fields, such as, *telemedicine*, *personal health-care*, and *ambient assisting living*.

Despite the existence of very similar products by *Philips* [Phi13] and *ViTrox Technologies* [Tec13] to the one proposed on this research work, none of these implements the Eulerian Video Magnification method. Moreover, the application to be developed during this research work will have additional features described on the next section 1.3.

1.3 Objectives

This research work goal is to test the feasibility of the implementation of the Eulerian Video Magnification method on smartphones by developing an *Android* application for monitoring vital signs based on the Eulerian Video Magnification method.

This application should include the following features:

- heart rate detection and assessment based on the Eulerian Video Magnification method;
- display real-time changes, such as, the magnified blood flow, obtained from the Eulerian Video Magnification method;
- deal with artifacts' motion, due to, person and/or smartphone movement.

It should be noted that a straightforward implementation of the Eulerian Video Magnification method is not possible, due to various reasons. First, the Eulerian Video Magnification method provides motion magnification along with color magnification which will introduce several problems with artifacts' motion. Second, the requirement of implementing a real-time smartphone application will create performance issues which will have to be addressed and trade-offs will have to be considered.

The application performance should then be evaluated through tests with several individuals and the assessed heart rate compared to the ones detected by another application [Tec13, Phi13], and to the measurement of an electronic sphygmomanometer.

1.4 Outline

The rest of the document is structured as follows:

Chapter 2 introduces the concepts necessary to understand the presented problem. In addition, it presents the existing related work, and a description of the technologies to be used.

Chapter 3 ...

Chapter ?? presents the approach taken to solve the problem. Moreover, it introduces the testing and evaluation methodologies.

Chapter 4 ...

Chapter 5 ...

Chapter 6 ...

Chapter 2

State of the art

This chapter presents several studies regarding the heart rate estimation from a person's face captured through a simple webcam.

Section 2.1 describe the concept that explains how the cardiac pulse is detected from a person's face in a remote, contact-free way.

Post-processing methods, which may be applied to the retrieved signal, are detailed on section 2.2.

In order to estimate the heart rate, a couple of techniques are also detailed on section 2.3.

Finally, section 2.4 reviews the main technologies and tools used throughout this work.

2.1 Photo-plethysmography

Photo-plethysmography (PPG) is the concept of measuring volumetric changes of an organ optically. Its most established use is in pulse oximeters.

PPG is based on the principle that blood absorbs more light than surrounding tissue thus variations on blood volume affect light reflectance [VSN08].

The use of dedicated light sources and infra-red wavelengths, and contact probes has been the norm [UT93, Gre97, GSMP07]. However, recently, remote, non-contact PPG imaging has been explored.

The method used on the article [VSN08] captures the pixel values (red, green, and blue channels) of the facial area of a previously recorded video where volunteers were asked to minimize movements. The pixel values within a region of interest (ROI) was then averaged for each frame. This spatial averaging was found to significantly increase signal-to-noise ratio. The heart rate estimation was then calculated by applying Fast Fourier transforms and the power spectrum as explained on section 2.3.1.

The authors of [VSN08] demonstrate that the fact that the green channel features a stronger heart rate signal as compared to the red and blue channels, is a strong evidence that the signal is due to variations in blood volume because (oxy-) hemoglobin absorbs green light.

2.2 Signal post-processing

After obtaining the raw pixel values (red, green, and blue channels), a conjunction of the following methods may be used to extract and improve the reflected plethysmography signal. However, each method introduces complexity and expensive computation.

2.2.1 Independent Component Analysis

Independent Component Analysis is a special case of *blind source separation* and is a relatively new technique for uncovering independent signals from a set of observations that are composed of linear mixtures of the underlying sources [Com94].

In this case, the underlying source signal of interest is the cardiac pulse that propagates throughout the body, which modify the path length of the incident ambient light due to volumetric changes in the facial blood vessels during the cardiac cycle, such that subsequent changes in amount of reflected light indicate the timing of cardiovascular events.

By recording a video of the facial region, the red, green, and blue (RGB) color sensors pick up a mixture of the reflected plethysmographic signal along with other sources of fluctuations in light due to artifacts. Each color sensor records a mixture of the original source signals with slightly different weights. These observed signals from the red, green and blue color sensors are denoted by $x_1(t)$, $x_2(t)$ and $x_3(t)$ respectively, which are amplitudes of the recorded signals at time point t. In conventional Independent Component Analysis model the number of recoverable sources cannot exceed the number of observations, thus three underlying source signals were assumed, represented by $s_1(t)$, $s_2(t)$ and $s_3(t)$. The Independent Component Analysis model assumes that the observed signals are linear mixtures of the sources, i.e. $x_i(t) = \sum_{j=1}^{3} a_{ij} s_j(t)$ for each $i = 1,2,3$. This can be represented compactly by the mixing equation

$$x(t) = As(t) \tag{2.1}$$

where the column vectors $x(t) = [x_1(t), x_2(t), x_3(t)]^T$, $s(t) = [s_1(t), s_2(t), s_3(t)]^T$ and the square 3×3 matrix A contains the mixture coefficients a_{ij}. The aim of Independent Component Analysis model is to find a separating or demixing matrix W that is an approximation of the inverse of the original mixing matrix A whose output

$$\hat{s}(t) = Wx(t) \tag{2.2}$$

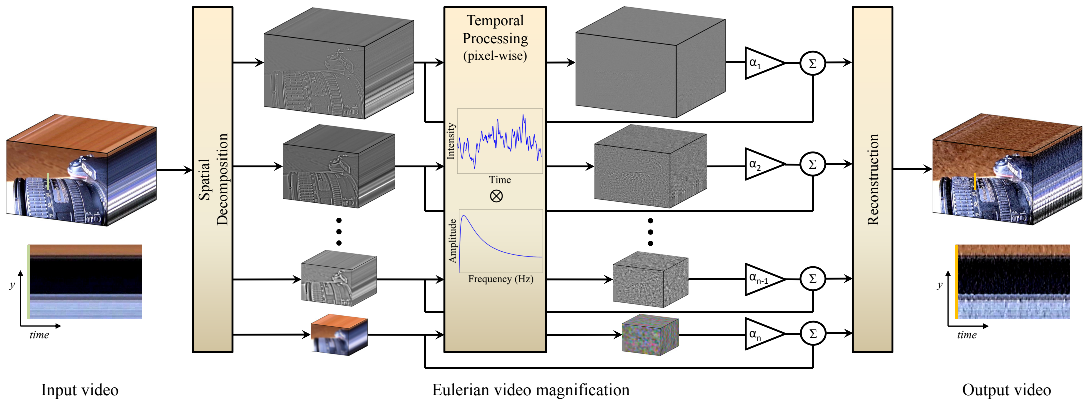


Figure 2.1: Overview of the Eulerian Video Magnification method.

is an estimate of the vector $s(t)$ containing the underlying source signals. To uncover the independent sources, W must maximize the non-Gaussianity of each source. In practice, iterative methods are used to maximize or minimize a given cost function that measures non-Gaussianity [PMP10, PMP11].

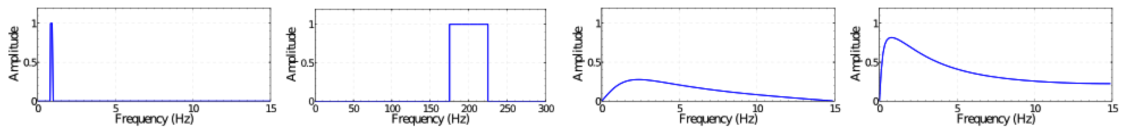
2.2.2 Eulerian Video Magnification

In contrast to the Independent Component Analysis model that focus on extracting a single number, the Eulerian Video Magnification uses localized spatial pooling and temporal filtering to extract and reveal visually the signal corresponding to the cardiac pulse. This allows for amplification and visualization of the heart rate signal at each location on the face. This creates potential for monitoring and diagnostic applications to medicine, i.e. the asymmetry in facial blood flow can be a symptom of arterial problems.

Besides color amplification, the Eulerian Video Magnification method is also able to reveal low-amplitude motion which may be hard or impossible for humans to see. Previous attempts to unveil imperceptible motions in videos have been made, such as, [LTF+05] which follows a *Lagrangian* perspective, as in fluid dynamics where the trajectory of particles is tracked over time. By relying on accurate motion estimation and additional techniques to produce good quality synthesis, such as, motion segmentation and image in-painting, the algorithm complexity and computation is expensive and difficult.

On the contrary, the Eulerian Video Magnification method is inspired by the *Eulerian* perspective, where properties of a voxel of fluid, such as pressure and velocity, evolve over time. The approach of this method to motion magnification is the exaggeration of motion by amplifying temporal color changes at fixed positions, instead of, explicitly estimation of motion.

This method approach, illustrated in figure 2.1, combines spatial and temporal processing to emphasize subtle temporal changes in a video. First, the video sequence is decomposed into different spatial frequency bands. Because they may exhibit different signal-to-noise ratios, they may be magnified differently. In the general case, the full Laplacian pyramid [BA83] may be



(a) Ideal 0.8-1 Hz (b) Ideal 175-225 Hz (c) Butterworth 3.6-6.2 Hz (d) Second-order IIR

Figure 2.2: Examples of temporal filters.

computed. Then, temporal processing is performed on each spatial band. The temporal processing is uniform for all spatial bands, and for all pixels within each band. After that, the extracted bandpass signal is magnified by a factor of α, which can be specified by the user, and may be attenuated automatically. Finally, the magnified signal is added to the original and the spatial pyramid collapsed to obtain the final output.

2.2.2.1 Spatial filtering

As mention before, the work of [WRS+12] computes the full Laplacian pyramid [BA83] as a general case for spatial filtering. Each layer of the pyramid may be magnified differently because it may exhibit different signal-to-noise ratios, or contain spatial frequencies for which the linear approximation used in motion magnification does not hold [WRS+12, Section 3].

Spatial filtering may also be used to significantly increases signal-to-noise ratio, as previously mention on section 2.1 and demonstrated on the work of [VSN08] and [WRS+12]. Subtle signals, such as, a person's heart rate from a video of its face, may be enhanced this way. For this purpose the work of [WRS+12] computes a layer of the Gaussian pyramid which may be obtained by successively scaling down the image by calculating the Gaussian average for each pixel.

However, for the signal of interest to be revealed, the spatial filter applied must be large enough. Section 5 of [WRS+12] provides an equation to estimate the size for a spatial filter needed to reveal a signal at a certain noise power level:

$$S(\lambda) = S(r) = \sigma'^2 = k\frac{\sigma^2}{r^2} \tag{2.3}$$

where $S(\lambda)$ represents the signal over spatial frequencies, and since the wavelength, λ, cutoff of a spatial filter is proportional to its radius, r, the signal may be represented as $S(r)$. The noise power, σ^2, can be estimated using to the technique of [LFSK06]. Finally, because the filtered noise power level, σ'^2, is inversely proportional to r^2, it is possible to solve the equation for r, where k is a constant that depends on the shape of the low pass filter.

2.2.2.2 Temporal filtering

Temporal filtering is used to extract the motions or signals to be amplified. Thus, the filter choice is application dependent. For motion magnification, a broad bandpass filter, such as, the butterworth filter, is preferred. A narrow bandpass filter produces a more noise-free result for color

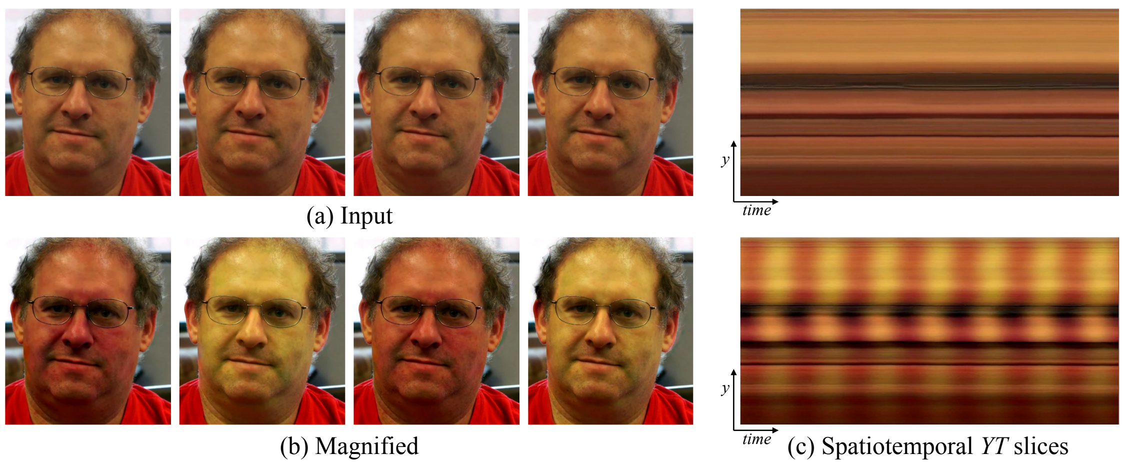


Figure 2.3: Emphasis of face color changes using the Eulerian Video Magnification method.

amplification of blood flow. An ideal bandpass filter is used on [WRS+12] due to its sharp cutoff frequencies. Alternatively, for a real-time implementation low-order IIR filters can be useful for both: color amplification and motion magnification. These filters are illustrated on 2.2.

2.2.2.3 Emphasize color variations for human pulse

The extraction of a person's cardiac pulse using the Eulerian Video Magnification method was demonstrated in [WRS+12]. It was also presented that using the right configuration can help extract the desired signal. There are four steps to take when processing a video using the Eulerian Video Magnification method:

1. select a temporal bandpass filter;
2. select an amplification factor, α;
3. select a spatial frequency cutoff (specified by spatial wavelength, λ_c) beyond which an attenuated version of α is used;
4. select the form of the attenuation for α —- either force α to zero for all $\lambda < \lambda_c$, or linearly scale α down to zero.

For human pulse color variation, two temporal filters may be used, first selecting frequencies within 0.4-4Hz, corresponding to 24-240 beats per minute (bpm), then a narrow band of 0.83-1Hz (50-60 bpm) may be used, if the extraction of the pulse rate was successful.

To emphasize the color change as much as possible, a large amplification factor, $\alpha \approx 100$, and spatial frequency cutoff, $\lambda_c \approx 1000$, is applied. With an attenuation of α to zero for spatial wavelengths below λ_c.

The resulting output can be seen in figure 2.3.

