OpenCV optimisations with Ne^{10} using auto-vectorisation for the Cortex-A9

Beau Johnston[*]
Open Parallel

(Dated: July 18, 2012)

The recent launch of ARM's Ne^{10} framework introduces the possibility of increasing runtime performance of software running on ARM architectures; specifically the Cortex-A9 hardware, commonly found in many modern smartphones. Meanwhile there is a growing interest in computer vision and signal processing; as such, OpenCV (Open Computer Vision) has become a mainstream software framework with long term exposure within the software development community, it is complete with some 500+ commonly used image processing operations.

It is beneficial to examine whether OpenCV could profit from the utilisation of Ne^{10} when running on ARM architectures. Ne^{10} is capable of operating with gcc (GNU Compiler Collection) to perform auto-vectorisation; which could reduce execution time of many commonly used image processing operations. As a majority of Cortex-A9 enabled devices are running the Android platform, it is beneficial to examine how the JNI (Java Native Interface) can be used to tackle such auto-vectorisations. Consequently an in-depth analysis of auto-vectorisation using Ne^{10} and gcc for the Android JNI will be performed. The greater part of this article will discuss the quantitative results of such auto-vectorisations on OpenCV.

Today there are many smartphones utilising ARM's Cortex-A9 processor, a large proportion of these use the Android Platform. As such, this article will specifically focus on reducing execution time of code developed for Android's JNI (Java Native Interface).

Image Processing and Computer vision operations are typically a bottle neck for many Android apps; this is especially true for those involved with playing, recording and/or editing photos and videos. Much of this multimedia congestion is compounded by the large amounts of data the recording devices can produce; as many of today's Android devices boast high resolution (up to 8 Megapixel) cameras.

The device where this choke point is most apparent is with the latest release of Android smartphones, the Samsung Galaxy S III. It sports a 4.8 inch 720x1280 Super AMOLED (active-matrix organic light-emitting diode) display, 2GB of RAM (random-access memory), quad-core 1.4 GHz Cortex-A9 processor fitted on Exynos 4 Quad chipset, an 8 megapixel rear camera and a 1.9 megapixel front facing camera. The findings and remainder of this article involve development, and benchmarking on the Samsung Galaxy S III.

ARM have recently launched the Ne^{10} software framework, which consist of a set of optimised system calls which target many ARM architectures, especially the Cortex-A9. The goal of this article is to introduce and benchmark a possible method to decrease execution time of ARMs Cortex-A9 hardware when performing commonly used image processing operations.

A rudimentary technique to perform runtime optimisations is known as auto-vectorisation. A brief introduction will be provided on auto-vectorisation and how gcc (GNU Compiler Collection) can be forced to perform it automatically. An outline of the fundamentals of Ne^{10}, specifically what it does, how it's used and how to compile it as JNI code. Next OpenCV will be briefly discussed where emphasis will be placed on some of the core image processing algorithms that exist in the framework. These points will be addressed in the Preparatory Considerations section of this article. The remainder of the article will probe deeper into a few selected image processing filters that were developed by chaining several of OpenCV's core operations/functions together. Each filters relevance will be discussed in addition some of its mathematical theory, a computational perspective of its application and its runtime results. A strong emphasis will be placed on the runtime results of each filter, where a version without Ne^{10} will be compared against one employing ARM's framework (Note both employ gcc's auto-vectorisation). The chosen filters are 'Sepia Toning', 'Fun House', 'Sketch Book' and 'Neonise'. In conclusion an outline is provided on the key findings of this research, followed by a discussion where several considerations are mentioned as a basis for potential areas of improvement.

1. PREPARATORY CONSIDERATIONS

Prior to examining the aspects and runtime performance of each of the four individual image filters, there are several restrictions and considerations that require discussion. This is especially true when regarding auto-vectorisation, Ne^{10}, OpenCV, and how benchmarking has been performed.

1.1. Auto-Vectorisation

Much of this article is focused on how auto-vectorisation is done implicitly by gcc, therefore it is

[*]Electronic address: beau.johnston@openparallel.com; URL: http://openparallel.com

not specifically mentioned during any of the image filters later described. However its relevance is fundamental and auto-vectorisation occurs on both filters with and without Ne^{10}.

Auto-vectorisation within gcc specifically examines loop vectorisation. It is a process that examines procedural loops and assigns a separate processing unit to all data items (instances of variables) during each iteration. The goal is to find and exploit SIMD (Single Instruction Multiple Data) parallelism at loop level. Automatic vectorisation is performed with gcc by enabling the flag `-ftree-vectorize` and by default at `-O3`.

Typical reservations around vectorisation are because improper auto-vectorisation can lead to slow execution due to pipeline synchronisation and data movement timing. Major considerations around Ne^{10}'s use of auto-vectorisation include that it requires images to be packed in an RGBA (red,green,blue,alpha) or any other 4 channel format for ready vectorisation, and all Ne^{10}'s set of functions operate on `float*`[4].

1.2. Ne^{10}

Ne^{10} is an open source library that uses NEON optimised functions. NEON consists as a set of optimised functions that boast improved performance of SIMD instructions for the ARM Cortex A processors. These functions are developed in C and Assembler and were designed to a accelerate vector and matrix operations. When dealing directly with low level vector arithmetic the framework has been targeted to reduce processing time.

NEON instructions perform "Packed SIMD" processing; this means that registers are treated as vectors of values as long as they are of the same data type. The supported data types are signed and unsigned 8, 16, 32 and 64 bit single precision `float*`. An obvious limitation when working with OpenCV is that the data should be normalised to ensure images exists as structs of floats rather than the generic OpenCV `IplImage` data type as a struct of `ints`. Examples of how to use and compile Ne^{10} are available.[1] The repository contains a precompiled version of Ne^{10} running in a blank Android App.

