Saliency Computation on Intel Galileo (Gen 2) Platform Using OpenCV

Aditya Sundararajan, Chinnakkannu Adaikkala Raj, Natarajan Rajkumar, Shehzad Jalaluddin
School of Computer Engineering, Nanyang Technological University

***Abstract*—We implemented a model on saliency-based visual attention[1] on Intel Galileo platform, and measured the performance and power consumption of the platform by executing different OpenCV[2] functions. Our tests which were performed on different images (varying in feature and resolution) prove that for low resolution images, Intel Galileo is a reasonable choice for compute intensive OpenCV applications. Also, Intel Galileo being a low power consuming platform is an added advantage.**

Keywords—Intel Galileo, OpenCV, filter2D, Saliency.

I. INTRODUCTION

Computers scan images in a pixen by pixel fashion, from top to bottom (or bottom to top, depending on the algorithm implemented). In primates, this is not the case. Primates' interpretion of visuals is completely different. Their attention falls on those features which are most attractive. This procedure follows and the entire scene is captured in the mind of primates. [1]

Fig. 1 is an image of the popular Walt Disney creation, Donald Duck. When this image is presented to a human being, the human eyes gets attracted towards Donald's head, his webbed feet, his hands, and so on. Had the same image been presented to a computer, the same image would have been interpreted pixel by pixel.

Fig. 1: Donald Duck

In this project, we have implemented a model on saliency-based visual attention[1] on Intel Galileo platform, so that the computer would interpret the image in a way, similar to primates.

The Intel Galileo (Gen 2) is an embedded platform powered by Intel Quark SoC X1000 application processor with Arduino compatible hardware shields. Quark X1000 is a single core with a clock frequency of 400 MHz. It has a 32-bit x86 Instruction Set Architecture. The Quark SoC has an embedded SRAM of 512 KB and DRAM of 256 MB. The flash memory is 8 MB (NOR Flash) and microSD card support upto 32 GB. The Intel Galileo opeates at 5V. [3] In our project, we have used this platform for saliency map computation.

Saliency map computation is a complex process which involves calculation of conspicuity maps for Intensity, Color, and Orientation. Combining all these maps, would help in computation of the saliency map of an image.

We have extensively used OpenCV for computing the saliency map in our project. OpenCV is set of open source computer vision libraries, designed for more computational efficiency with strong focus on real time applications.

II. MOTIVATION

With the advancement of semiconductor technologies, today we have lot of low cost, high performance, power efficient processors available for embedded application. Also, we have open source computer vision software algorithms such as OpenCV for image processing. This gave us the motivation to develop a low cost embedded platform to perform saliency detection. And we chose to implement this on Intel Galileo Embedded Platform.

III. METHOD

Input is provided in the form of static color images, usually digitized at M x N resolution (where, M = number of rows, N = number of columns). Fig. 2 shows the flowchart of the Saliency Map model.

First stage is linear filtering of the input image. The input image is separated into three separate images corresponding to the three channels (Red, Green, and Blue) of the input image. The average of these three images gives the Intensity image, *Intensity*. Then, a Gaussian Pyramid of 9 images (0 to 8) is constructed from the Intensity image for Intensity Map, $I(\sigma)$. The addition of these Intensity Map by proper centre-surround operation[1] gives the Intensity Conspicuity Map, *I_bar*.

The input image is normalized and the normalized image is used to extract Red, Green, Blue, and Yellow Channels. Then four separate Gaussian Pyramids of 9 images each is constructed for each channel, that is, $R(\sigma)$, $G(\sigma)$, $B(\sigma)$, $Y(\sigma)$. The addition of these Color Maps by proper centre-surround operation [1] gives the Color Conspicuity Map, *C_bar*.