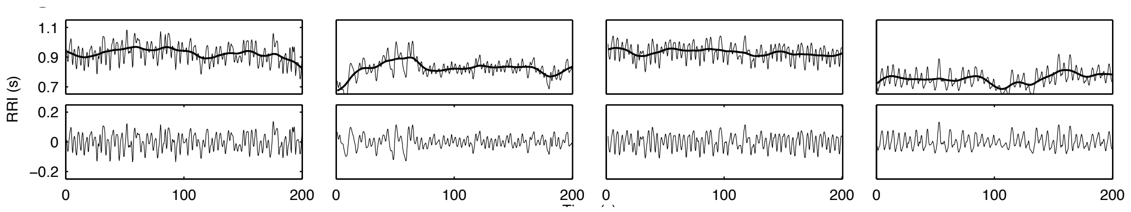


Figure 2.4: Original and detrended RR series.

2.2.3 Detrending

Detrending is a method of removing very large ultralow-frequency trends an input signal without any magnitude distortion, acting as an high-pass filter.

The main advantage of the method presented on the work of [TRaK02], compared to methods presented in [LOCS95] and [PB90], is its simplicity.

The method consists of separating the input signal, z, into two components, as $z = z_{stat} + z_{trend}$, where z_{stat} is the nearly stationary component, and z_{trend} is the low frequency aperiodic trend component.

An estimation of the nearly stationary component, $\hat{z}_{stat}$, can be obtained using the equation below. The detailed derivation of the equation can be found in [TRaK02].

$$\hat{z}_{stat} = (I - (I + \lambda^2 D_2^T D_2)^{-1})z \tag{2.4}$$

where I is the identity matrix, D_2 is the discrete approximation of the second order, and λ is the regularization parameter.

Figure 2.4 presents an example of what this method is able to achieve. The example, taken from the work of [TRaK02], uses real RR series and the effect of the method on time and frequency domain analysis of heart rate variability is demonstrated not to lose any useful information.

2.3 Heart rate estimation

In order to convert the extracted plethysmographic signal into the number of beats per minute (bpm), further processing must be done. Below are highlighted two methods capable of achieving this goal.

2.3.1 Power spectrum

Fourier transform is a mathematical transform capable of converting a function of time, $f(t)$, into a new function representing the frequency domain of the original function.

To calculate the power spectrum, the resulting function from the *Fourier transform* is then multiplied by itself.

Since the values are captured from a video, sequence of frames, the function of time is actually discrete, with a frequency rate equal to the video frame rate, FPS.

The *index*, i, corresponding to the maximum of the power spectrum can then be converted into a frequency value, F, using the equation:

$$F = \frac{i * FPS}{2N} \tag{2.5}$$

where N is the size of the signal extracted. F can then be multiplied by 60 to convert it to beats per minute, and have an estimation of the heart rate from the extracted signal.

2.3.2 Pulse wave detection

In [NI10], it is presented an automated algorithm for fast pulse wave detection. The algorithm is capable of obtaining an estimative of the heart rate from PPG signal, as an alternative to the power spectrum described above. Moreover, it also introduces validation to the waveform detection by verifying its shape and timing. Below is presented a simplified description of the algorithm. A more detailed description can be found in [NI10].

1. Identification of possible peaks and foots of individual pulses

    (a) Maximum (MAX)
    The signal is divided into consecutive $200ms$ time intervals and for every segment the absolute maximum is determined. Some of these maximums are rejected: if they fall below a predetermined amplitude threshold; or if the distance between two maximums is less than or equal to $200ms$, then the lower maximum is rejected.

    (b) Minimum (MIN)
    The absolute minimum is determined between every two adjacent maximums. A minimum is rejected, it is above a predetermined amplitude threshold. When a minimum is rejected, the lower-amplitude maximum of the two maximum adjacent to the rejected minimum is discarded too.

2. Examination and verification of the rising edges

    (a) Validation of a single rising edge
    If a rising edge is rejected, its maximum and minimum are rejected. A rising edge is rejected, if its amplitude ($AMPL = MAX - MIN$) is lower than amplitude threshold; or its duration is lower than a threshold that depends on the sampling rate; or its amplitude does not increase smoothly.

    (b) Estimation of the similarity of a rising edge to preceding and following rising edges accepted as valid
    Two rising edges are considered similar, if the amplitude of the lower-amplitude rising edge is greater than 50% of the amplitude of the higher-amplitude rising edge; and if the maximum of the lower-amplitude rising edge is between $\pm 60\%$ of the maximum of the higher-amplitude rising edge; and if the minimum of the lower-amplitude rising edge is between $\pm 60\%$ of the minimum of the higher-amplitude rising edge; and if the

duration of the shorter rising edge is greater then 33% of the duration of the longer rising edge. The valid rising edges are then categorized according to its characteristics for the following step. The categorization description is suppressed for brevity and can be found at [NI10].

(c) Verification of the current rising edge
The rising edges categorized on the previous step are considered valid edges of a pulse wave if they fulfill at least one of the decision rules presented on [NI10] and suppressed for brevity.

The validation process described here is important for discarding signals which are not representative of pulse waves. Providing a way of calculating the heart rate estimation only on valid pulse signals.

2.4 Technologies

Below are short descriptions of two of the main technologies that will be used during this research work.

2.4.1 Android SDK

Android SDK is the development kit for the *Android* platform. The *Android* platform is an open source, Linux-based operating system, primarily designed for touchscreen mobile devices, such as, smartphones.

Because of its open source code and permissive licensing, it allows the software to be freely modified and distributed. This have allowed *Android* to be the software of choice for technology companies who require a low-cost, customizable, and lightweight operating system for mobile devices and others.

Android has also become the world's most widely used smartphone platform with a worldwide smartphone market share of 75% during the third quarter of 2012 [IDC13].

Android consists of a kernel based on Linux kernel with middleware, libraries and APIs written in C. Applications, usually, run on an application framework which includes Java-compatible libraries based on *Apache Harmony*, an open source, free Java implementation. *Java bytecode* is then translated to run on the *Dalvik virtual machine*.

Porting existing Linux application or libraries to *Android* is difficult due to the lack of a native *X Window System* and lack of support for *GNU* libraries. Support for simple C and SDL application is possible, though, by the usage of *JNI*, a programming framework that allows Java code to call and be called by libraries written in C/C++.

2.4.2 OpenCV – Computer Vision Library

OpenCV is a library of programming functions mainly aimed at real-time image processing. To support these, it also includes a statistical machine learning library. Moreover, it is a cross-platform and open source library that is free to use and modify under the BSD license.

> "OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products." [Its13]

OpenCV is written in C/C++. There are binding for other languages, such as, Python, Java, and even Android. However, Java and Android implementation is recent and lacks features and stability.

2.5 Chapter summary

This chapter starts by describing the concept behind the extraction of cardiac pulse is possible from a person's face captured through a simple video or webcam.

It then presents several possible post-processing methods for to improve the extraction of the actual pulse signal. These methods include:

- *Independent Component Analysis*, a method capable of uncovering independent signals from a set of observations that are composed of linear mixtures of the underlying sources;
- *Eulerian Video Magnification*, a method inspired by the *Eulerian* perspective that exaggerates color variations by analyzing how each pixel value changes over time;
- *Detrend*, a method which removes small trends from an input signal without distorting its amplitude.

Then algorithms for obtaining the actual beats per minutes of the heart rate from the signal are described:

- *Power spectrum*, a set of equations capable of finding the frequency of a signal using the *Fourier transform*;
- *Pulse wave detection*, an algorithm for detecting and validating rising edges from a pulse signal.

Finally, important technologies for the work are described and explored:

- *Android*, a Linux-based operating system, primarily designed for touchscreen mobile devices;
- *OpenCV*, a *Computer Vision* library of programming functions mainly aimed at real-time image processing.

Chapter 3

Problem description

This chapter provides a detailed description of the problem addressed, defining its scope and dividing it in smaller problems.

Section 3.1 describes the main objective of the work which consists of an implementation a video magnification method based on the Eulerian perspective capable of running on a mobile device.

Then, section 3.2 provides a description of a simple application of the Eulerian Video Magnification method.

3.1 Android-based implementation of Eulerian Video Magnification

Fraunhofer Portugal is interested in testing the feasibility of implementing an Eulerian Video Magnification-based method on a mobile device with the Android platform.

As stated on the previous chapters, the Eulerian Video Magnification method is capable of magnifying small motion and amplifying color variation which may be invisible to the naked eye. Examples of the method application include: estimation of a person's heart rate from the variation of its face's color; respiratory rate from a person's chest movements; and even, detect asymmetry in facial blood flow, which may be a symptom of arterial problems.

The benefits of the Eulerian perspective is its low requirements for computational resources and algorithm complexity, in comparison to other attempts which rely on accurate motion estimation [LTF+05]. However, the existing limits of computational power on mobile devices may not allow the Eulerian Video Magnification method to execute in real-time.

The main project's goal is to develop a lightweight, real-time Eulerian Video Magnification-based method capable of executing on a mobile device. Which will require performance optimizations and trade-offs will have to taken into account.

3.2 Vital signs monitoring

As an objective to demonstrate that the Eulerian Video Magnification-based method developed is working as expected, the creation of an Android application which estimates a person's heart rate in real-time using the device's camera was pursued.

This goal requires comprehension of the photo-plethysmography concept, extraction of a frequency from a signal, and recognition / validation of a signal as a cardiac pulse.

The application will then need to be tested in order to verify its estimations. The test will be achieved by comparing results from a sphygmomanometer and other existing application [Tec13, Phi13] which use different methods to estimate a person's heart rate.

3.3 Chapter summary

In this chapter, a more detailed description of the problem and its scope is presented.

It describes the main goal and motivation for developing a lightweight, real-time Eulerian Video Magnification-based method for the Android platform.

Moreover, the goal of creating an Android application for heart rate monitoring is explained. Which serves for testing the developed method and demonstrating what can be achieved with the implemented Eulerian Video Magnification method.

Chapter 4

Implementation details

This chapter provides the implementation details of the work developed in order to create an Android application for estimating a person's heart rate based on the Eulerian Video Magnification method.

Section 4.1 provides an overview of the overall algorithm implemented.

Then, various implementations of the Eulerian Video Magnification method are described on section 4.2.

The face detection, signal validations, and heart rate estimation are also detailed on sections 4.3, 4.4, and 4.5, respectively.

Finally, section 4.6 details the interactions between the Android platform and the implemented library.

4.1 Overview

In order to create an Android application capable of estimating a person's heart rate, a desktop application was developed because of its faster implementation speed, and easier testing. The overall algorithm was divided into several steps, illustrated in figure 4.1, which, later, was extracted into a library, named Pulse, to be integrated into an Android application, also named Pulse. The language used to implement the desktop application and library was C/C++. In addition, for the image processing operations, the computer vision library, OpenCV, was used.

A short description of the overall algorithm's steps on figure 4.1 and application workflow is as follows:

1. *Original frame*, read frame from device's webcam;

2. *Face detector*, detect faces in current frame and match with previously detected faces in order to track multiple faces. Each face information is then fed into the following steps:

   (a) *Eulerian Video Magnification*, magnify detected face's rectangle;

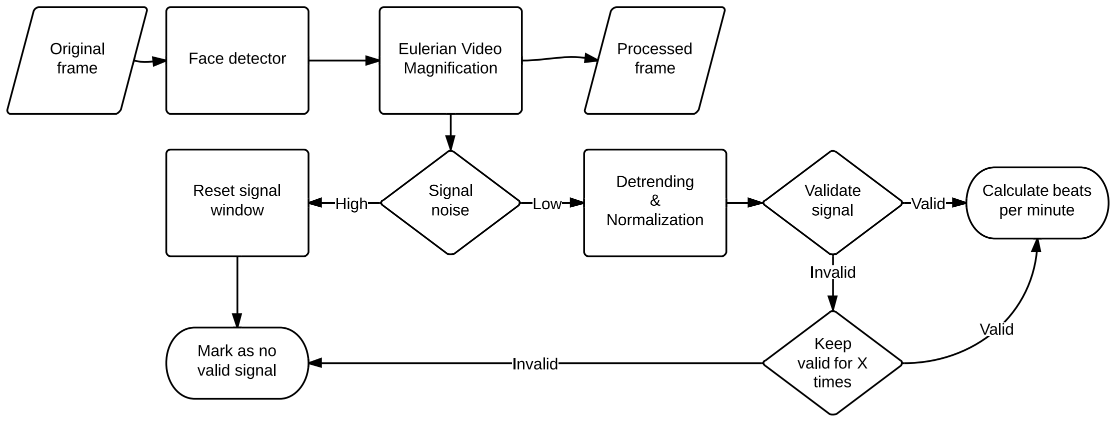


Figure 4.1: Overview of the implemented algorithm to obtain the heart rate of a person from a webcam or video using the Eulerian Video Magnification method.

(b) *Signal noise*, verify if the signal extracted from the current face is too noisy. If so, that face's signal is reset and marked as not valid;

(c) *Detrending & Normalization*, if the current face's signal is not too noisy, then detrend and normalize the signal in order to facilitate further operations with the signal;

(d) *Validate signal*, the face's signal is then validated by verifying its shape and timing, in a similar but simpler manner as found in [NI10]. If the signal is given as invalid, it is kept as valid for a couple of time, because the validation algorithm may miss some peaks;

(e) *Calculate beats per minute*, if the current face's signal is valid, it is then used to estimate the person's heart rate;

3. *Processed frame*, the resulting frame with each magnified face rectangle added back to the original frame.

Below, there are more detailed information and descriptions of the algorithm's steps.

4.2 Eulerian Video Magnification implementations

This section presents the details of several different implementations of the Eulerian Video Magnification method.

The first implementations, described on sections 4.2.1, 4.2.2 and 4.2.3, were developed in Java to facilitate the integration into the Android application. However, the OpenCV Java binding was still in its early stages which end up creating difficulties for the development. Thus, the final implementation, on section 4.2.4, was implemented in C/C++, which also reduces the number of JNI calls from the Android JVM and increases the application performance.

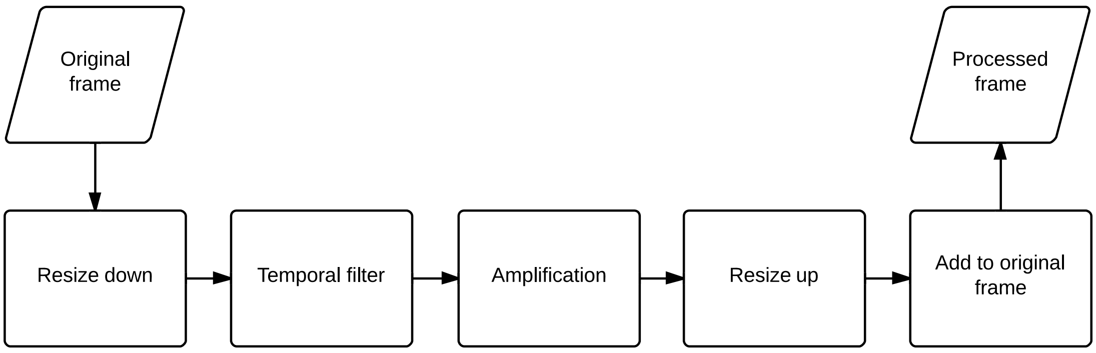


Figure 4.2: Overview of the Eulerian Video Magnification method steps.