1.3. OpenCV

OpenCV (Open Source Computer Vision) is a library of programming functions for computed vision, signal and image processing. It has support for matrix mathematics, fitting, utilities and data structures (typically needed when working with images), camera calibration, image pyramids, geometric descriptors, feature detection, recognition and tracking, machine learning, transforms, segmentation and more general image processing functions. Most functions are written in C, and have been optimised by several programmers during its long development cycle. The majority of functions and data types used throughout this article focus on the general image processing functions and image transforms.

1.4. Benchmarking

The major form of benchmarking of the runtime results of each image filter function has been via absolute system time stamps. They provide an impartial system that adequately examines execution time of image processing in OpenCV whilst examining its hit on absolute system resources. Benchmarking for each image filter and optimisation combination used 100 iterations, where each iteration was recorded and used to give an unbiased mean and standard deviations on a large test set. As the benchmarking was bound by the Samsung Galaxy S III's resources and Android's Gingerbread OS (Operating System), note that there are occasional spikes in runtimes. These could be explained by a hold up on system resources, or most likely several heavy weight OS processes requiring immediate execution.

Throughout the remainder of this article standardised test images have been used. This is performed to ensure an equitable metric for all image filters and with all Ne^{10} combinations. For the purposes of a suitable simulation the standard test image has been computed at several key resolutions. These resolutions match the optimal ratio and sampling rate for both of the Samsung Galaxy S III cameras. The test image Figure 1 is of Sydney Harbour containing approximately equal proportions of natural and artificial features.

FIG. 1: The bench marking test images at resolutions:
(a) 1632×1224 (Samsung Galaxy S III Rear camera),
(b) 640×480 (Samsung Galaxy S III Front camera).

[1] A Github repository on working with Ne^{10}: https://github.com/openparallel/NeonBuild

2. SEPIA TONING

Sepia toning is a specialised treatment to give a black-and-white photographic print a warmer tone. It is an ageing effect, and is commonly used by most photographers. Sepia Toning is performed by applying a simple tint to a black and white image. As the camera device collects images in colour, an additional greyscale conversion step is required.

2.1. Mathematical Theory

Sepia Toning contains two steps; first to perform a greyscale conversion, secondly to apply a sepia (brown-grey) tinge. For a digital image denoted $I(x, y)$ of N_x pixels wide by N_y pixels heigh, the greyscale conversion step is described as follows:

$$P_I(x,y) = \frac{P_R(x,y) + P_G(x,y) + P_B(x,y)}{3} \tag{1}$$

for all pixels (x, y) in the domain of the image where:
x is the column index,
y is the row index,
$P_R(x, y)$ is the red value of the selected pixel,
$P_G(x, y)$ is the green value of the selected pixel,
$P_B(x, y)$ is the blue value of the selected pixel.

Tinging is applied by first creating an empty RGB image array of 3 $\times N_x \times N_y$ elements, adding the greyscale image $P_I(x, y)$ to each of these three bands, and then applying a band-specific scalar bias to each element of the form,

$$\begin{aligned} P_T(x,y) &= [P_{R^\star}, P_{G^\star}, P_{R^\star}] \qquad (2)\\ &= [P_I(x,y) + 40, P_I(x,y) + 20, P_I(x,y) - 20]. \end{aligned}$$

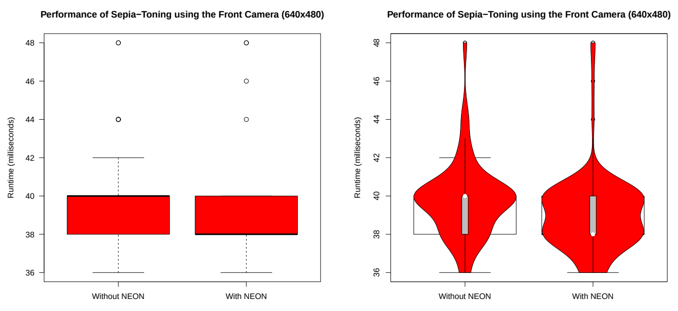
So that an RGB image is recreated from the greyscale image P_I.

2.2. Computational Perspective

BGR is the default channel order for OpenCV not RGB, this must be considered when working with OpenCV. The greyscale conversion can be done automatically using `cvCvtColor(src, dst, CV_BGR2GRAY);` where `src` and `dst` are source and destination `IplImage*`'s respectively. Each pixel component can be collected using `cvGet2D(*IplImage,x,y)` and a BGR scalar value can be created using `cvScalar(p-20, p+20, p+40);`[3].

2.3. Runtime Results

Below are the runtime results for comparing Ne^{10} on the Sepia Toning filter, 100 iterations have been performed to provide a fair sample set. Both high (1224×1632) and low resolution (640×480) image runtime results, with, and without Ne^{10} optimisations are shown. They are each displayed as both a box and violin plot, this is done to offer a different perspective of runtime performance.



(a) (b)

FIG. 2: The benchmarking runtimes of 100 iterations at resolution 640×480 (Samsung Galaxy S III Front camera):
(a) as a boxplot,
(b) as a violinplot.

From Figure 2 we can see that the runtime result for the low resolution image without Ne^{10} is 39.78 ± 2.01 milliseconds. Whilst the runtime result for the low resolution image with Ne^{10} is 39.00 ± 2.04 milliseconds.This indicates that Ne^{10} makes runtimes marginally faster. To confirm that the two means are statistically significantly different we perform a Welch two-sample t test. The results were:

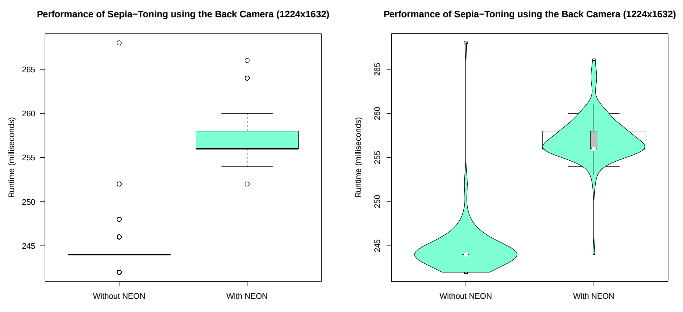
```
> tTest <- t.test(WO,WN, alternative = 'greater')

Welch Two Sample t-test

data:  WO and WN
t = 2.725, df = 197.95, p-value = 0.003504
alternative hypothesis: true difference \\
in means is greater than 0
95 percent confidence interval:
 0.3069575       Inf
sample estimates:
mean of x mean of y
    39.78     39.00
```

Therefore we reject the null hypothesis that the means are the same as the p-value is less than 0.05 (the value below which the differences in the means is deemed statistically significant). In other words the mean runtime performance using Ne^{10} is statistically significantly better than without using Ne^{10}. It is concluded that using

Ne[10] provides a marginal improvement in code execution time for low resolution images when Sepia Toning.



