Speedy Tilt Shift Acceleration

by

Raunak [redacted]
rraunak@asu.edu

Viraj Savaliya [redacted]
vsavaliy@asu.edu

Srihari Danduri [redacted]
ddanduri@asu.edu

Project 1 Report for EEE 598: Mobile Systems Architecture

Master of Science in Computer Engineering (Electrical Engineering)

Course Instructor:

Dr. Robert LiKamWa

ARIZONA STATE UNIVERSITY

October 2020

ACKNOWLEDGMENTS

We would like to thank Dr. Robert LiKamWa for providing us the opportunity to work on the project. Also, we would like to thank him again for conducting lectures on important topics in Mobile Systems like Instructions Set Architecture, Memory Management, Power Management, Android Runtime, Sensor Processing, hierarchy and abstractions in systems, etc. These topics have helped us to get insight on various aspects of Mobile Systems which is currently on a rise and dominating in most sectors of Industry.

TABLE OF CONTENTS

Declaration of Honor Code

We, the students of Arizona State University, adopt this code as an affirmation of our commitment to academic integrity and our participation in ethical education.

We embrace our duty to uphold ASU's Honor Code, and in light of that duty,

We promise to refrain from academic dishonesty.

We pledge to act with integrity and honesty and to promote these values among our peers. We agree to always abide by the Sun Devil Way and uphold the values of the New American University."

The undersigned acknowledge and certify that all the tasks completed for the current stage of the project are within the ASU honor code and are our own work and with equal contributions from each team member.

Viraj Savaliya	Raunak	Danduri Srihari
Member 1	Member 2	Member 3

PREFACE

This report is about the implementation of tilt-shift blur on an android platform. The tilt-shift blur is implemented in Java, C++ and Neon and is benchmarked on three different images to determine the performance of each implementation. It is written for the first project/Report submission for EEE 598: Mobile Systems Architecture in Fall 2020 at Arizona State University. The report structure is based on the implementation algorithm followed as the requirement of the project and results observed to compare the performance of each implementation. Various online resources, textbooks as well as course notes were referred for understanding the subject matter and to form the detailed report for the topic. All the references which were used are mentioned at the end of the document and any content which is exactly taken from source is cited as well.

CHAPTER 1: GOALS

1.1 High-level Objectives

The following are the high-level objectives of the project we sought to accomplish for the project:

- Minimal computation time complexity for execution.
- Achieve a clear, smooth and distinct blurring effect on the processed image.
- To find out the system bottlenecks to performance in image processing.
- Finding out different optimization techniques to achieve fast tilt shift blurring effect.
- Understanding how an android app works and how it is implemented on the android studio platform.
- Understanding the various design manipulations or variations required in C++, Java and ARM Neon versions for the same implementation in all.
- How the pixels can be extracted, manipulated and stored back in the pixels array of the bitmap image.
- Understanding how the gaussian blur kernel is calculated, and how the gaussian vector is computed and what are the approaches for the gaussian blur.
- Understanding the effect of gaussian blur on images of different resolutions and sizes.

1.2 Assignment Description

The following is the re-stated description of the assignment statement from out point of view:

- Understanding Gaussian Blur.
- Learn how to create an Android app and use Android Studio.
- Figure out Android native and Java functions involved in Android and its working.
- Practice working with ARGB8888 representation of image.
- Perform Image processing on different image sizes and varying parameters and observe the differences.
- How the pixels can be extracted, manipulated and stored back in the pixels array of the bitmap image.
- Use Gaussian Blur to implement a tilt shift effect that simulates focal blurring depending on the assumed physical depth of subject in image.
- Understanding how the gaussian blur kernel is calculated, and how the gaussian vector is computed and what are the approaches for the gaussian blur.
- Achieve tilt shift blurring on an image with minimal computation time.
- Understanding Arm neon and assembly intrinsics for faster tilt shift blur implementation.
- Comparison and analysis of C++, Java and Neon implementations.
- Observe performance for images with different size and resolutions.

Chapter 2: DESIGN IMPLEMENTATION

For implementing tilt shift blur using java, c++, arm neon the following pseudo code is used for implementation steps. The pseudo code demonstrates general steps for all 3 implementations.

```
1. Load Input Pixels
2. Get height, width of Image
3. Get Sigma near and sigma far input
4. Get y values for gradient transition (a0,a1,a2,a3)
5.
6. Perform First Transformation:
7. //Iterate over all pixels
8. for j range(height)
9.    for i range(width)
10.                Calculate sigma based on height
11.         Calculate Gaussian Kernel Vector (Gk) based on sigma for each
   pixel (i,j)
12.         Calculate Pixel Vector for each pixel (i,j) from input pixels
13.         Multiply Pixel Vector and Gk to get intermediate Pixel values
   for each pixel (i,j)
14.  Store intermediate Pixels in intermediate Pixel Array
15.
16.  Perform Second Transformation:
17.  //Iterate over all pixels
18.  for j range(height)
19.     for i range(width)
20.                Calculate sigma based on height
21.                Calculate Gaussian Kernel Vector (Gk) based on sigma
   for each pixel (i,j)
22.                Calculate Pixel Vector for each pixel (i,j) from
   intermediate pixels
23.                Multiply Pixel Vector and Gk to get Final Pixel values
   for each pixel (i,j)
24.
25.  Store Final Pixel values in Output Pixel Array
```

Figure: Pseudo code for tilt-shift implementation

We implemented the gaussian blur through weight vector approach using the following code structure and the apis in c++ and similar approach is used in java and neon as well.