The purpose of implementing multiple variants of the method was to study how the method worked and select which spatial and temporal filters would better fit the application goal: amplify color variation in real-time.

Figure 4.2 shows generic steps of the method which will be detailed on each of the following sections. The final step, *add to original frame*, however, remains the same in all implementations. Which is when the magnified values are added back to the original frame in order to obtain the processed frame.

4.2.1 EvmGdownIdeal

This was the first implementation, thus, its goal was to understand how the method worked, and match the implementation provided, in MATLAB, by [WRS+12]. In addition, real-time support was implemented by using a sliding window of 30 frames.

Resize down

This step applies a spatial filter by calculating a level of the Gaussian pyramid. This is achieved by looping to the desired level where the input to the next loop is the result from the previous loop, starting with the original frame. A Gaussian pyramid level is calculated by, first, convolving the input frame with the kernel, K:

$$K = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{4.1}$$

and then, downsampling the frame by rejecting even rows and columns.

Temporal filter

It was used an ideal bandpass filter to remove any amplification of undesired frequency from the color variation of each pixel. To construct this ideal filter, the Fourier transform was calculated for each pixel over the sliding window of 30 frames. Then, frequencies below 45 and above 240 where set to zero, and the frame was rebuilt using the inverse Fourier transform.

Amplification

In this step, the result of the temporal filter is multiplied by an α value. Which results in the magnification of the color variation selected by the temporal filter.

Resize up

This step performs the inverse operation of the *resize down* step, where it upsamples the frame by inserting even rows and columns with zeros, and then, convolves the input frame with the same kernel multiplied by 4. However, when the original frame is not multiple of two, an additional resize operation as to be done in order for the upsampled frame to match the original frame's size.

4.2.2 EvmGdownIIR

This implementation is very similar to the one above, but uses a different temporal filter which does not require a sliding window of frames to support real-time results. The filter used was an IIR bandpass filter, which was constructed from the subtraction of two first-order lowpass IIR filters. Each lowpass filter is computed as follows:

$$L_n = L_{n-1} * (1 - \omega) + \omega * M \tag{4.2}$$

where M is the current frame, L is the lowpass filter accumulator for each frame, and ω is the cutoff frequency percentage.

The IIR temporal bandpass filter demonstrated similar results to the ideal temporal filter used on the first implementation, without the need for persisting a sliding window of frames, which simplifies solution and reduces the computational power required by the device.

4.2.3 EvmLpyrIIR

Using the same IIR temporal filter as above, this implementation uses a different spatial filter, which, instead of, computing a level of the Gaussian pyramid, it constructs the full Laplacian pyramid and then applies the temporal filter to each of its bands and each band is amplified differently.

Resize down

Figure 4.3 shows the steps to decompose and reconstruct an image for the purpose of building a Laplacian pyramid. The *original image* must be decomposed into two images, *blurred*

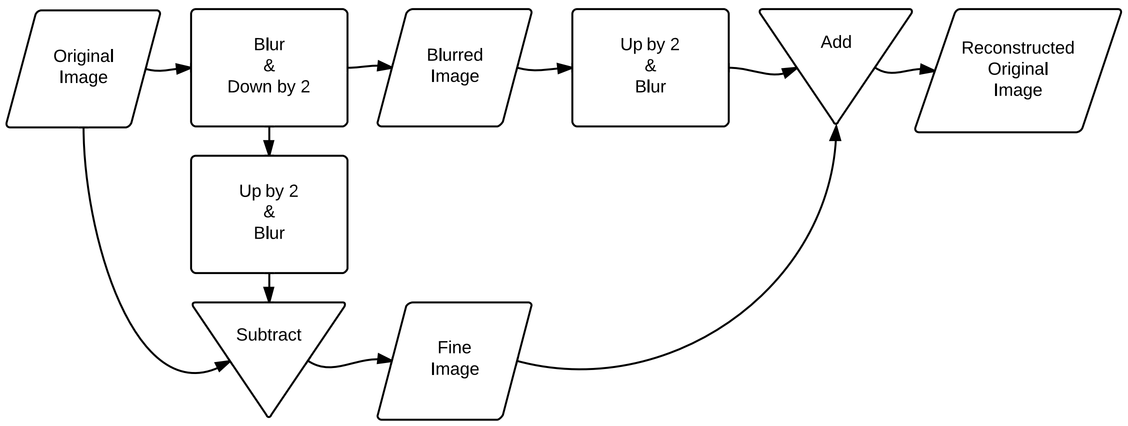


Figure 4.3: Overview of image deconstruction and reconstruction for building a Laplacian pyramid.

and *fine*, by applying any type of spatial lowpass filter and scaling the image down or up by 2. In this case, a Gaussian filter was applied as described on steps *resize down* and *resize up* of the first implementation. Further levels of the pyramid can be computed by decomposing the *blurred image* in the same manner.

Temporal filter

The temporal filter used is the IIR bandpass filter, as described above for the previous implementation, only this time it is applied to each level of the pyramid.

Amplification

The amplification method in this implementation is more complex than the one previously used. It is based on the implementation provided by [WRS+12]. It uses a different α value for each band of spatial frequencies, which corresponds to the Laplacian pyramid levels. The magnification value, α, follows the equation:

$$(1+\alpha)\delta(t) < \frac{\lambda}{8} \tag{4.3}$$

where $\delta(t)$ represents the displacement function and λ the spatial wavelength. Further details about this equation may be found on [WRS+12, Section 3.2].

Resize up

This step reconstructs the *original image* by iteratively reconstructing each *blurred image* until the now processed *original image* is reached.

This implementation demonstrated that by constructing a Laplacian pyramid for the spatial filter finer motion detail would be revealed, whereas the color variation, the property to be analyzed, was less evident.

4.2.4 Performance optimized EvmGdownIIR

Because the OpenCV Java binding was not complete at the time, the Java desktop implementations were not executing fast enough for real-time processing. Thus, a C/C++ implementation of the method that demonstrated better color variation was developed. In addition, after studying the performance of the application, detailed on section 5.1, the resize operations proved to be computationally expensive. Therefore, the *Resize down* and *Resize up* steps were modified to faster resize operations that did not alter the resulting image.

Resize down
This step was changed to a single resize operation using the OpenCV interpolation method named *area*, which produces a similar result to the one provided by using a Gaussian filter and downsampling the image. However, instead of, iteratively downsampling the frame multiple times, it is now a single resize to a predefined size.

Resize up
This step was also modified to a single resize operation using the linear interpolation method, which produces a similar result to the one used previously where the image was upsampled iteratively using a Gaussian filter.

4.3 Face detection

The *face detection* step uses the OpenCV object detector initially proposed on [VJ01] and improved on [LM02]. Which has been previously trained to detect faces.

Because object detectors are computationally expensive, in order to improve performance, a minimum size for the face detector was set to 40% of the frame width and height. In addition, since it was expected for the person to remain still during the reading, the face detector was set to execute only once a second.

Face tracking is simply done by matching the previous and newly detected faces to the closest one.

- If there are less newly detected faces than the previously detected ones, then match each new face to the nearest older face. Any older face that is not matched with a newly detected face is marked for deletion. However, it is only deleted if it fails to find a match on the next time the face detector executes. This measure allows the face detector to miss a detection of a face one time.

- Otherwise, if the number of newly detected faces is equal or more than the older faces, then match each older face to the nearest newly detected face. Any newly detected face that is not matched with an older face is then marked as a new face.

Another performance boost was achieved by only magnifying the each face rectangle instead of the whole frame. Because of this, the face rectangle must remain as still as possible, in order to introduce as few artifacts and noise to the Eulerian Video Magnification method. To achieve this, the position and size of the face's rectangle that is fed into the Eulerian Video Magnification method is interpolated between the previous and newly detected faces, if the distance between the two, d, is less than one third of the previous face's rectangle width, w:

$$d < \frac{w}{3} \tag{4.4}$$

And, the interpolation percentage, r, used is the ratio between these values:

$$r = \frac{3d}{w} \tag{4.5}$$

4.4 Signal validations

The signal validation has two phases. First, the *raw* signal, obtained by averaging the mean value of the green channel of a rectangle's face, is checked for noise, on step *signal noise* in figure 4.1. Then, on step *validate signal*, the shape and timing of the detrended and normalized signal is verified.

The raw signal noise is simply verified if the signal standard deviation is higher than 50% of the α, amplification factor.

Then, each of the following operations are applied to the raw signal:

1. *detrend*, as in [TRaK02], also shortly described on section 2.2.3;
2. *normalization*, the normalized signal, S', is obtained from the detrended signal, S, by subtracting the mean of the signal, $\bar{S}$, and divide it by the signal standard deviation, σ:

$$S' = \frac{S - \bar{S}}{\sigma} \tag{4.6}$$

3. *mean filter*, is done by convolving the signal 3 times with the kernel, K:

$$K = \frac{1}{5*5}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{4.7}$$

Finally, the shape and timing of signal is validated by a simplification of the fast pulse wave detection algorithm [NI10]. It identifies possible signal peaks by dividing the signal into consecutive 200ms time intervals and finding the absolute maximum. Some of these maximums are rejected: if they are a boundary value of that segment; or if they fall below a threshold, which is 60% of the mean of the amplitude of the possible identified peaks so far; or if the distance between two maximums is less than or equal to 200ms, then the lower maximum is rejected.

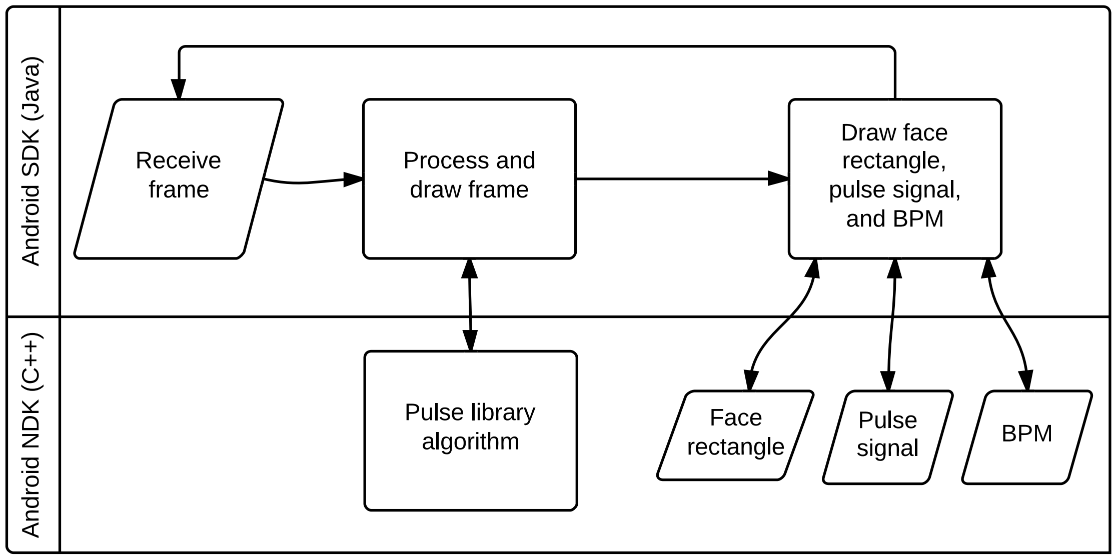


Figure 4.4: Overview of the interaction between the implemented Android application and library.

The signal is said to be valid if:

- it includes at least two peaks;
- the peak count is between the valid human range, between 40 and 240 bpm;
- the peaks' amplitude standard deviation is less than 0.5;
- the standard deviation of the time interval between peaks is less than 0.5;

To prevent invalidating the signal if a peak is missed or miscounted, the signal is kept as valid for the next 30 frames.

4.5 Heart rate estimation

Even though the simplification of the fast pulse wave detection algorithm presented on the previous section was capable of counting pulse wave peaks, it would frequently miscount at least by one peak. Thus, since the period analyzed was short, only 5 seconds at 20 frames per second, a miscounted peak would introduce a large error to the final value.

In order to obtain the value of *beats per minute* from the pulse signal, the method presented on section 2.3.1 was used every time the signal was marked as valid. To prevent big fluctuations, the value was averaged over the values obtained from the power spectrum method every 1 second.

4.6 Android integration

Due to performance, speed implementation, and easier testing, the algorithm's library was implemented in C/C++. To integrate with the Android SDK, the framework *Java Native Interface* was used. Because each JNI call from the Android SDK to the Android NDK introduces performance overhead, the fewer calls made to the library, the faster the application executes.

The Android application workflow and interaction with the Pulse library is illustrated on figure 4.4. The arrows that cross the modules' borders represent JNI methods' calls. As it is indicated, the whole algorithm is execute in one JNI call each frame in order to improve application performance. Three extra calls are made to obtain and display data to the user.

4.6.1 OpenCV for Android SDK

The OpenCV library is supported in the Android platform by installing a separate Android application which deals with selecting the appropriate OpenCV library for that device's architecture.

Because OpenCV for Android SDK was still in its early stage, it did not support a couple of features which had to be implemented. Such as, switching cameras at runtime, portrait mode, stretching frame to container, removing the alpha channel from the frame without introducing more operations.

4.7 Chapter summary

This chapter starts by describing an overview of the implemented algorithm's steps. The algorithm begins by detecting a person's face and magnifying it using the Eulerian Video Magnification method. Then, it extracts a possible pulse signal by averaging the green channel of a rectangle's face, which is validated, processed by detrending and normalization, and validated again, by verifying its shape and timing. Finally, the heart rate is estimated using the power spectrum technique to obtain a signals frequency.

The chapter then details various implementations of the Eulerian Video Magnification method which were implemented:

- *EvmGdownIdeal*, applies a spatial filter by computing a level of the Gaussian pyramid, and uses the *Fourier transform* to implement a temporal ideal bandpass filter;
- *EvmGdownIIR*, applies the same spatial filter as the previous, but uses a temporal IIR bandpass filter, which was constructed from the subtraction of two first-order lowpass IIR filters, which are more suitable for a real-time implementation;
- *EvmLpyrIIR*, applies the same temporal filter as above, but computes a full Laplacian pyramid for the spatial filter, where each level is amplified differently;
- *Performance optimized EvmGdownIIR*, is implemented in C/C++, while the others were implemented in Java, and the spatial filter is modified to single resize operations but the result is similar to computing a level of the Gaussian pyramid.