(a) (b)

FIG. 3: The benchmarking runtimes of 100 iterations at resolution 1224×1632 (Samsung Galaxy S III Front camera):
(a) as a boxplot,
(b) as a violinplot.

In Figure 3 we can also see that the runtime result for the high resolution image without Ne[10] is 244.52 $\pm$ 2.94 milliseconds. In contrast, the runtime result for the high resolution image with Ne[10] is 257.26 $\pm$ 2.18 milliseconds. From this result we can conclude that Ne[10] significantly worsens runtime performance of Sepia Toning. This is supported statistically via the following t-test which has a p-value of 1.0.

```
Welch Two Sample t-test

data:  WO and WN
t = -34.8609, df = 182.619, p-value = 1
alternative hypothesis: true difference
in means is greater than 0
95 percent confidence interval:
 -13.34418        Inf
sample estimates:
mean of x mean of y
   244.52    257.26
```

3. FUN HOUSE

The fun house filter effect is similar to that of a house of mirrors at a carnival. It heavily utilises mirror extensions throughout the entire filter, as well as the erode and dilate image morphology operations. This filter was used to examine the performance of Ne[10] and its auto-vectorisation optimisations on a subset of OpenCV's image morphology group of functions.

3.1. Mathematical Theory

The Fun House filter is partitioned into three distinct operations, namely; the mirror, erode and dilate phase.

3.1.1. Mirror

First define the parameter c via:

$$c = \frac{N_x}{2}$$

where c is the horizontal centre of the image.

The mirroring operation $\mathcal{M}\{\cdot\}$ is then defined by,

$$\mathcal{M}\{I(x,y)\} = I_{RGB}(c-x,y) \tag{3}$$

where $I_{RGB}(x,y)$ indicates that the input image has three channels (RGB) with the subtraction operation occurring independently on each of these bands

3.1.2. Erosion

Erosion operations use a square kernel $K(x,y)$ of length two in order to perform set operations on the image pixels. Within each RGB channel a sliding window operation is performed and the minimum value of those image pixels captured by the kernel is assigned to the value of the eroded image at location (x,y) for the channel under consideration. This operation is performed for all channels and locations. Denoting the *eroded* image as $E_{RGB}(x,y)$ this operation is expressed mathematically as:

$$E_{RGB}(x,y) \;=\; \min_{(x',y')\,\epsilon\,K(x,y)} I_{RGB}(x+x^{'},y+y^{'}) \tag{4}$$

3.1.3. Dilation

The dilation operation is very similar to that of the erosion except that the maximum value (rather than the minimum) within the set if found.

$$D_{RGB}(x,y) \;=\; \max_{(x',y')\,\epsilon\,K(x,y)} I_{RGB}(x+x^{'},y+y^{'}) \tag{5}$$

where the dilation image is denoted $D_{RGB}(x,y)$.

3.2. Computational Perspective

The mirroring phase requires explicit indexing of pixels; this requires two nested `for` loops. Accessing each pixel element is performed identically to the method discussed in Section 2.2. The Erosion and Dilation operations are handled entirely by the OpenCV functions `cvErode(src,dst,kernel,iterations)` and `cvDilate(src,dst,kernel,iterations);` where `src` and `dst` are of type `IplImage*`. The `kernel` is used to determine the window in which all $x^{'}$ and $y^{'}$ are located around the targeted pixel. Finally `iterations` is an `int` to determine how many repeated treatments are to occur.

The `kernel` is generated using `cvCreateStructuringElementEx(cols,rows,anchor_x,anchor_y,shape,values)`, with the `cols` and `rows` are `ints` which indicate the size of the rectangle that holds the structuring element. `anchor_x` and `anchor_y` are the coordinates of the anchor point within the enclosing rectangle of the kernel. The variable `shape` uses an int value to determine which shape is to be used; candidates are `CV_SHAPE_RECT` (the kernel is rectangular), `CV_SHAPE_CROSS` (the kernel is cross shaped), `CV_SHAPE_ELLIPSE` (the kernel is elliptical) and `CV_SHAPE_CUSTOM`. If `CV_SHAPE_CUSTOM` is used, the final argument `values` are needed; it is of type `int*` these values (points) are used to define the custom shape of the kernel.

3.3. Runtime Results

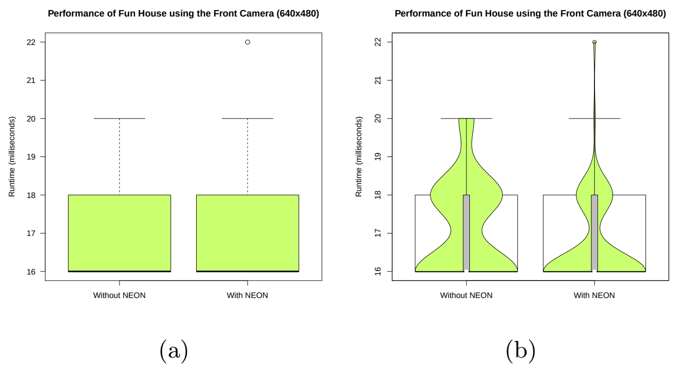


FIG. 4: The benchmarking runtimes of 100 iterations at resolution 640×480 (Samsung Galaxy S III Front camera):
(a) as a boxplot,
(b) as a violinplot.