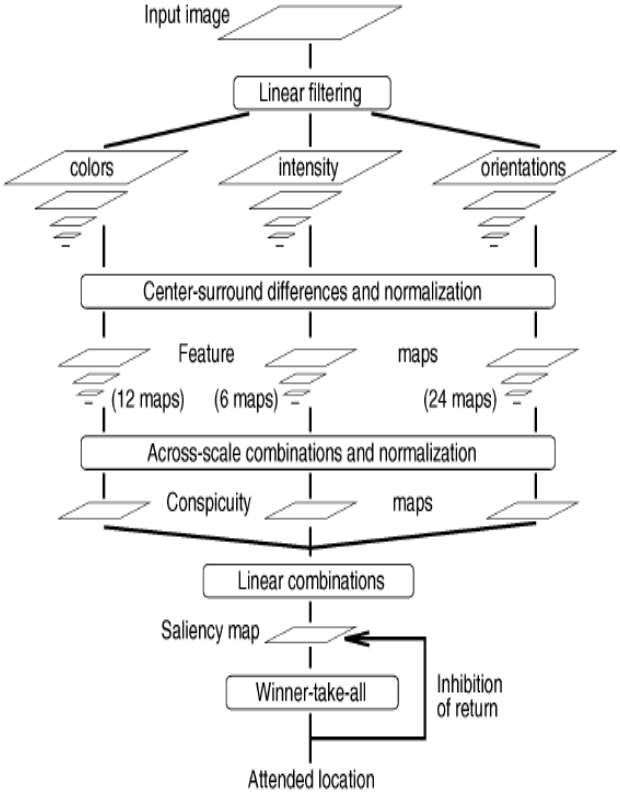

Fig. 2: Flowchart Saliency Map model

Gabor kernel for four different angles, that is, 0°, 45°, 90°, and 135°, is generated. These four Gabor kernels are then applied to each pixel of intensity image and four images are obtained, *gaborImage(0), gaborImage(1), gaborImage(2), gaborImage(3),* corresponding to 0°, 45°, 90°, and 135°, respectively. The Gaussian Pyramid of these four images are obtained and added up after applying proper centre-surround operation [1] to get four orientation maps, *totalOriMap_0, totalOriMap_45, totalOriMap_90, totalOriMap_135*. The addition of the four orientation maps gives is the Orientation Conspicuity Map, *O_bar*.

The Saliency Map is obtained by talking average of *I_bar, C_bar, and O_bar.* The location of the highest pixel intensity in Saliency Map is marked with a circle, indicating a salient object. This process of finding salient object is repeated till no further salient objects can be identified. [1]

IV. EXPERIMENTS

The experimental setup consists of Personal Computer (PC) and Intel Galileo board, booted with vmlinux-v3.8.7-yocto-standard from microSD card. The IP address of the board ca nbe obtained, either by connecting it to the router or using Arduino sketch program to assign static IP to the board. Once IP is configured, we connected to the board by using "ssh root@IP_address" on the command line. The OpenCV library v2.8 [2] is supported in this standard Linux image by default.

The following OpenCV functions [4] are used in different stages of saliency computation.

Operation	OpenCV functions used
Gaussian Pyramid	split(), buildPyramid()
resizeMap()	zeros(), pyrDown()
buildMap()	pyrUp(), abs(), resizeMap
Intensity Map	buildMap()
Color Map	minMaxLoc(), split(), threshold(), buildPyramid(), pyrUp(), abs(), resizeMap()
Orientation Map	getGaborKernel(), filter2D, buildPyramid(), buildMap()
Saliency Map	resize(), threshold(), minMaxLoc(), circle(), line()

Table 1: OpenCV functions used for different operations

V. OPTIMIZATIONS

A. *Code Optimization*

For image resizing, OpenCV provides two functions: resize(), and pyrUp()/pyrDown(). We did a comparison between these functions and observed that pyrUp()/pyrDown() perfoms resizing of images much faster than resize(). In one case, where we had to build a Gaussian Pyramid, pyrDown() performed the task approximately 3 times faster than resize (28ms vs 89ms). After this observation, we replaced resize with pyrUp()/pyrDown(), in locations which required multiple resizing of images.