Face detection uses the OpenCV object detector module. In order to increase performance, the face detector only executes every one second. Moreover, only the face's rectangle is magnified, instead of the whole frame.

Two phases for validating the extracted signal exists. First, the raw signal standard deviation is checked against a threshold. Then, the raw signal is detrended and normalized, and the processed signal shape and timing is verified by detecting its peaks.

Finally, the algorithm is integrated into an Android application by using the JNI framework, so the Android SDK is capable of interacting with the native code.

Chapter 5

Results

This chapter...

5.1 Performance

In order to improve the algorithm and application performance, metrics were obtained using the *High performance C++ Profiler* [And13]. This profiler was chosen because of its performance claim and easy integration in the project.

The machine specifications on which all metrics were obtained are:

Machine MacBook Pro, 15-inch, Late 2008

Processor 2.53 GHz Inter Core 2 Duo

Memory 4 GB 1067 MHz DDR3

Operating System OS X Mountain Lion

Below are the dates of the various performance milestones accomplished. For each date there are two tables with the respective metrics that can be found on appendix A. The metrics are represented in *CPU cycles* obtained by using the instruction rdtsc [And13].

Mar 20, tables on page 34

These metrics match the initial implementation of the C/C++ version of the desktop application when performance profiling was integrated. The important functions found on the first table are:

- Pulse::onFrame, which is the function that detects a person face and estimates the heart rate for each face detected. Currently taking approximately 52 million cycles;

- EvmGdownIIR::onFrame, which is the function that implements the Eulerian Video Magnification method. Currently taking approximately 28 million cycles.

The operations taking more CPU cycles were easily identified by the second table on A.1, also represented on table 5.1:

Operation	Calls	Self MCycles	Self Avg
wait key	301	9892.1852 (32%)	32.8644
pyrUp	300	3472.5267 (11%)	11.5751
detect faces	301	2941.3343 (10%)	9.7719
imshow	301	2555.0751 (8%)	8.4886
capture	302	2268.6110 (7%)	7.5120
pyrDown	301	1988.4146 (7%)	6.6060
convert to 8 bit	300	1603.1581 (5%)	5.3439
void cv::detrend(...)	301	1538.8180 (5%)	5.1124
resize face box	301	1270.4278 (4%)	4.2207
resize and draw face box back to frame	301	1214.4998 (4%)	4.0349
add back to original frame	300	811.3654 (3%)	2.7046
convert to float	301	604.0527 (2%)	2.0068

Table 5.1: Operations taking more CPU cycles on the initial implementation of the C/C++ version of the desktop application when performance profiling was added.

Mar 26 tables on page 36

...

Apr 3 tables on page 38

...

May 23 tables on page 41

...

The tables generated by *High performance C++ Profiler* can be found on appendix A.

5.2 Heart rate comparison

5.3 Chapter summary

Chapter 6

Conclusions

6.1 Objective satisfaction

6.2 Future work

References

[And13] Andrew. High performance c++ profiling | andrew. http://floodyberry.wordpress.com/2009/10/07/high-performance-cplusplus-profiling/, March 2013.

[BA83] P. Burt and E. Adelson. The laplacian pyramid as a compact image code. *Communications, IEEE Transactions on*, 31(4):532–540, 1983.

[Com94] P. Comon. Independent component analysis, a new concept? *Signal processing*, 36(3):287–314, 1994.

[Gre97] EF Greneker. Radar sensing of heartbeat and respiration at a distance with applications of the technology. In *Radar 97 (Conf. Publ. No. 449)*, pages 150–154. IET, 1997.

[GSMP07] M. Garbey, N. Sun, A. Merla, and I. Pavlidis. Contact-free measurement of cardiac pulse based on the analysis of thermal imagery. *Biomedical Engineering, IEEE Transactions on*, 54(8):1418–1426, 2007.

[IDC13] IDC. Android marks fourth anniversary since launch with 75.0in third quarter, according to idc. http://www.idc.com/getdoc.jsp?containerId=prUS23771812, January 2013.

[Its13] OpenCV Developers Team: Itseez. About | opencv. http://opencv.org/about.html, January 2013.

[LFSK06] C. Liu, W.T. Freeman, R. Szeliski, and S.B. Kang. Noise estimation from a single image. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 901–908. IEEE, 2006.

[LM02] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, volume 1, pages I–900. IEEE, 2002.

[LOCS95] Daniel A Litvack, Tim F Oberlander, Laurel H Carney, and J Philip Saul. Time and frequency domain methods for heart rate variability analysis: a methodological comparison. *Psychophysiology*, 32(5):492–504, 1995.

[LTF$^+$05] C. Liu, A. Torralba, W.T. Freeman, F. Durand, and E.H. Adelson. Motion magnification. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 519–526. ACM, 2005.

[NI10] Bistra Nenova and Ivo Iliev. An automated algorithm for fast pulse wave detection. *International Journal Bioantomation*, 14(3):203–216, 2010.

[PB90] Stephen W Porges and Robert E Bohrer. The analysis of periodic processes in psychophysiological research. 1990.

[Phi13] Philips. Philips vital signs camera. http://www.vitalsignscamera.com, January 2013.

[PMP10] M.Z. Poh, D.J. McDuff, and R.W. Picard. Non-contact, automated cardiac pulse measurements using video imaging and blind source separation. *Optics Express*, 18(10):10762–10774, 2010.

[PMP11] M.Z. Poh, D.J. McDuff, and R.W. Picard. Advancements in noncontact, multiparameter physiological measurements using a webcam. *Biomedical Engineering, IEEE Transactions on*, 58(1):7–11, 2011.

[Por13] Fraunhofer Portugal. Fraunhofer portugal - about us. http://www.fraunhofer.pt, January 2013.

[Tec13] ViTrox Technologies. What's my heart rate. http://www.whatsmyheartrate.com, May 2013.

[TRaK02] Mika P Tarvainen, Perttu O Ranta-aho, and Pasi A Karjalainen. An advanced detrending method with application to hrv analysis. *Biomedical Engineering, IEEE Transactions on*, 49(2):172–175, 2002.

[UT93] S.S. Ulyanov and V.V. Tuchin. Pulse-wave monitoring by means of focused laser beams scattered by skin surface and membranes. In *OE/LASE'93: Optics, Electro-Optics, & Laser Applications in Science& Engineering*, pages 160–167. International Society for Optics and Photonics, 1993.

[VJ01] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.

[VSN08] Wim Verkruysse, Lars O Svaasand, and J Stuart Nelson. Remote plethysmographic imaging using ambient light. *Optics express*, 16(26):21434–21445, 2008.

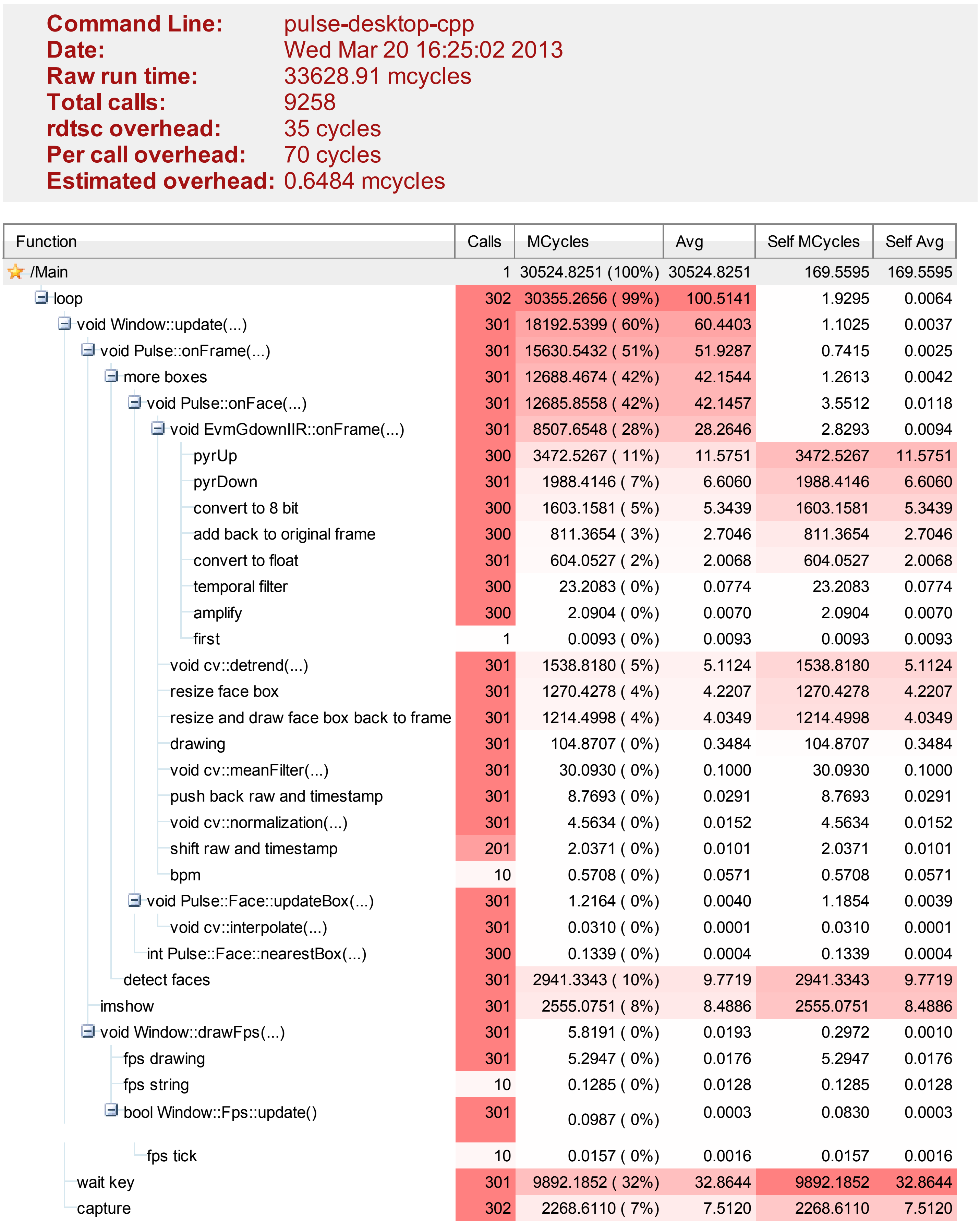
[WRS$^+$12] Hao-Yu Wu, Michael Rubinstein, Eugene Shih, John Guttag, Frédo Durand, and William T. Freeman. Eulerian video magnification for revealing subtle changes in the world. *ACM Trans. Graph. (Proceedings SIGGRAPH 2012)*, 31(4), 2012.

Appendix A

Performance metrics

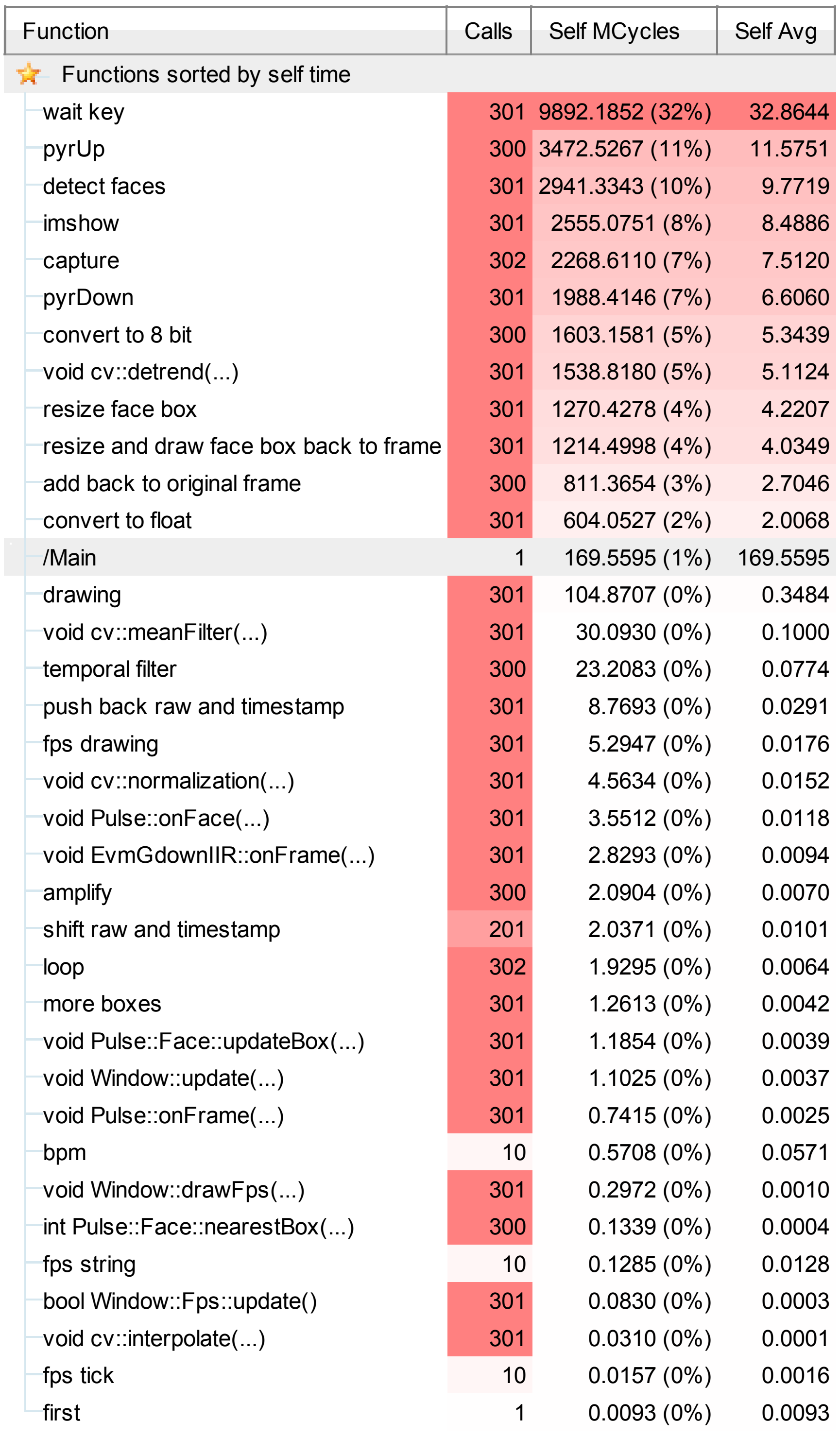
This appendix presents performance metrics for the desktop application implemented using the *High performance C++ Profiler*. For further details refer to section 5.1.