The summary statistics of this test are as follows:
`Mean without Neon` = 17.08 ± 1.284563 milliseconds.
`Mean with Neon` = 16.62 ± 1.089713 milliseconds.

By inspection of these results it is difficult to determine if there is a significant difference in mean computation times, therefore a t-test was used to detect the presence of a statistical difference. The results were:

```
Welch Two Sample t-test

data:  WO and WN
t = 2.7308, df = 192.873, p-value = 0.003452
alternative hypothesis: true difference
in means is greater than 0
95 percent confidence interval:
 0.1815854       Inf
sample estimates:
mean of x mean of y
    17.08     16.62
```

indicating that on average the processing with Ne[10] ran significantly quicker although it must be kept in mind that this was only 0.46 milliseconds quicker on average.

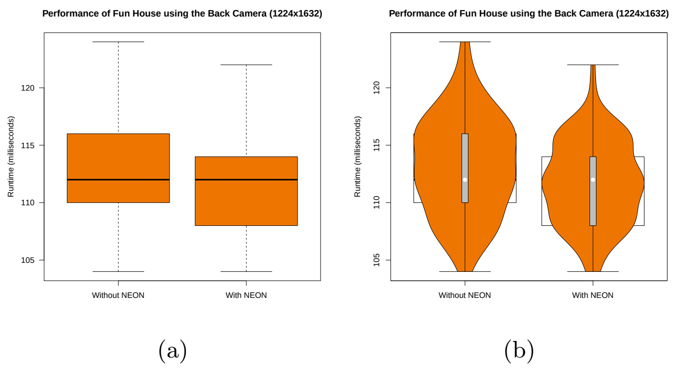


FIG. 5: The benchmarking runtimes of 100 iterations at resolution 1224×1632 (Samsung Galaxy S III Front camera):
(a) as a boxplot,
(b) as a violinplot.

The Fun house filter was also tested on the high resolution back camera, the results were as follows.
`Mean without Neon` = 113.26 ± 4.279668 milliseconds.
`Mean with Neon` = 111.68 ± 3.603814 milliseconds.

Comparison of these results suggests little difference in processing times but the t-test (whose results are displayed below) suggests otherwise as the p-value is less than 0.05. In other words the processing time with NEON for this filter is statistically significantly quicker (1.58 milliseconds on average) than without Ne[10].

```
Welch Two Sample t-test

data:  WO and WN
t = 2.824, df = 192.425, p-value = 0.002621
alternative hypothesis: true difference in means
is greater than 0
95 percent confidence interval:
 0.655267      Inf
sample estimates:
mean of x mean of y
   113.26    111.68
```

4. SKETCH BOOK

The sketch book filter effect aims to achieve automatic conversion of a given image, acquired from the Samsung Galaxy S III hardware, to appear as if it were a hand drawn sketch. This filter was used to examine the performance of Ne[10] and its auto-vectorisation optimisations on OpenCV's smoothing capabilities. Smoothing is a frequently used image processing operation both to reduce noise, camera artefacts and to reduce the resolution of images. Optimising the run time of such a commonly used operation would be beneficial within many applications.

4.1. Mathematical Theory

The Sketch Book filter effect is partitioned into four prominent operations, namely: (i) bitwise inversion, (ii) image smoothing,(iii) comparison, and (iv) greyscale transformation phase.

4.1.1. Bitwise Inversion

$$N_{RGB}(x, y) = -B_{RGB}(x, y) \tag{6}$$

where $B_{RGB}(x, y)$ is a binary number representation of the RGB input image, $I_{RGB}(x, y)$, and $N_{RGB}(x, y)$ is the binary bitwise inverted image.

4.1.2. Image Smoothing

A Gaussian filter $G_\sigma(x, y)$ is generated by

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{x^2+y^2}{2\sigma^2}} \tag{7}$$

where $\sigma \, \epsilon \, \mathbb{R}^+$ is a parameter controlling its smoothness. The 'smoothed' image $S_\sigma(x, y)$ is then formed by the discrete cyclic channel-wise convolution of this filter with the original image $I_{RGB}(x, y)$.

$$S_{RGB}(x, y) = I_{RGB}(x, y) \otimes G_\sigma(x, y) \tag{8}$$

where $\otimes$ is a discrete convolution

where $I_{RGB}(x, y)$ is the input image, σ is the standard deviation parameter of the Gaussian filter, and $S_{RGB}(x, y)$ is the smoothed image.

4.1.3. Comparison Filter

The comparison filter $K_{RGB}(x, y)$ compares the sums of the bitwise inversion and smoothed image components for each pixel. If the sum is greater than a pre-specified value (M) the output of the comparison filter (for that channel and pixel) is 'clipped' to a maximum otherwise it is left unchanged. This filter is specified mathematically as,

$$K_{RGB}(x, y) = \begin{cases} \mathsf{M}, \text{ if } \{N_{RGB}(x, y) + S_{RGB}(x, y)\} < \mathsf{M} \\ \{N_{RGB}(x, y) + S_{RGB}(x, y)\} \text{ otherwise.} \end{cases} \tag{9}$$

In this project the parameter M was pre-set as the maximum value of each pixel channel intensity.

4.1.4. Greyscale Transformation

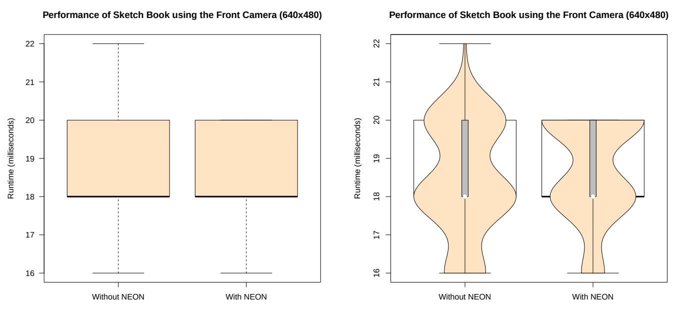
Greyscale Transformation is identical to the operation described in Section 2.1 and acts on $K_{RGB}(x, y)$ to generate $K_I(x, y)$ which is the final output for the sketch book filter.