In the creation of Orientation Maps, we achieved optimization by reducing the number of calls of filter2D function. By changing the sequence of the code, we managed to reduce filter2D calls from 24 to 4. The reduction in the number of filter2D calls did not affect the final saliency map in any way and we obtained proper saliency map.

All these optimizations led to a drastic reduction in calculation of saliency maps from 41 seconds to 20 seconds, for an image of resolution 1024x768.

B. *Functional Optimization*

We removed image normalization in calculation of Color Map. Also, we directly used the input image to extract Red, Green, and Blue channels (i. e., without using formulas given in paper [1]). This led to a further reduction in latency by approximately 4 seconds. We compared the two output images:

one with Code Optimization and the other with both optimizations (Code and Functional Optimizations). We did not find any visual degradation in quality of the output images. Fig. 3 shows the Code Optimized output image (Left) and output image with both optimizations (Right).

Fig. 3: Optimized output images

Additionally we tried multithreading using POSIX thread support API's but it didn't improve the execution latency.

VI. RESULTS

The performance analysis is based on execution time to find the saliency map of a static input image. Fig.4 shows the time taken by each stage in saliency computation for an image of size 1024x768 pixels.

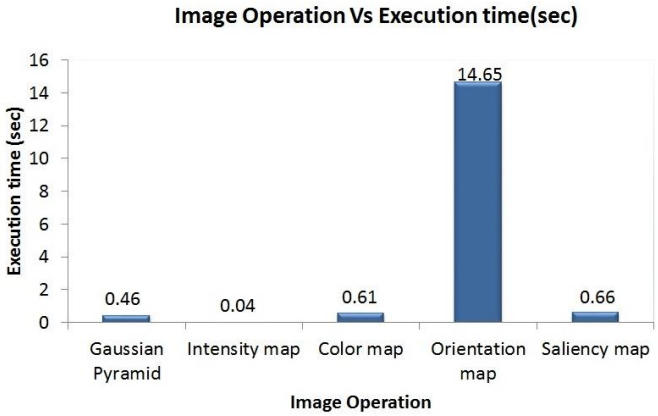

Fig. 4: Time taken by each operation

As shown by Fig. 4 , the graph is skewed towards Orientation Map. Profiling the code for Orientation Map led to the conclusion that the filter2D() function is the culprit. The filter2D() function takes about 80% of the total time taken by Orientation Map. Unfortunately, we could not find a better alternative to filter2D() which could do the same job much faster.

The output images and intermittent process images are shown in Fig 5.

We executed the same operation for different image resolutions. The result shows that image with lower image resolution takes less time for computation. Fig 6 shows the variation in execution time for different images.

The Power measurement at each phase of the board is listed in fig. 7 As seen from fig. 7, the power consumption of Intel Galileo is not high even when the board is executing.