Each performance metric contains two tables. The first is ordered by function structure and total cycles the CPU spent on that function. The second table is ordered by self cycles, which represents the total cycles the CPU spent on that function minus the total cycles the CPU spent on the children functions.

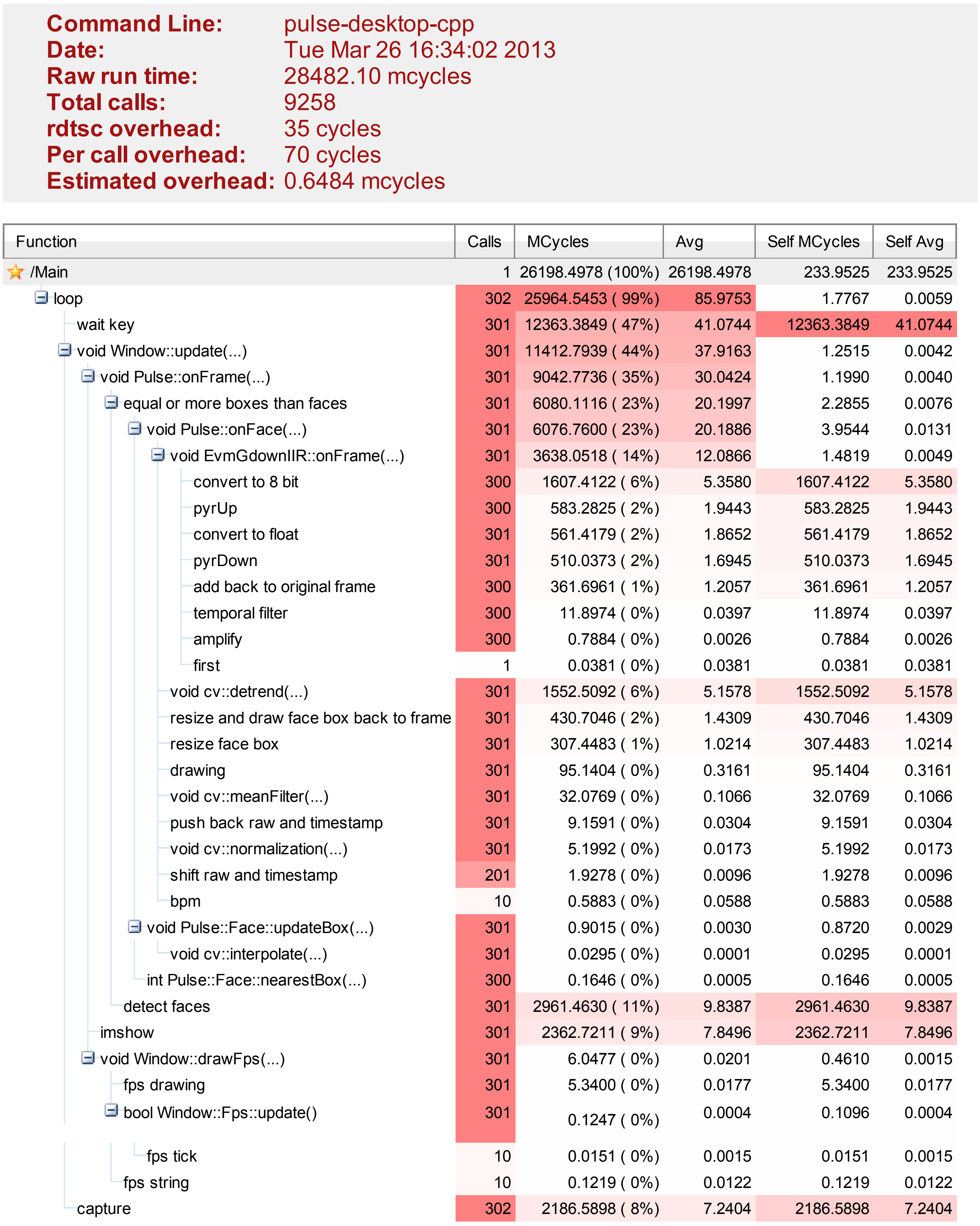


Command Line:	pulse-desktop-cpp
Date:	Wed Mar 20 16:25:02 2013
Raw run time:	33628.91 mcycles
Total calls:	9258
rdtsc overhead:	35 cycles
Per call overhead:	70 cycles
Estimated overhead:	0.6484 mcycles

Function	Calls	MCycles	Avg	Self MCycles	Self Avg
/Main	1	30524.8251 (100%)	30524.8251	169.5595	169.5595
loop	302	30355.2656 (99%)	100.5141	1.9295	0.0064
void Window::update(...)	301	18192.5399 (60%)	60.4403	1.1025	0.0037
void Pulse::onFrame(...)	301	15630.5432 (51%)	51.9287	0.7415	0.0025
more boxes	301	12688.4674 (42%)	42.1544	1.2613	0.0042
void Pulse::onFace(...)	301	12685.8558 (42%)	42.1457	3.5512	0.0118
void EvmGdownIIR::onFrame(...)	301	8507.6548 (28%)	28.2646	2.8293	0.0094
pyrUp	300	3472.5267 (11%)	11.5751	3472.5267	11.5751
pyrDown	301	1988.4146 (7%)	6.6060	1988.4146	6.6060
convert to 8 bit	300	1603.1581 (5%)	5.3439	1603.1581	5.3439
add back to original frame	300	811.3654 (3%)	2.7046	811.3654	2.7046
convert to float	301	604.0527 (2%)	2.0068	604.0527	2.0068
temporal filter	300	23.2083 (0%)	0.0774	23.2083	0.0774
amplify	300	2.0904 (0%)	0.0070	2.0904	0.0070
first	1	0.0093 (0%)	0.0093	0.0093	0.0093
void cv::detrend(...)	301	1538.8180 (5%)	5.1124	1538.8180	5.1124
resize face box	301	1270.4278 (4%)	4.2207	1270.4278	4.2207
resize and draw face box back to frame	301	1214.4998 (4%)	4.0349	1214.4998	4.0349
drawing	301	104.8707 (0%)	0.3484	104.8707	0.3484
void cv::meanFilter(...)	301	30.0930 (0%)	0.1000	30.0930	0.1000
push back raw and timestamp	301	8.7693 (0%)	0.0291	8.7693	0.0291
void cv::normalization(...)	301	4.5634 (0%)	0.0152	4.5634	0.0152
shift raw and timestamp	201	2.0371 (0%)	0.0101	2.0371	0.0101
bpm	10	0.5708 (0%)	0.0571	0.5708	0.0571
void Pulse::Face::updateBox(...)	301	1.2164 (0%)	0.0040	1.1854	0.0039
void cv::interpolate(...)	301	0.0310 (0%)	0.0001	0.0310	0.0001
int Pulse::Face::nearestBox(...)	300	0.1339 (0%)	0.0004	0.1339	0.0004
detect faces	301	2941.3343 (10%)	9.7719	2941.3343	9.7719
imshow	301	2555.0751 (8%)	8.4886	2555.0751	8.4886
void Window::drawFps(...)	301	5.8191 (0%)	0.0193	0.2972	0.0010
fps drawing	301	5.2947 (0%)	0.0176	5.2947	0.0176
fps string	10	0.1285 (0%)	0.0128	0.1285	0.0128
bool Window::Fps::update()	301	0.0987 (0%)	0.0003	0.0830	0.0003
fps tick	10	0.0157 (0%)	0.0016	0.0157	0.0016
wait key	301	9892.1852 (32%)	32.8644	9892.1852	32.8644
capture	302	2268.6110 (7%)	7.5120	2268.6110	7.5120

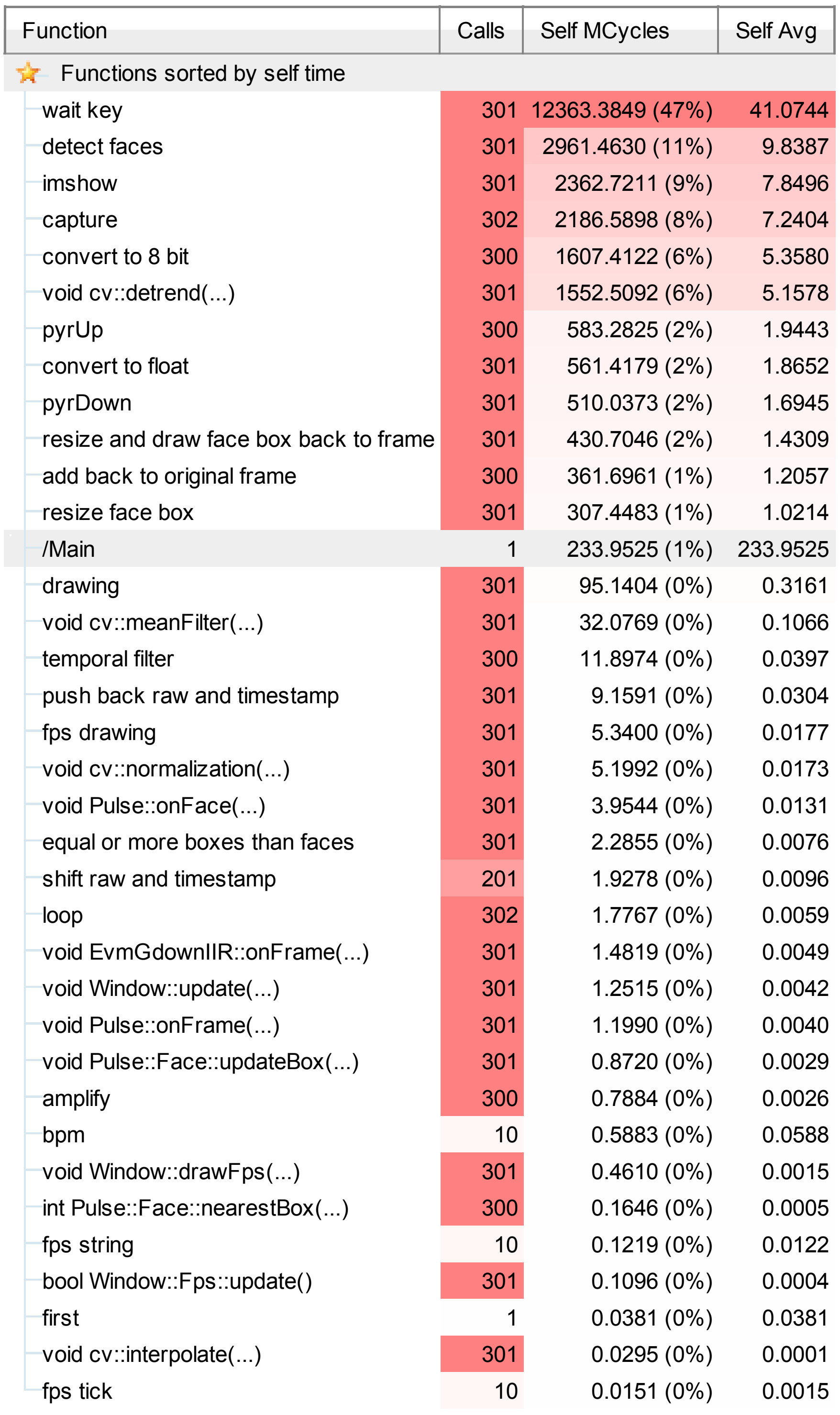


Function	Calls	Self MCycles	Self Avg
Functions sorted by self time			
wait key	301	9892.1852 (32%)	32.8644
pyrUp	300	3472.5267 (11%)	11.5751
detect faces	301	2941.3343 (10%)	9.7719
imshow	301	2555.0751 (8%)	8.4886
capture	302	2268.6110 (7%)	7.5120
pyrDown	301	1988.4146 (7%)	6.6060
convert to 8 bit	300	1603.1581 (5%)	5.3439
void cv::detrend(...)	301	1538.8180 (5%)	5.1124
resize face box	301	1270.4278 (4%)	4.2207
resize and draw face box back to frame	301	1214.4998 (4%)	4.0349
add back to original frame	300	811.3654 (3%)	2.7046
convert to float	301	604.0527 (2%)	2.0068
/Main	1	169.5595 (1%)	169.5595
drawing	301	104.8707 (0%)	0.3484
void cv::meanFilter(...)	301	30.0930 (0%)	0.1000
temporal filter	300	23.2083 (0%)	0.0774
push back raw and timestamp	301	8.7693 (0%)	0.0291
fps drawing	301	5.2947 (0%)	0.0176
void cv::normalization(...)	301	4.5634 (0%)	0.0152
void Pulse::onFace(...)	301	3.5512 (0%)	0.0118
void EvmGdownIIR::onFrame(...)	301	2.8293 (0%)	0.0094
amplify	300	2.0904 (0%)	0.0070
shift raw and timestamp	201	2.0371 (0%)	0.0101
loop	302	1.9295 (0%)	0.0064
more boxes	301	1.2613 (0%)	0.0042
void Pulse::Face::updateBox(...)	301	1.1854 (0%)	0.0039
void Window::update(...)	301	1.1025 (0%)	0.0037
void Pulse::onFrame(...)	301	0.7415 (0%)	0.0025
bpm	10	0.5708 (0%)	0.0571
void Window::drawFps(...)	301	0.2972 (0%)	0.0010
int Pulse::Face::nearestBox(...)	300	0.1339 (0%)	0.0004
fps string	10	0.1285 (0%)	0.0128
bool Window::Fps::update()	301	0.0830 (0%)	0.0003
void cv::interpolate(...)	301	0.0310 (0%)	0.0001
fps tick	10	0.0157 (0%)	0.0016
first	1	0.0093 (0%)	0.0093