4.2. Computational Perspective

Many of these operations are handled implicitly in OpenCV. To generate $N_{RGB}(x, y)$, `cvNot(src,dst)` is used; this inverts every bit in each element of `src`, then places the result in `dst`. `src` and `dst` are of type `IplImage*`. The filter $S_{RGB}(x, y)$ is computed via `cvSmooth(src,dst,smooth_type,param1,param2)`. `src` and `dst` operate as typical in Sections 2.2, 3.2; `smooth_type` uses an `int` to determine which of five types of smoothing operation should be applied. This paper assumes smoothing against a Gaussian filter which is set with the flag `CV_GAUSSIAN`.

Finally `param1` and `param2` are `ints` to determine the window size of the filter. As stated in Section 4.1.4 this article employs a 7 × 7 window (`param1 = 7, param2 = 7`). The comparison step can be trivially implemented based off individual pixel access, described in Sections 3.2 and 2.2. The final greyscale conversion on O_{RGB} to generate O_I is also identical to that done in Section 2.2.

4.3. Runtime Results



(a) (b)

FIG. 6: The benchmarking runtimes of 100 iterations at resolution 640×480 (Samsung Galaxy S III Front camera):
(a) as a boxplot,
(b) as a violinplot.

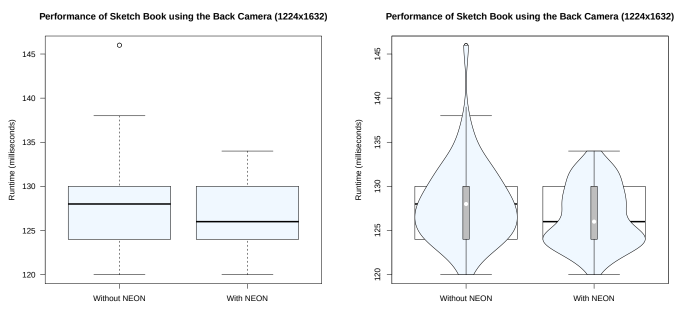
Figure 6 displays both the boxplots and violinplots for the benchmarking of NEON using Sketch Book Filter. There appears to be little difference in processing times between the two devices. This is supported by the calculation of the means and standard deviations.
`Mean without Neon` $= 18.40 \pm 1.477098$ milliseconds.

`Mean with Neon` = 18.78 ± 1.330148 milliseconds. A statistical t-test was used to detect any potential differences in these mean values. A p-value of 0.02869 indicates that there is in fact a statistically significant difference with the NEON processor completing calculations that are on average 0.02 milliseconds faster than those without the NEON.

```
Welch Two Sample t-test
data:  WO and WN
t = -1.9117, df = 195.865, p-value = 0.02869
alternative hypothesis: true difference in means
is less than 0
95 percent confidence interval:
Inf -0.05149205
sample estimates:
mean of x mean of y
    18.40     18.78
```

The Sketch Book filter was also tested using the high-resolution camera of the Galaxy device, the results are displayed in Figure 7. The summary statistics for the processing times were:
`Mean without Neon` = 128.1 ± 4.489044 milliseconds.
`Mean with Neon` = 126.52 ± 3.370939 milliseconds.

Again there appears to be a slight difference between the mean processing times but substantial overlap. Nonetheless a t-test indicated that the processing using the NEON device was statistically significantly quicker (p = 0.002709) than that without using the NEON device. Again though the NEON device was only marginally (1.58 milliseconds) quicker on average.



(a) (b)

FIG. 7: The benchmarking runtimes of 100 iterations at resolution 1224×1632 (Samsung Galaxy S III Front camera):
(a) as a boxplot,
(b) as a violinplot.

```
Welch Two Sample t-test
data:  WO and WN
t = 2.8145, df = 183.714, p-value = 0.002709
alternative hypothesis: true difference
in means is greater than 0
95 percent confidence interval:
 0.6519322       Inf
sample estimates:
mean of x mean of y
   128.10    126.52
```

5. NEONISE

Neonise is an image processing filter that emphasises core edges in an image, once the major edges are located, contouring is applied and a set of thick neon coloured lines are traced on top. The resultant effect is similar to that seen on neon street signs. A strong emphasis is placed on a few fundamental lines providing a high contrast. Neonise employs contour detection, and much preprocessing of the image to pass simple fundamental features to simplify contour detection. This filter is used to test Ne^{10}'s influence on speeding up one of the most computationally intensive operations in OpenCV (contour detection), as well as benchmarking multiple passes of image morphology and histogram equalisations.

5.1. Mathematical Theory

The Neonise function applies a chain of image operations: greyscale conversion, histogram equalisation, maximum and minimum intensity determination, binary thresholding, closing operations, Gaussian smoothing, and finally contour detection. Contour detection can only be applied after all the pre-processing functions have been applied. As greyscale conversion and Gaussian smoothing have been discussed in Section 4.1.4 they will not be discussed further.

5.1.1. Histogram Equalisation

Histogram equalisation is a method where an image contrast can be automatically adjusted. The generic formula for contrast enhancement is:

$$h(v) = \text{round}\left\{\frac{cdf(v) - cdf_{\min}}{(N_x \times N_y) - cdf_{\min}} \times (L-1)\right\} \quad (10)$$

where: $cdf_{\min}$ is the minimum value of the cumulative distribution function, $N_x \times N_y$ determines the number of pixels in the image, L is the selected number of bins, or grey levels in the image, and $cdf(v)$ is the lookup of the cumulative distribution function at that value, i.e. the value 53 occurred 5 times; e.g. $cdf(53) = 5$.

5.1.2. Maximum and Minimum Determination

For the single channel (greyscale) intensity image $P_I(x, y)$ the maximum $m^\star$ and the minimum intensities m^* of all pixel values was determined using a sequential scan. These operations were called the maximum and minimum determination.