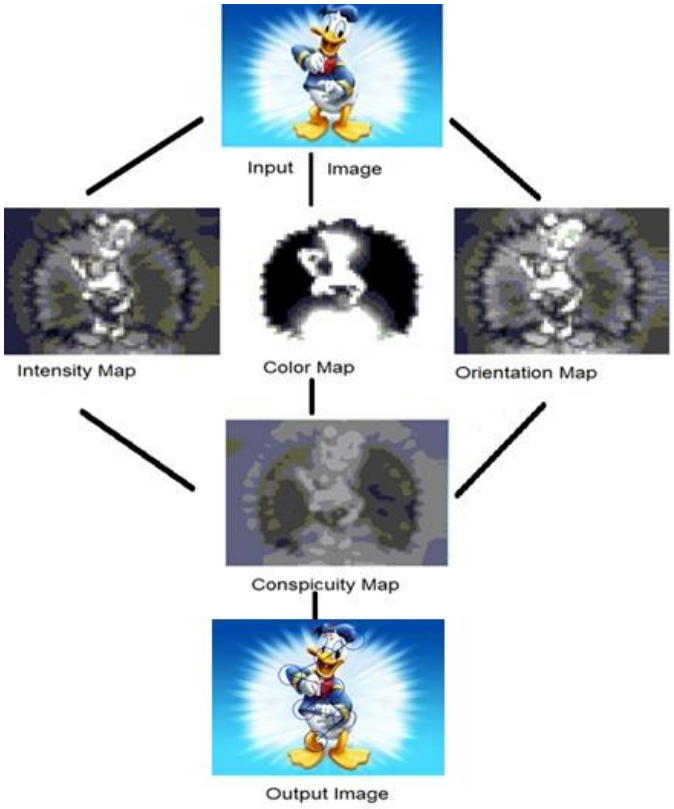

Fig. 5: Stages of operation

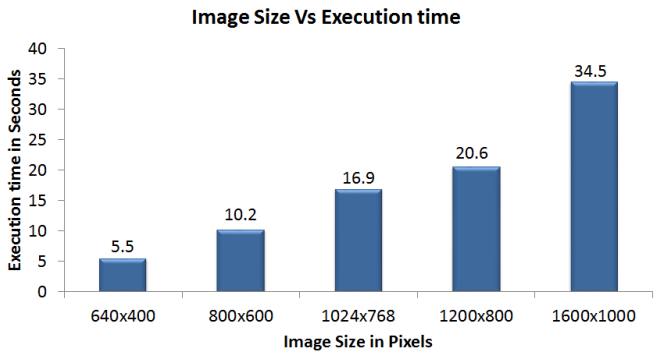

Fig. 6: Image Resolution v/s Execution Time

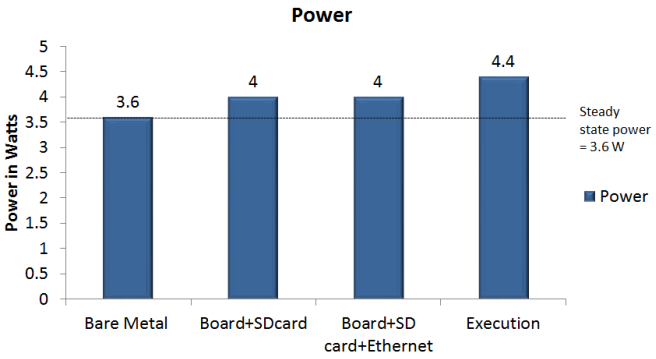

Fig. 7: Power Consumption

VII. CONCLUSION

After replacing the resize() function with pyrUp()/pyrDown(), and by changing the sequence of the code to reduce filter2D() calls, we managed to reduce the execution time from 41 seconds to 20 seconds. Further reduction in execution time (4 seconds) without loss in quality, was obtained by optimizing the code for Color Map. Orientation Map constituted the biggest chunk in the execution time, mainly due to filter2D(). We observed an increase in execution time with increase in image resolution.

Overall, Intel Galileo does a decent job in the case of low resolution images, with low power consumption being the stand-out feature.

VIII. FUTURE WORK

In the course of this project, we tried to get the webcam working with the board, so as to perform saliency computation with dynamic images. Unfortunately, we were not successful So, we should put in more efforts to get the webcam working with Intel Galileo.

The Achilles' Heel in our project was the filter2D() function which drastically increased the execution time. A solution to this may be using an alternative filter2D() function, if available, or create our own optimized algorithm which does the same job, but in a much efficient way.

REFERENCES

[1] L. Itti, C. Koch, and E. Niebur, "A model of saliency-based visual attention for rapid scene analysis," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 20, no. 11, pp. 1254– 1259, Nov 1998.

[2] OpenCV [Online]. Available: www.opencv.org

[3] Galileo Datasheet [Online]. Available: https://communities.intel.com/servlet/JiveServlet/downloadBody/21835-102-5-25148/Galileo_Datasheet_329681_003.pdf

[4] OpenCV Docs [Online]. Available: www.docs.opencv.org