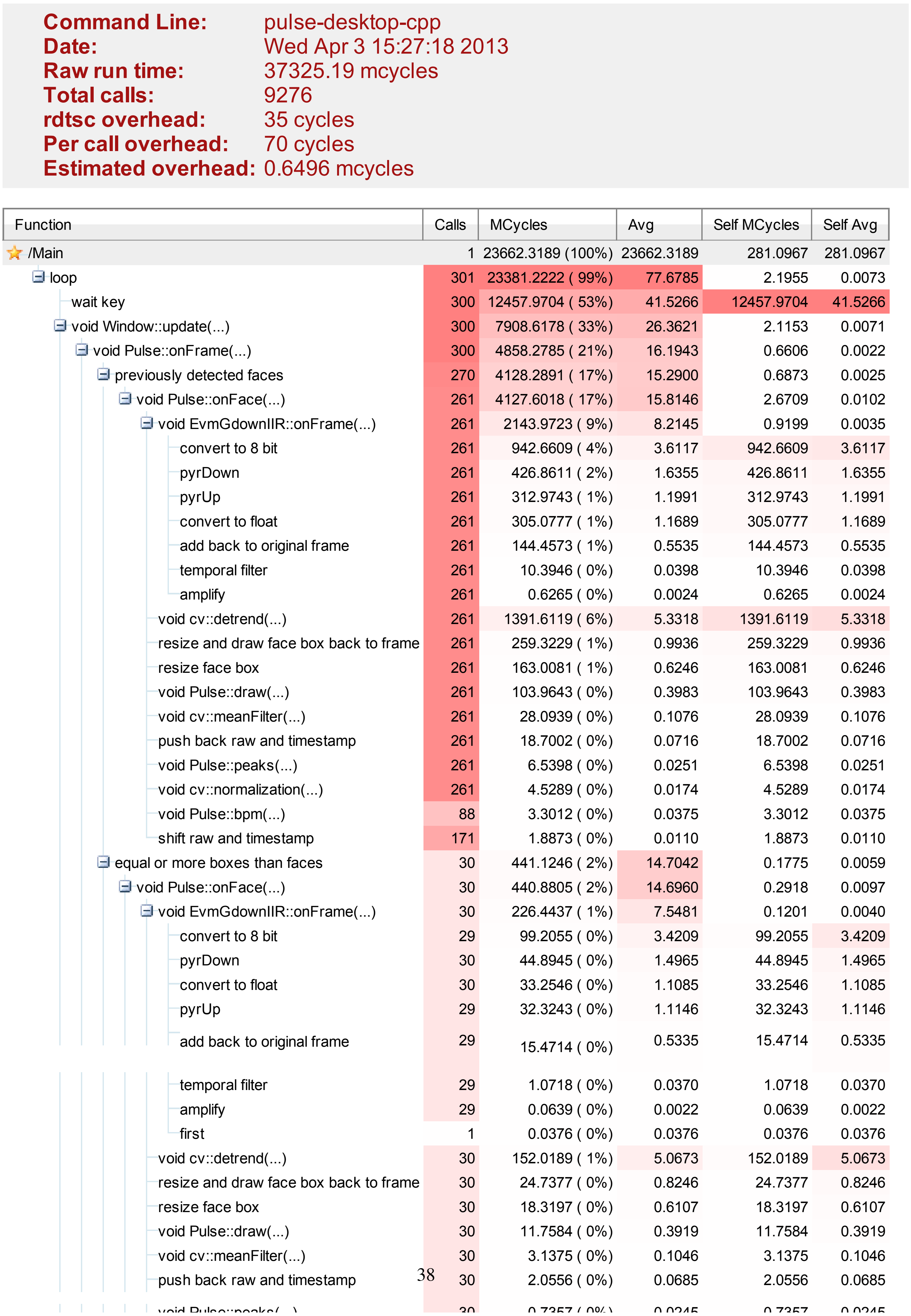


Command Line:	pulse-desktop-cpp
Date:	Tue Mar 26 16:34:02 2013
Raw run time:	28482.10 mcycles
Total calls:	9258
rdtsc overhead:	35 cycles
Per call overhead:	70 cycles
Estimated overhead:	0.6484 mcycles

Function	Calls	MCycles	Avg	Self MCycles	Self Avg
/Main	1	26198.4978 (100%)	26198.4978	233.9525	233.9525
loop	302	25964.5453 (99%)	85.9753	1.7767	0.0059
wait key	301	12363.3849 (47%)	41.0744	12363.3849	41.0744
void Window::update(...)	301	11412.7939 (44%)	37.9163	1.2515	0.0042
void Pulse::onFrame(...)	301	9042.7736 (35%)	30.0424	1.1990	0.0040
equal or more boxes than faces	301	6080.1116 (23%)	20.1997	2.2855	0.0076
void Pulse::onFace(...)	301	6076.7600 (23%)	20.1886	3.9544	0.0131
void EvmGdownIIR::onFrame(...)	301	3638.0518 (14%)	12.0866	1.4819	0.0049
convert to 8 bit	300	1607.4122 (6%)	5.3580	1607.4122	5.3580
pyrUp	300	583.2825 (2%)	1.9443	583.2825	1.9443
convert to float	301	561.4179 (2%)	1.8652	561.4179	1.8652
pyrDown	301	510.0373 (2%)	1.6945	510.0373	1.6945
add back to original frame	300	361.6961 (1%)	1.2057	361.6961	1.2057
temporal filter	300	11.8974 (0%)	0.0397	11.8974	0.0397
amplify	300	0.7884 (0%)	0.0026	0.7884	0.0026
first	1	0.0381 (0%)	0.0381	0.0381	0.0381
void cv::detrend(...)	301	1552.5092 (6%)	5.1578	1552.5092	5.1578
resize and draw face box back to frame	301	430.7046 (2%)	1.4309	430.7046	1.4309
resize face box	301	307.4483 (1%)	1.0214	307.4483	1.0214
drawing	301	95.1404 (0%)	0.3161	95.1404	0.3161
void cv::meanFilter(...)	301	32.0769 (0%)	0.1066	32.0769	0.1066
push back raw and timestamp	301	9.1591 (0%)	0.0304	9.1591	0.0304
void cv::normalization(...)	301	5.1992 (0%)	0.0173	5.1992	0.0173
shift raw and timestamp	201	1.9278 (0%)	0.0096	1.9278	0.0096
bpm	10	0.5883 (0%)	0.0588	0.5883	0.0588
void Pulse::Face::updateBox(...)	301	0.9015 (0%)	0.0030	0.8720	0.0029
void cv::interpolate(...)	301	0.0295 (0%)	0.0001	0.0295	0.0001
int Pulse::Face::nearestBox(...)	300	0.1646 (0%)	0.0005	0.1646	0.0005
detect faces	301	2961.4630 (11%)	9.8387	2961.4630	9.8387
imshow	301	2362.7211 (9%)	7.8496	2362.7211	7.8496
void Window::drawFps(...)	301	6.0477 (0%)	0.0201	0.4610	0.0015
fps drawing	301	5.3400 (0%)	0.0177	5.3400	0.0177
bool Window::Fps::update()	301	0.1247 (0%)	0.0004	0.1096	0.0004
fps tick	10	0.0151 (0%)	0.0015	0.0151	0.0015
fps string	10	0.1219 (0%)	0.0122	0.1219	0.0122
capture	302	2186.5898 (8%)	7.2404	2186.5898	7.2404

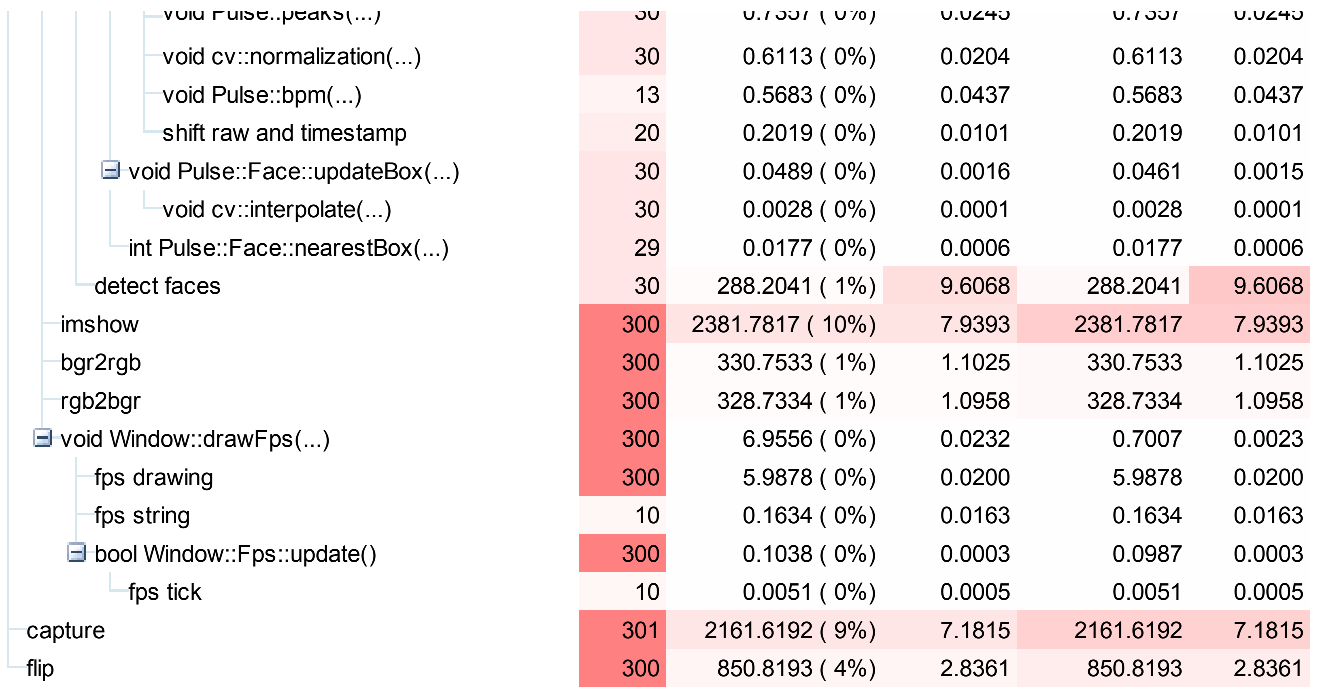


Function	Calls	Self MCycles	Self Avg
Functions sorted by self time			
wait key	301	12363.3849 (47%)	41.0744
detect faces	301	2961.4630 (11%)	9.8387
imshow	301	2362.7211 (9%)	7.8496
capture	302	2186.5898 (8%)	7.2404
convert to 8 bit	300	1607.4122 (6%)	5.3580
void cv::detrend(...)	301	1552.5092 (6%)	5.1578
pyrUp	300	583.2825 (2%)	1.9443
convert to float	301	561.4179 (2%)	1.8652
pyrDown	301	510.0373 (2%)	1.6945
resize and draw face box back to frame	301	430.7046 (2%)	1.4309
add back to original frame	300	361.6961 (1%)	1.2057
resize face box	301	307.4483 (1%)	1.0214
/Main	1	233.9525 (1%)	233.9525
drawing	301	95.1404 (0%)	0.3161
void cv::meanFilter(...)	301	32.0769 (0%)	0.1066
temporal filter	300	11.8974 (0%)	0.0397
push back raw and timestamp	301	9.1591 (0%)	0.0304
fps drawing	301	5.3400 (0%)	0.0177
void cv::normalization(...)	301	5.1992 (0%)	0.0173
void Pulse::onFace(...)	301	3.9544 (0%)	0.0131
equal or more boxes than faces	301	2.2855 (0%)	0.0076
shift raw and timestamp	201	1.9278 (0%)	0.0096
loop	302	1.7767 (0%)	0.0059
void EvmGdownIIR::onFrame(...)	301	1.4819 (0%)	0.0049
void Window::update(...)	301	1.2515 (0%)	0.0042
void Pulse::onFrame(...)	301	1.1990 (0%)	0.0040
void Pulse::Face::updateBox(...)	301	0.8720 (0%)	0.0029
amplify	300	0.7884 (0%)	0.0026
bpm	10	0.5883 (0%)	0.0588
void Window::drawFps(...)	301	0.4610 (0%)	0.0015
int Pulse::Face::nearestBox(...)	300	0.1646 (0%)	0.0005
fps string	10	0.1219 (0%)	0.0122
bool Window::Fps::update()	301	0.1096 (0%)	0.0004
first	1	0.0381 (0%)	0.0381
void cv::interpolate(...)	301	0.0295 (0%)	0.0001
fps tick	10	0.0151 (0%)	0.0015

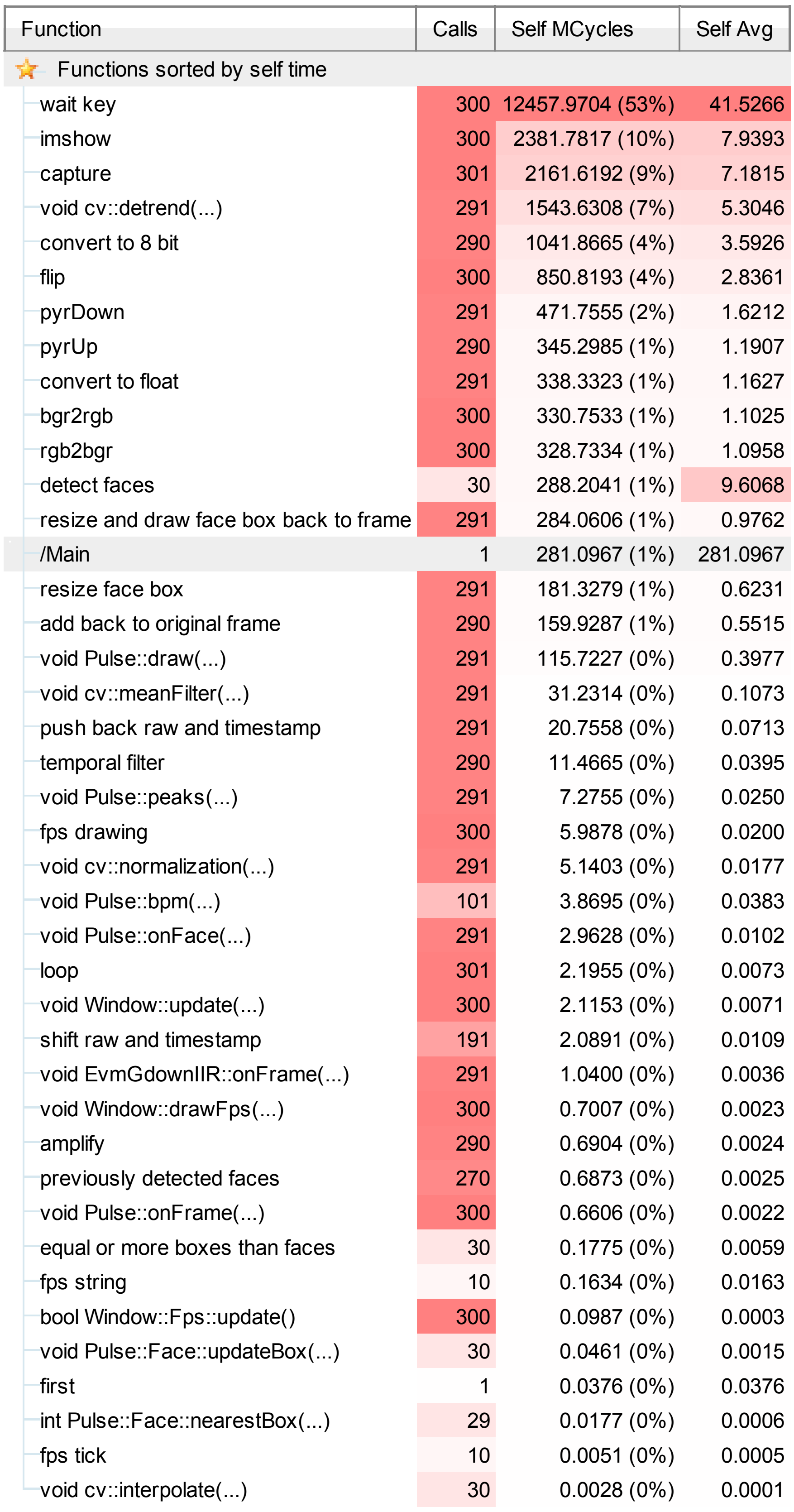


Command Line: pulse-desktop-cpp
Date: Wed Apr 3 15:27:18 2013
Raw run time: 37325.19 mcycles
Total calls: 9276
rdtsc overhead: 35 cycles
Per call overhead: 70 cycles
Estimated overhead: 0.6496 mcycles