5.1.3. *Binary Thresholding*

Binary thresholding is a process that assigns each and every pixel value in a image the value zero or one depending on whether or not it is greater than or equal to a threshold, $T \epsilon \mathbb{R}$. It is defined mathematically as:

$$TH_I(x,y) = \begin{cases} 1 & \text{if} \quad P_I(x,y) > T \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

5.1.4. *Closing Operations*

A closing operation is useful when wishing to emphasise the image background, it is defined simply as a dilation (Eq. 5) operation followed by an erosion. (Eq. 4)

5.1.5. *Contour Detection*

Contour detection involves examining all components of an image and quantifying the presence of a boundary at a given location. It requires gradient calculations on brightness, colour and texture channels. The end result is a ragged array of vectors each describing a set of image locations through which each particular level set of a particular magnitude passes. The specific algorithm is quite complicated and extensive with further details provided by Arbelaez (2010) [1].

5.2. Computational Perspective

All of these operations are commonly supported by the OpenCV framework. Histogram equalisation is achieved with the function `cvEqualizeHist(src, dst);` where `src` and `dst` are of type `IplImage*`.

Maximum and Minimum determination of pixel values is found with the OpenCV call `cvMinMaxLoc(src, &minVal, &maxVal);`, where `src` is the image to examine of type `IplImage*`; `minVal` and `maxVal` are of type `int` and are set on the termination of the function call.

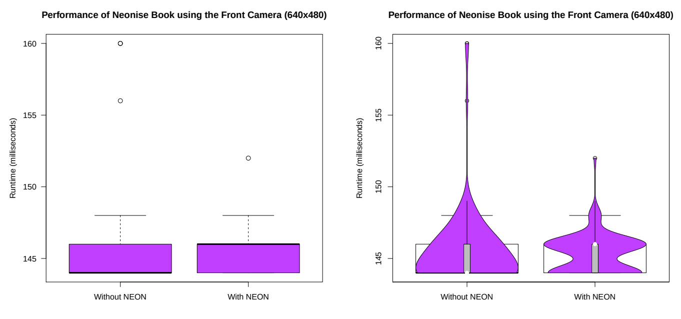
Binary thresholding is performed using the function `cvThreshold(src, dst, minT, maxT, CV_THRESH_BINARY);` where `src` and `dst` are of the type `IplImage*`, `minT` and `maxT` are `ints` used to determine the upper and lower intensities in which binary thresholding will be performed. The `CV_THRESH_BINARY` flag is an `int` value used specify the exact thresholding operation.

The closing operation requires a shape/kernel to define what pixels it is to operate on; kernel generation was discussed in Section 3.2. Once the kernel has been generated, a closing morphology operation is achieved with `cvMorphologyEx(src, dst, tmp, kernel, CV_MOP_CLOSE, it);`. Where `src` and `dst` are of type `IplImage*`, `tmp` is the memory allocated for temporary storage for the function to operate on (this article assumes `tmp = NULL`; the `CV_MOP_CLOSE` is a flag used to determine the image morphology operation, and `it` is an `int` that specifies the number of iterations desired (this article assumes one).

Contour detection is then performed on the preprocessed image, and is achieved with `cvFindContours(src, contour_storage, &contours, sizeof(CvContour), CV_RETR_LIST, CV_CHAIN_APPROX_SIMPLE);` function[2], where `src` is the source image of type `IplImage*`, `contour_storage` is a memory buffer used for the dynamic memory allocation of arrays of contours, it is of type `CvMemStorage*`; `contours` is a `CvSeq*` where the vector of contours are stored, `CV_RETR_LIST` determines the order in which the contours are stored, the `CV_CHAIN_APPROX_SIMPLE` flag is used compress horizontal, vertical, and diagonal segments of the contour, leaving only their ending points.

Finally drawing the contours is performed via `cvDrawContours(src, c, ins, outs, maxL, lineWidth, lineType);` where `src` is the image which will have contours drawn onto, it is of type `IplImage*`; `c` is an array of contours of type `cvSeq*`, `ins` and `outs` are different neon-like colours (in RGB colourspace), `maxL` an `int` used to set the max level, `lineWidth` is an `int` to specify the width of the line when performing a trace (default = 25), and `lineType` uses an `int` to determine line type (default = 8 'solid line'). The Neonise filter involves twice as many sweeps of the image than all other filters (of Sections (2.2, 3.2 and 4.2)) and we expect it to be the most computationally demanding of all the filters examined.

5.3. Runtime Results



(a) (b)

FIG. 8: The benchmarking runtimes of 100 iterations at resolution 640×480 (Samsung Galaxy S III Front camera):
(a) as a boxplot,
(b) as a violinplot.

Figure 8 displays the results for the benchmarks for the Neonise filter using the low-resolution image data set. The numerical summaries were as follows:
`Mean without Neon` = 145.48 ± 2.706072 milliseconds.
`Mean with Neon` = 145.28 ± 1.378625 milliseconds.
These results suggest that there is little difference between processing time on average of the Neonise filter either with or without Ne^{10}. Note though that the variation in processing time using Ne^{10} appears to be 5.33 milliseconds (as the difference in standard deviations is 1.3 milliseconds) less. In other words Ne^{10} can consistently help keep the run times down when using this filter. Despite these apparent differences a non-parametric Fligner test produced a p-value of 0.3462 providing no evidence of a difference in the variances between the two groups. In contrast a parametric Bartlett test produced a p-value of 1.028 e-10, providing strong evidence of a difference in the variances between these two groups. The difference can be reconciled by the fact that the Bartlett test assumes normally distributed data. Shapiro-Wilk tests of normality indicated that this assumption did not hold (p = 2.3e-16 for not Ne^{10}, p = 2.4e-12 for Ne^{10}). Therefore the evidence suggests that the results of the Fligner test are the most reliable, it also suggests that the apparent differences in standard deviations are produced by another factor such as the Ne^{10} assisted filter having a less chance of taking an extremely long time to process. This hypothesis is in fact supported by the violin plot of Figure 8(b) where it is evident that without Ne^{10} assisted processing there are a greater number of runs that exceed the upper whisker of the plot.

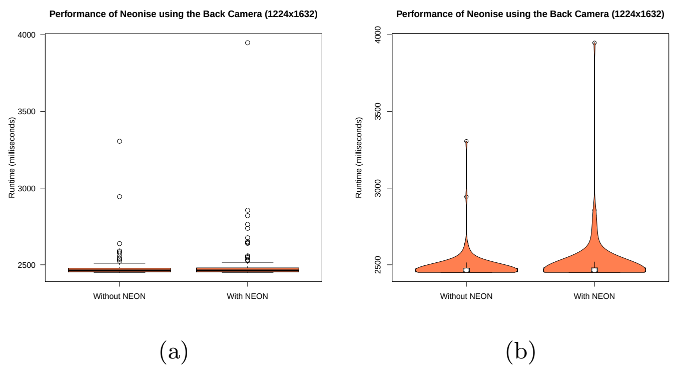


FIG. 9: The benchmarking runtimes of 100 iterations at resolution 1224×1632 (Samsung Galaxy S III Back camera) :
(a) as a boxplot,
(b) as a violinplot.

The Neonise filter was also tested on the high resolution camera data. The results are displayed in Figure 9. In these plots extreme 'heavy-tail' or 'skewed' statistical behaviour is observed. In this case though it appears that the processing with Ne^{10} can produce the longest run times. The summary statistics for this trial were:
`Mean without Neon` = 2486.66 ± 100.6548 milliseconds.
`Mean with Neon` = 2503.28 ± 164.6483 milliseconds.
There appears to be a substantial difference in mean processing times (Ne^{10} is 16.62 milliseconds slower) but also a difference in the standard deviations (Ne^{10} is 63.9935 milliseconds greater). Since the uncertainties about the mean values overlap substantially there is unlikely to be a true difference in mean processing times to be certain a non-parametric Kolmogorov-Smirnov test was performed (this test being selected due to the heavy-tailed nature of the data). This test also allows us to detect any other differences in the statistical characteristics of this data such as differences in the standard deviations. The results were:

```
        Two-sample Kolmogorov-Smirnov test

data:  WO and WN
D = 0.07, p-value = 0.967
alternative hypothesis: two-sided
```

These results indicated that the two records are likely to belong to the same statistical distribution. That is both the means and the standard deviations between the processing times of with and without Neonise are the same. There is no difference. It is important to note that there was one extremely long processing time for the Ne^{10} based processing (approximately 4000 milliseconds) which has greatly influenced these results. Without this particular record it is possible that conclusions similar to those found for the Neonise operation using low-resolution images would be found. It is likely that a much larger data set (on the order of 1000 benchmark tests) would be able to better ascertain the number and magnitude of these extremely long processing times and thereby allow more robust statistical comparisons.

6. CONCLUSION

When Ne^{10} is performing well there is only a slight statistical advantage when considering runtime performance. An interesting finding is that when Ne^{10} is applied to large heavy-weight operations, Ne^{10} does not necessarily make the task execute quicker but does make it more reliable. When used the standard deviation is significantly less than without using Ne^{10}, this only becomes truly apparent when benchmarking Ne^{10} on really computationally intensive tasks (such as contouring). Given the remarkably small variance that occurs when operating with Ne^{10}, heavy weight image processing operations which use Ne^{10} will consistently outperform the same operations without Ne^{10} utilisation. Ne^{10}'s smaller variance ensures many iterations will run with fewer outliers when compared against a non-Ne^{10} implementation. An obstacle that interferes with making this observation, was given the significant outlier detected during the Neonise filter benchmarking. Upon further analysis (such as removing that sole outlier), we find that the run times are almost equivalent. Much more benchmarking is needed to truly understand how Ne^{10} can be used to stabilise heavyweight image processing operations. A deeper statistical study examining the variance of runtimes between with and without Ne^{10} would be beneficial. A full list of runtime results are shown in Table I.

Another key finding from this report is that while Ne^{10} is used to automatically speed up end user code, when employed on a tremendously refined library such as OpenCV this speed up ranges from negligible to non-existent. We can postulate that Ne^{10} and auto-vectorisation will speed up generic code produced by a regular developer where there is much that auto-vectorisation and Ne^{10} can operate upon; however when dealing with a library that is almost entirely composed of heavily optimised C, as is the case with OpenCV, Ne^{10}s contributions are negligible.

The visual results regarding how the filters act on the test image 1 are shown in Fig. 10, 11, 12 and 13. From this we note that all filters visually perform as expected; these results are identical regardless of Ne^{10} involvement and that the images are processed uniformly regardless of resolution.

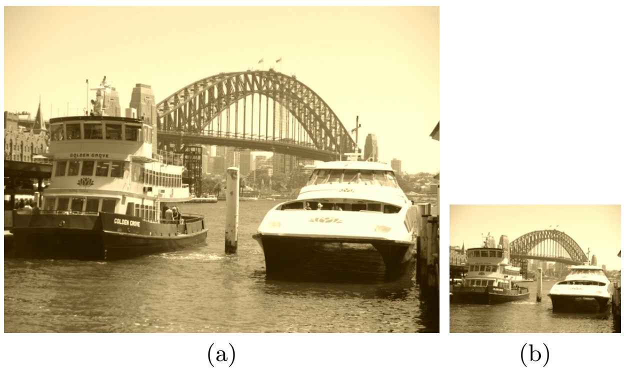


FIG. 10: The benchmarking test images with Sepia Tone at resolutions:
(a) 1632×1224 (Samsung Galaxy S III Rear camera)
(b) 640×480 (Samsung Galaxy S III Front camera).