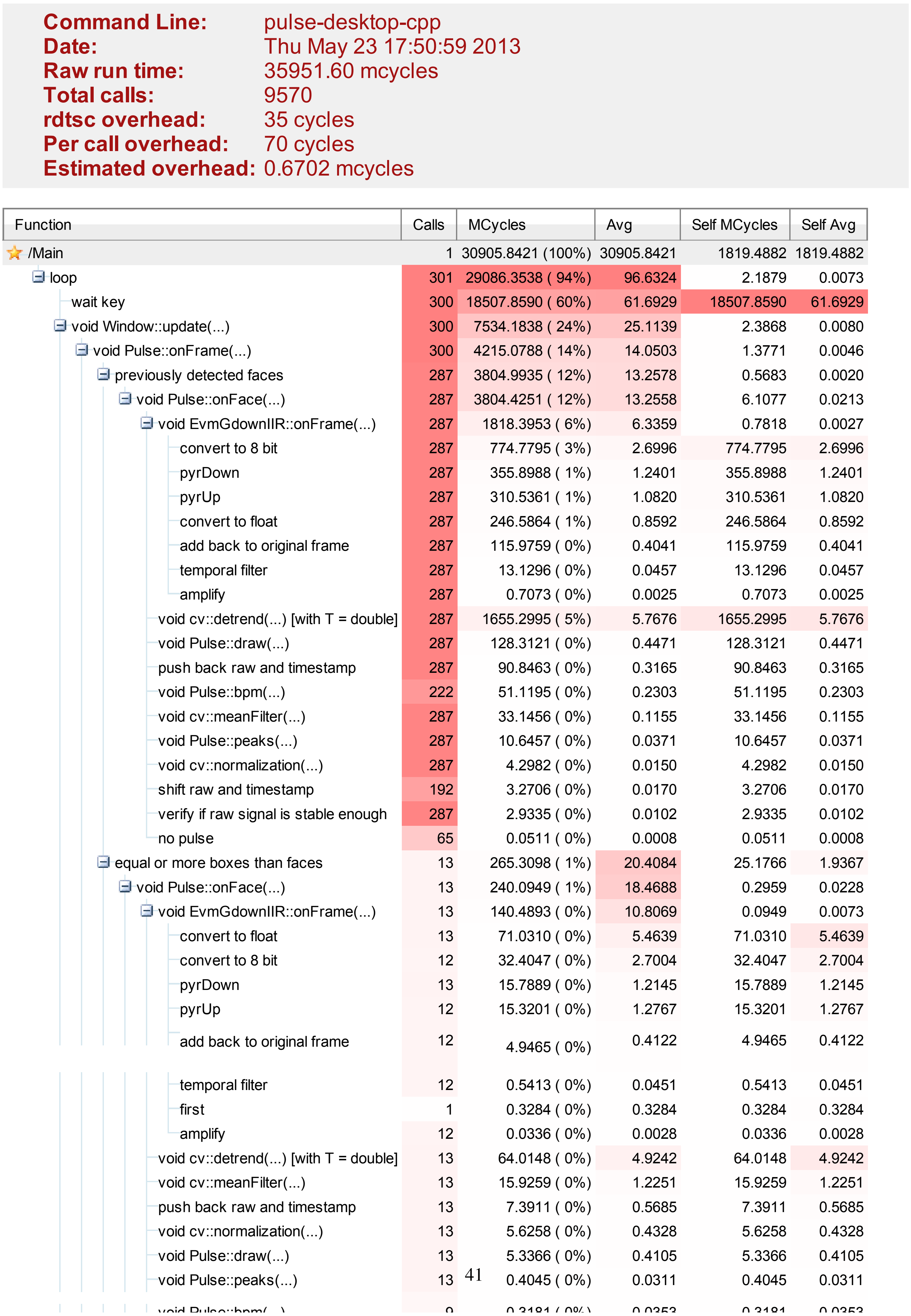
Function	Calls	MCycles	Avg	Self MCycles	Self Avg
⭐ /Main	1	23662.3189 (100%)	23662.3189	281.0967	281.0967
⊟ loop	301	23381.2222 (99%)	77.6785	2.1955	0.0073
wait key	300	12457.9704 (53%)	41.5266	12457.9704	41.5266
⊟ void Window::update(...)	300	7908.6178 (33%)	26.3621	2.1153	0.0071
⊟ void Pulse::onFrame(...)	300	4858.2785 (21%)	16.1943	0.6606	0.0022
⊟ previously detected faces	270	4128.2891 (17%)	15.2900	0.6873	0.0025
⊟ void Pulse::onFace(...)	261	4127.6018 (17%)	15.8146	2.6709	0.0102
⊟ void EvmGdownIIR::onFrame(...)	261	2143.9723 (9%)	8.2145	0.9199	0.0035
convert to 8 bit	261	942.6609 (4%)	3.6117	942.6609	3.6117
pyrDown	261	426.8611 (2%)	1.6355	426.8611	1.6355
pyrUp	261	312.9743 (1%)	1.1991	312.9743	1.1991
convert to float	261	305.0777 (1%)	1.1689	305.0777	1.1689
add back to original frame	261	144.4573 (1%)	0.5535	144.4573	0.5535
temporal filter	261	10.3946 (0%)	0.0398	10.3946	0.0398
amplify	261	0.6265 (0%)	0.0024	0.6265	0.0024
void cv::detrend(...)	261	1391.6119 (6%)	5.3318	1391.6119	5.3318
resize and draw face box back to frame	261	259.3229 (1%)	0.9936	259.3229	0.9936
resize face box	261	163.0081 (1%)	0.6246	163.0081	0.6246
void Pulse::draw(...)	261	103.9643 (0%)	0.3983	103.9643	0.3983
void cv::meanFilter(...)	261	28.0939 (0%)	0.1076	28.0939	0.1076
push back raw and timestamp	261	18.7002 (0%)	0.0716	18.7002	0.0716
void Pulse::peaks(...)	261	6.5398 (0%)	0.0251	6.5398	0.0251
void cv::normalization(...)	261	4.5289 (0%)	0.0174	4.5289	0.0174
void Pulse::bpm(...)	88	3.3012 (0%)	0.0375	3.3012	0.0375
shift raw and timestamp	171	1.8873 (0%)	0.0110	1.8873	0.0110
⊟ equal or more boxes than faces	30	441.1246 (2%)	14.7042	0.1775	0.0059
⊟ void Pulse::onFace(...)	30	440.8805 (2%)	14.6960	0.2918	0.0097
⊟ void EvmGdownIIR::onFrame(...)	30	226.4437 (1%)	7.5481	0.1201	0.0040
convert to 8 bit	29	99.2055 (0%)	3.4209	99.2055	3.4209
pyrDown	30	44.8945 (0%)	1.4965	44.8945	1.4965
convert to float	30	33.2546 (0%)	1.1085	33.2546	1.1085
pyrUp	29	32.3243 (0%)	1.1146	32.3243	1.1146
add back to original frame	29	15.4714 (0%)	0.5335	15.4714	0.5335
temporal filter	29	1.0718 (0%)	0.0370	1.0718	0.0370
amplify	29	0.0639 (0%)	0.0022	0.0639	0.0022
first	1	0.0376 (0%)	0.0376	0.0376	0.0376
void cv::detrend(...)	30	152.0189 (1%)	5.0673	152.0189	5.0673
resize and draw face box back to frame	30	24.7377 (0%)	0.8246	24.7377	0.8246
resize face box	30	18.3197 (0%)	0.6107	18.3197	0.6107
void Pulse::draw(...)	30	11.7584 (0%)	0.3919	11.7584	0.3919
void cv::meanFilter(...)	30	3.1375 (0%)	0.1046	3.1375	0.1046
push back raw and timestamp	30	2.0556 (0%)	0.0685	2.0556	0.0685
void Pulse::peaks(...)	30	0.7357 (0%)	0.0245	0.7357	0.0245



void Pulse::peaks(...)	30	0.7357 (0%)	0.0245	0.7357	0.0245
void cv::normalization(...)	30	0.6113 (0%)	0.0204	0.6113	0.0204
void Pulse::bpm(...)	13	0.5683 (0%)	0.0437	0.5683	0.0437
shift raw and timestamp	20	0.2019 (0%)	0.0101	0.2019	0.0101
void Pulse::Face::updateBox(...)	30	0.0489 (0%)	0.0016	0.0461	0.0015
void cv::interpolate(...)	30	0.0028 (0%)	0.0001	0.0028	0.0001
int Pulse::Face::nearestBox(...)	29	0.0177 (0%)	0.0006	0.0177	0.0006
detect faces	30	288.2041 (1%)	9.6068	288.2041	9.6068
imshow	300	2381.7817 (10%)	7.9393	2381.7817	7.9393
bgr2rgb	300	330.7533 (1%)	1.1025	330.7533	1.1025
rgb2bgr	300	328.7334 (1%)	1.0958	328.7334	1.0958
void Window::drawFps(...)	300	6.9556 (0%)	0.0232	0.7007	0.0023
fps drawing	300	5.9878 (0%)	0.0200	5.9878	0.0200
fps string	10	0.1634 (0%)	0.0163	0.1634	0.0163
bool Window::Fps::update()	300	0.1038 (0%)	0.0003	0.0987	0.0003
fps tick	10	0.0051 (0%)	0.0005	0.0051	0.0005
capture	301	2161.6192 (9%)	7.1815	2161.6192	7.1815
flip	300	850.8193 (4%)	2.8361	850.8193	2.8361

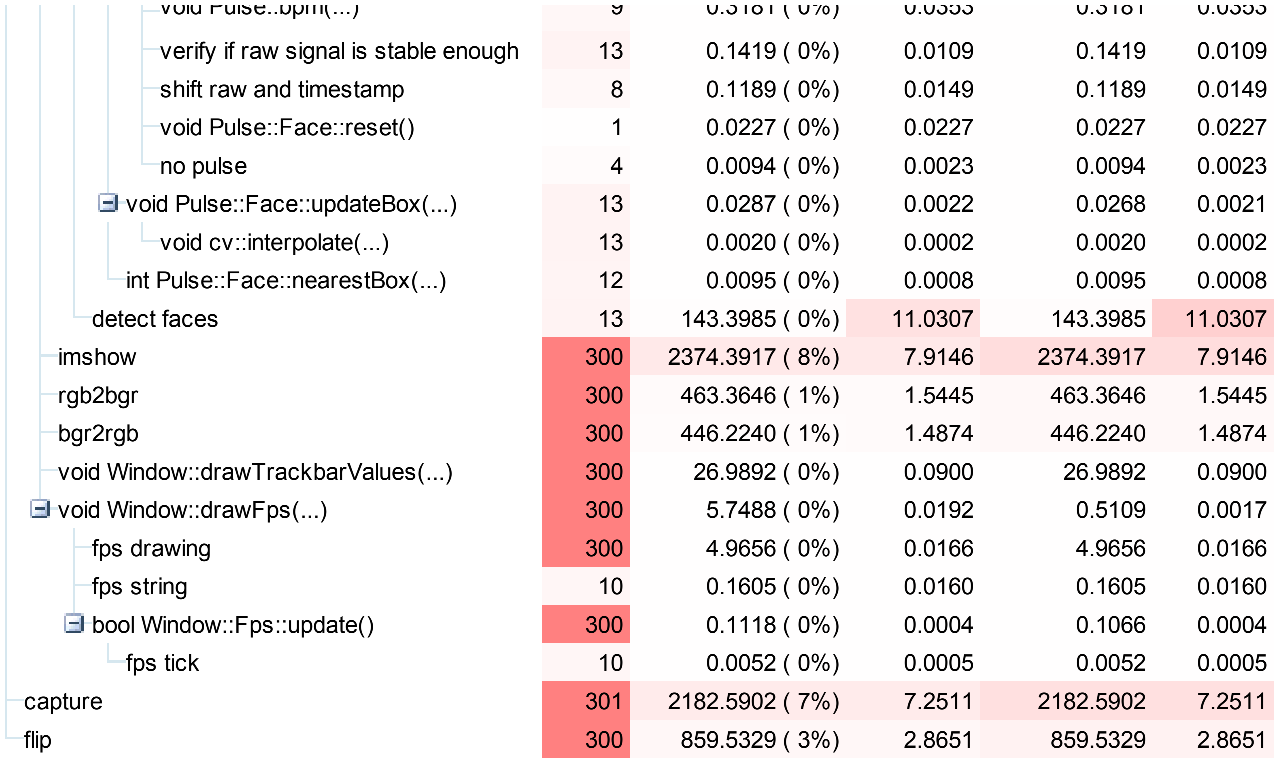


Function	Calls	Self MCycles	Self Avg
Functions sorted by self time			
wait key	300	12457.9704 (53%)	41.5266
imshow	300	2381.7817 (10%)	7.9393
capture	301	2161.6192 (9%)	7.1815
void cv::detrend(...)	291	1543.6308 (7%)	5.3046
convert to 8 bit	290	1041.8665 (4%)	3.5926
flip	300	850.8193 (4%)	2.8361
pyrDown	291	471.7555 (2%)	1.6212
pyrUp	290	345.2985 (1%)	1.1907
convert to float	291	338.3323 (1%)	1.1627
bgr2rgb	300	330.7533 (1%)	1.1025
rgb2bgr	300	328.7334 (1%)	1.0958
detect faces	30	288.2041 (1%)	9.6068
resize and draw face box back to frame	291	284.0606 (1%)	0.9762
/Main	1	281.0967 (1%)	281.0967
resize face box	291	181.3279 (1%)	0.6231
add back to original frame	290	159.9287 (1%)	0.5515
void Pulse::draw(...)	291	115.7227 (0%)	0.3977
void cv::meanFilter(...)	291	31.2314 (0%)	0.1073
push back raw and timestamp	291	20.7558 (0%)	0.0713
temporal filter	290	11.4665 (0%)	0.0395
void Pulse::peaks(...)	291	7.2755 (0%)	0.0250
fps drawing	300	5.9878 (0%)	0.0200
void cv::normalization(...)	291	5.1403 (0%)	0.0177
void Pulse::bpm(...)	101	3.8695 (0%)	0.0383
void Pulse::onFace(...)	291	2.9628 (0%)	0.0102
loop	301	2.1955 (0%)	0.0073
void Window::update(...)	300	2.1153 (0%)	0.0071
shift raw and timestamp	191	2.0891 (0%)	0.0109
void EvmGdownIIR::onFrame(...)	291	1.0400 (0%)	0.0036
void Window::drawFps(...)	300	0.7007 (0%)	0.0023
amplify	290	0.6904 (0%)	0.0024
previously detected faces	270	0.6873 (0%)	0.0025
void Pulse::onFrame(...)	300	0.6606 (0%)	0.0022
equal or more boxes than faces	30	0.1775 (0%)	0.0059
fps string	10	0.1634 (0%)	0.0163
bool Window::Fps::update()	300	0.0987 (0%)	0.0003
void Pulse::Face::updateBox(...)	30	0.0461 (0%)	0.0015
first	1	0.0376 (0%)	0.0376
int Pulse::Face::nearestBox(...)	29	0.0177 (0%)	0.0006
fps tick	10	0.0051 (0%)	0.0005
void cv::interpolate(...)	30	0.0028 (0%)	0.0001

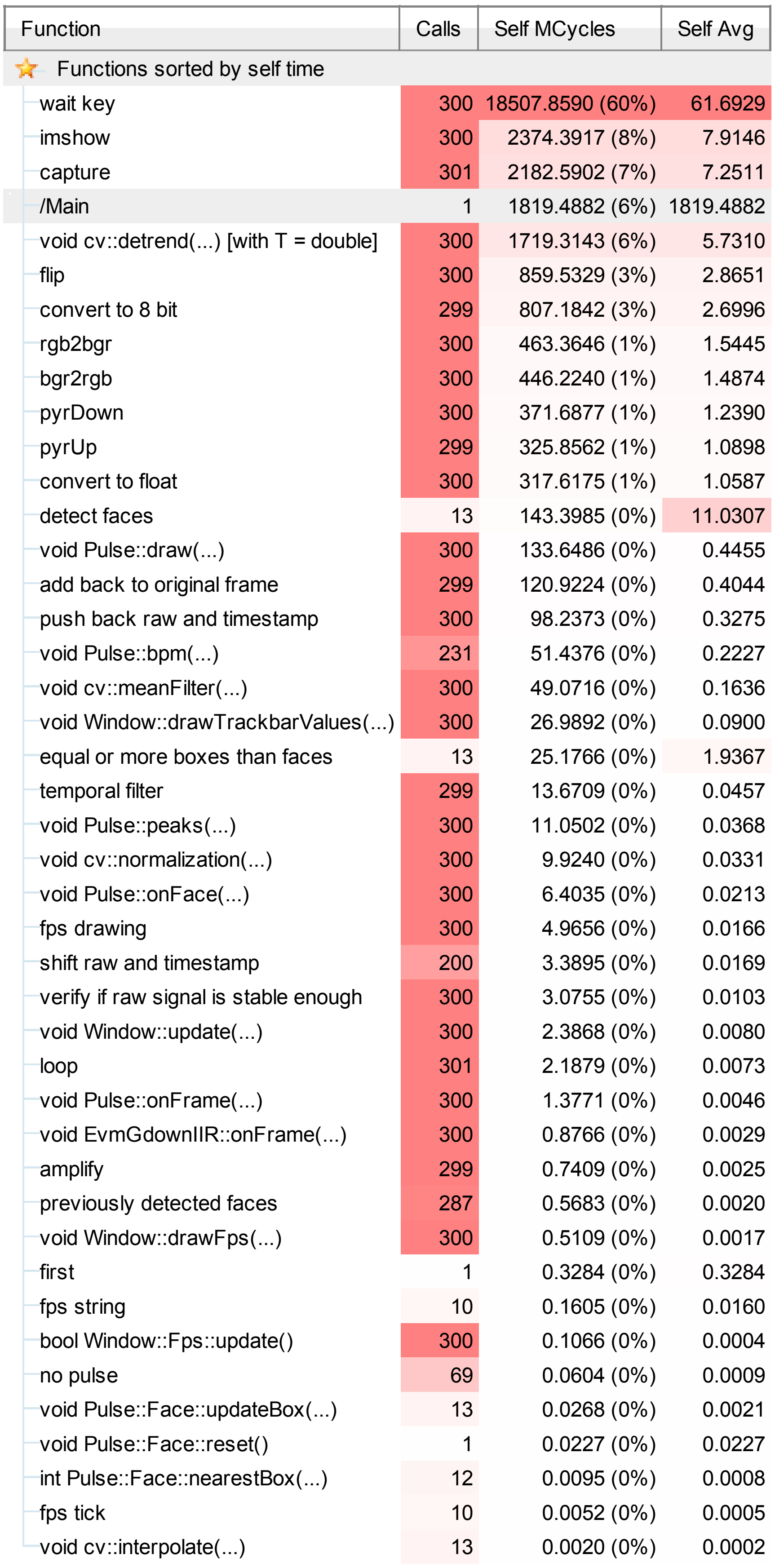


Command Line: pulse-desktop-cpp
Date: Thu May 23 17:50:59 2013
Raw run time: 35951.60 mcycles
Total calls: 9570
rdtsc overhead: 35 cycles
Per call overhead: 70 cycles
Estimated overhead: 0.6702 mcycles

Function	Calls	MCycles	Avg	Self MCycles	Self Avg
⭐ /Main	1	30905.8421 (100%)	30905.8421	1819.4882	1819.4882
⊟ loop	301	29086.3538 (94%)	96.6324	2.1879	0.0073
wait key	300	18507.8590 (60%)	61.6929	18507.8590	61.6929
⊟ void Window::update(...)	300	7534.1838 (24%)	25.1139	2.3868	0.0080
⊟ void Pulse::onFrame(...)	300	4215.0788 (14%)	14.0503	1.3771	0.0046
⊟ previously detected faces	287	3804.9935 (12%)	13.2578	0.5683	0.0020
⊟ void Pulse::onFace(...)	287	3804.4251 (12%)	13.2558	6.1077	0.0213
⊟ void EvmGdownIIR::onFrame(...)	287	1818.3953 (6%)	6.3359	0.7818	0.0027
convert to 8 bit	287	774.7795 (3%)	2.6996	774.7795	2.6996
pyrDown	287	355.8988 (1%)	1.2401	355.8988	1.2401
pyrUp	287	310.5361 (1%)	1.0820	310.5361	1.0820
convert to float	287	246.5864 (1%)	0.8592	246.5864	0.8592
add back to original frame	287	115.9759 (0%)	0.4041	115.9759	0.4041
temporal filter	287	13.1296 (0%)	0.0457	13.1296	0.0457
amplify	287	0.7073 (0%)	0.0025	0.7073	0.0025
void cv::detrend(...) [with T = double]	287	1655.2995 (5%)	5.7676	1655.2995	5.7676
void Pulse::draw(...)	287	128.3121 (0%)	0.4471	128.3121	0.4471
push back raw and timestamp	287	90.8463 (0%)	0.3165	90.8463	0.3165
void Pulse::bpm(...)	222	51.1195 (0%)	0.2303	51.1195	0.2303
void cv::meanFilter(...)	287	33.1456 (0%)	0.1155	33.1456	0.1155
void Pulse::peaks(...)	287	10.6457 (0%)	0.0371	10.6457	0.0371
void cv::normalization(...)	287	4.2982 (0%)	0.0150	4.2982	0.0150
shift raw and timestamp	192	3.2706 (0%)	0.0170	3.2706	0.0170
verify if raw signal is stable enough	287	2.9335 (0%)	0.0102	2.9335	0.0102
no pulse	65	0.0511 (0%)	0.0008	0.0511	0.0008
⊟ equal or more boxes than faces	13	265.3098 (1%)	20.4084	25.1766	1.9367
⊟ void Pulse::onFace(...)	13	240.0949 (1%)	18.4688	0.2959	0.0228
⊟ void EvmGdownIIR::onFrame(...)	13	140.4893 (0%)	10.8069	0.0949	0.0073
convert to float	13	71.0310 (0%)	5.4639	71.0310	5.4639
convert to 8 bit	12	32.4047 (0%)	2.7004	32.4047	2.7004
pyrDown	13	15.7889 (0%)	1.2145	15.7889	1.2145
pyrUp	12	15.3201 (0%)	1.2767	15.3201	1.2767
add back to original frame	12	4.9465 (0%)	0.4122	4.9465	0.4122
temporal filter	12	0.5413 (0%)	0.0451	0.5413	0.0451
first	1	0.3284 (0%)	0.3284	0.3284	0.3284
amplify	12	0.0336 (0%)	0.0028	0.0336	0.0028
void cv::detrend(...) [with T = double]	13	64.0148 (0%)	4.9242	64.0148	4.9242
void cv::meanFilter(...)	13	15.9259 (0%)	1.2251	15.9259	1.2251
push back raw and timestamp	13	7.3911 (0%)	0.5685	7.3911	0.5685
void cv::normalization(...)	13	5.6258 (0%)	0.4328	5.6258	0.4328
void Pulse::draw(...)	13	5.3366 (0%)	0.4105	5.3366	0.4105
void Pulse::peaks(...)	13	0.4045 (0%)	0.0311	0.4045	0.0311
void Pulse::bpm(...)	9	0.3181 (0%)	0.0353	0.3181	0.0353



void Pulse::bpm(...)	9	0.3181 (0%)	0.0353	0.3181	0.0353
verify if raw signal is stable enough	13	0.1419 (0%)	0.0109	0.1419	0.0109
shift raw and timestamp	8	0.1189 (0%)	0.0149	0.1189	0.0149
void Pulse::Face::reset()	1	0.0227 (0%)	0.0227	0.0227	0.0227
no pulse	4	0.0094 (0%)	0.0023	0.0094	0.0023
void Pulse::Face::updateBox(...)	13	0.0287 (0%)	0.0022	0.0268	0.0021
void cv::interpolate(...)	13	0.0020 (0%)	0.0002	0.0020	0.0002
int Pulse::Face::nearestBox(...)	12	0.0095 (0%)	0.0008	0.0095	0.0008
detect faces	13	143.3985 (0%)	11.0307	143.3985	11.0307
imshow	300	2374.3917 (8%)	7.9146	2374.3917	7.9146
rgb2bgr	300	463.3646 (1%)	1.5445	463.3646	1.5445
bgr2rgb	300	446.2240 (1%)	1.4874	446.2240	1.4874
void Window::drawTrackbarValues(...)	300	26.9892 (0%)	0.0900	26.9892	0.0900
void Window::drawFps(...)	300	5.7488 (0%)	0.0192	0.5109	0.0017
fps drawing	300	4.9656 (0%)	0.0166	4.9656	0.0166
fps string	10	0.1605 (0%)	0.0160	0.1605	0.0160
bool Window::Fps::update()	300	0.1118 (0%)	0.0004	0.1066	0.0004
fps tick	10	0.0052 (0%)	0.0005	0.0052	0.0005
capture	301	2182.5902 (7%)	7.2511	2182.5902	7.2511
flip	300	859.5329 (3%)	2.8651	859.5329	2.8651



Function	Calls	Self MCycles	Self Avg
Functions sorted by self time			
wait key	300	18507.8590 (60%)	61.6929
imshow	300	2374.3917 (8%)	7.9146
capture	301	2182.5902 (7%)	7.2511
/Main	1	1819.4882 (6%)	1819.4882
void cv::detrend(...) [with T = double]	300	1719.3143 (6%)	5.7310
flip	300	859.5329 (3%)	2.8651
convert to 8 bit	299	807.1842 (3%)	2.6996
rgb2bgr	300	463.3646 (1%)	1.5445
bgr2rgb	300	446.2240 (1%)	1.4874
pyrDown	300	371.6877 (1%)	1.2390
pyrUp	299	325.8562 (1%)	1.0898
convert to float	300	317.6175 (1%)	1.0587
detect faces	13	143.3985 (0%)	11.0307
void Pulse::draw(...)	300	133.6486 (0%)	0.4455
add back to original frame	299	120.9224 (0%)	0.4044
push back raw and timestamp	300	98.2373 (0%)	0.3275
void Pulse::bpm(...)	231	51.4376 (0%)	0.2227
void cv::meanFilter(...)	300	49.0716 (0%)	0.1636
void Window::drawTrackbarValues(...)	300	26.9892 (0%)	0.0900
equal or more boxes than faces	13	25.1766 (0%)	1.9367
temporal filter	299	13.6709 (0%)	0.0457
void Pulse::peaks(...)	300	11.0502 (0%)	0.0368
void cv::normalization(...)	300	9.9240 (0%)	0.0331
void Pulse::onFace(...)	300	6.4035 (0%)	0.0213
fps drawing	300	4.9656 (0%)	0.0166
shift raw and timestamp	200	3.3895 (0%)	0.0169
verify if raw signal is stable enough	300	3.0755 (0%)	0.0103
void Window::update(...)	300	2.3868 (0%)	0.0080
loop	301	2.1879 (0%)	0.0073
void Pulse::onFrame(...)	300	1.3771 (0%)	0.0046
void EvmGdownIIR::onFrame(...)	300	0.8766 (0%)	0.0029
amplify	299	0.7409 (0%)	0.0025
previously detected faces	287	0.5683 (0%)	0.0020
void Window::drawFps(...)	300	0.5109 (0%)	0.0017
first	1	0.3284 (0%)	0.3284
fps string	10	0.1605 (0%)	0.0160
bool Window::Fps::update()	300	0.1066 (0%)	0.0004
no pulse	69	0.0604 (0%)	0.0009
void Pulse::Face::updateBox(...)	13	0.0268 (0%)	0.0021
void Pulse::Face::reset()	1	0.0227 (0%)	0.0227
int Pulse::Face::nearestBox(...)	12	0.0095 (0%)	0.0008
fps tick	10	0.0052 (0%)	0.0005
void cv::interpolate(...)	13	0.0020 (0%)	0.0002