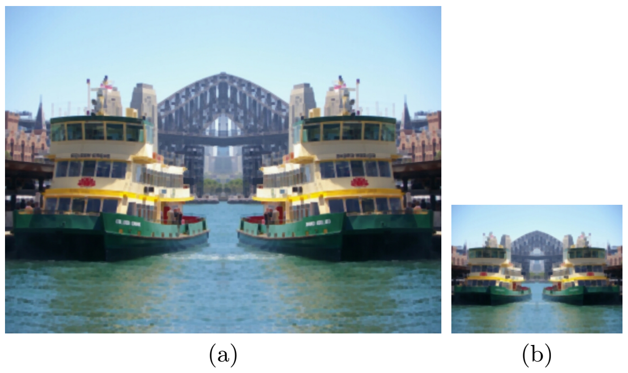


FIG. 11: The benchmarking test images with Fun House effects at resolutions:
(a) 1632×1224 (Samsung Galaxy S III Rear camera)
(b) 640×480 (Samsung Galaxy S III Front camera).

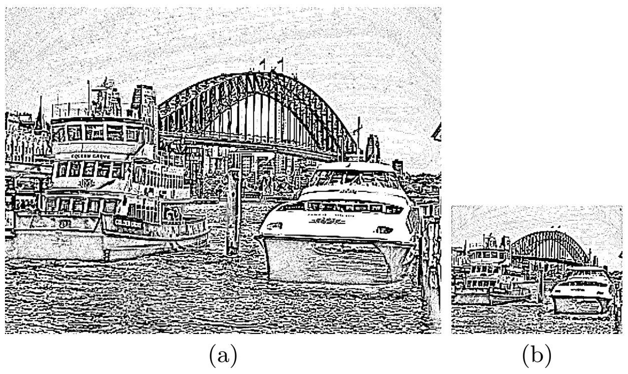


FIG. 12: The benchmarking test images with Sketch Book effect at resolutions:
(a) 1632×1224 (Samsung Galaxy S III Rear camera)
(b) 640×480 (Samsung Galaxy S III Front camera).

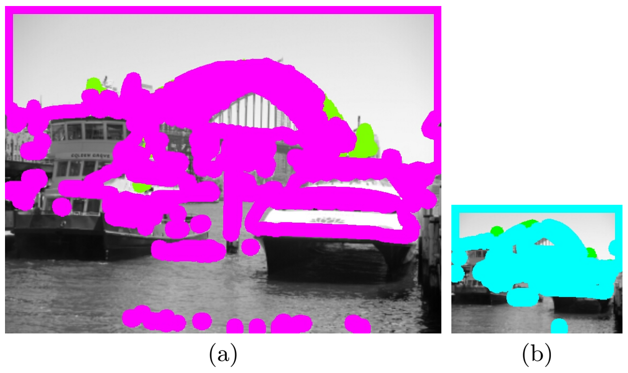


FIG. 13: The benchmarking test images with Neonise filter at resolutions:
(a) 1632×1224 (Samsung Galaxy S III Rear camera)
(b) 640×480 (Samsung Galaxy S III Front camera).

Total average runtimes (milliseconds) on Low Res(640×480)		
Filter	Without Ne^{10}	With Ne^{10}
Sepia Tone	39.78 ± 2.01	39 ± 2.04
Fun House	17.08 ±1.28	16.62 ± 1.09
Sketch Book	18.4 ± 1.48	18.78 ± 1.33
Neonise	145.48 ± 2.71	145.28 ± 1.38

Total average runtimes (milliseconds) on High Res(1632×1224)		
Filter	Without Ne^{10}	With Ne^{10}
Sepia Tone	244.52 ± 2.94	257.26 ± 2.18
Fun House	113.26 ± 4.28	111.68 ± 3.6
Sketch Book	128.1 ± 4.49	126.52 ± 3.37
Neonise	2486.66 ± 100.66	2503.28 ± 164.65

TABLE I: Runtimes of applying various OpenCV based image filters on a high and low resolution test image

7. DISCUSSION & FURTHER CONSIDERATIONS

There is still significant optimisations that Ne^{10} can provide to increase the runtime performance with OpenCV. Possible plans of action to allow OpenCV to be more compatible with Ne^{10} are to:

1. Use normalised images for all image processing, this would force OpenCV to use float*. All of Ne^{10} optimisation occurs when dealing with `float*`.

2. Modify the Core OpenCV functions and data objects, such as adapting the `IplImage* struct` to implicitly use `float*` instead of `int*`. This would ensure all OpenCV operations would use Ne10 optimisations.

3. Employ thread based image processing functions. These could use select OpenCV functions and explicitly use Ne^{10} calls when doing operations such as vector arithmetic.

It would also be advantageous for further statistical research into the stability (reduction of variance) that Ne^{10} offers, larger benchmarks should be run (in the order of 1,000's to 10,000's iterations) to help identify the cause and frequency of these sporadic immense outliers prevalent with Ne^{10} usage.

8. SOURCE CODE

Both Ne^{10} enabled and regular Android NeSnap.apk files are freely available for download https://github.com/openparallel/NeSnap/tree/master/ .

All source code can be found here https://github.com/openparallel/Ne10BoostedImageThresh. To use Ne^{10} from source, uncomment `//#define USINGNEON` in `ImageProcessor.h`. To collect 100 iterations of runtimes, set `benchmarking = true` in `ImageThreshActivity.java` located at line number 385. The runtimes will be saved to /sdcard/runtimes.txt.

[1] Arbelaez, P., Maire, M., Fowlkes, C. & Malik, J. (2010). *Contour Detection and Hierarchical Image Segmentation* Berkeley, California, USA: University of California.

[2] Bradski, G., & Kaehler, A. (2008). *Learning OpenCV* Sebastopol, California, USA: OReilly Media, Inc.

[3] Russ, J. (2011). *The Image Processing Handbook (6th Edition)* Boca Raton, Florida, USA: CRC Press

[4] ARM. (2008). *ARM NEON support in the ARM compiler* http://www.arm.com/files/pdf/neon_support_in_the_arm_compiler.pdf