- Algorithm:

Each pixel value in the output image is obtained by applying a gaussian weighted sum on a patch of pixels and after undergoing two independent transformations across x-y dimensions.

2.1 Apis in the algorithm:

1. pixelval(jint y, jint x,jint* pixels,jint height,jint width)

    Given (y,x) coordinates of a pixel it returns the pixel value of the corresponding pixel. It takes care of boundary conditions of gaussian blur. If we try to access out of bound pixel value it returns zero i.e. black pixel.

2. Gk(jint k, jfloat sigma)

$$G(k) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{k^2}{2\sigma^2})$$

    Given k value and sigma it computes gk (gaussian kernel vector) value.

3. sigmacal(jint y,jfloat sigma_far,jfloat sigma_near,jint a0, jint a1,jint a2, jint a3)

    Given y value based on question requirement this api returns what value of sigma is needed to be used for gaussian blur.

4.finalPixelval(jint y, jint x,jint *pixels,jint height,jint width,jfloat sigma,jint flag)

    This api takes in parameters and depending on the flag it decides to compute whether the first transformation needs to be done or the second transformation and returns the final pixel value calculated.

5. Main api

    Main api calls sigmacal to get the required sigma value and then it calls finalPixelval api to get the pixel value after each transformation. Initially the flag is set to 0 to calculate intermediate pixels after one transformation and later the flag is made 1 for the second transformation to get final pixel values after applying the Gaussian Blur feature.

2.2 Neon Intrinsics

1. Loading and Storing vector (vld4q_u8, vst4q_u8)

   These functions in neon are used to load and store a vector consisting of size uint8x16x4 by providing a pointer to the location you want to load from or store to. The vector consists of 4 channels having 16 values each of 8 bit size which is used for ARGB values for 16 consecutive elements from the array.

2. Multiplication of Gaussian Kernel multiplier with the pixel value.

   The functions of Neon used over here are:

   a. vget_low_u8, vget_high_u8 - These are used to extract lower and higher bits 8 from the pixel vector so that they can be worked on individually and the overall value does not overflow.
   b. vmovl_u8 - This is used to convert the 8 bit vector to 16 bit for computations.
   c. vmulq_n_u16 - This is used to multiply an integer value to the vector.
   d. vshrq_n_u16 - This is used to right shift back (i.e divide) by the same amount as multiplied.
   e. vqmovn_u16 - This is used to push back the 8 bits after the computation to the original 8 bit vector.
   f. vcombine_u8 - This is used to combine back both the lower and higher bits of the vector.

CHAPTER 3: EVALUATION OF BENCHMARKS

The Java, C++ and Neon implementations of gaussian blur were tested on three images of different sizes with Image 3 being the largest and Image 2 being the smallest. We can see the comparative differences in time between the three images, image 3 takes the longest time to execute tilt-shift blur while image 2 takes the least amount of time to do the same.

Out of the three Java, C++ and Neon implementations, Java takes the longest to execute while neon is the fastest out of the three implementations. Java is an interpreted language, it needs a Java virtual machine to convert the source code into bytecode, then the code is compiled to native by the Java virtual machine. Whereas, C++ is written in the native library itself and it compiles to native code directly. It also allows a controlled memory allocation and includes lower memory footprint as it doesn't have garbage collection as compared to Java. Therefore, C++ implementation runs faster than Java.

Neon Implementation is also written in the native library, it follows ARM neon intrinsics. The ARM Neon intrinsics allows SIMD operations, which makes parallel multiple data operations possible. Thus, we are able to get more number computations in less time as compared to C++ implementation. That's why Neon implementation gives better performance than the C++ tilt-shift blur implementation.

Hence, Neon gives us the best performance while Java gives us the worst performance for the tilt shift blur implementation.

3.1 Performance Analysis

The Table below shows how the runtime performance for all the three implementations vary and also how an implementation varies for different images.

	Java	**C++**	**Neon**
Image 1	9.477s	6.7s	2.27s
Image 2	1.709s	1.263s	0.439s
Image 3	25.236s	17.992s	6.604s

Table: Comparison of Java, C++ and Neon implementation on all the 3 images.

The Table given above can be described more in detail using the below chart. The chart clearly depicts that the runtime for Neon is fastest amongst all implementations and image 3 takes when compared with other images takes the longest time to compute while image 2 is the fastest amongst the three. This observation is also true for all implementations. The change in the computation time for different images is due to the change in image size. Larger images are more pixelated and hence have more pixels to work and therefore the computation time for the tilt shift blur will be more. This is proven by the observations made in this project.

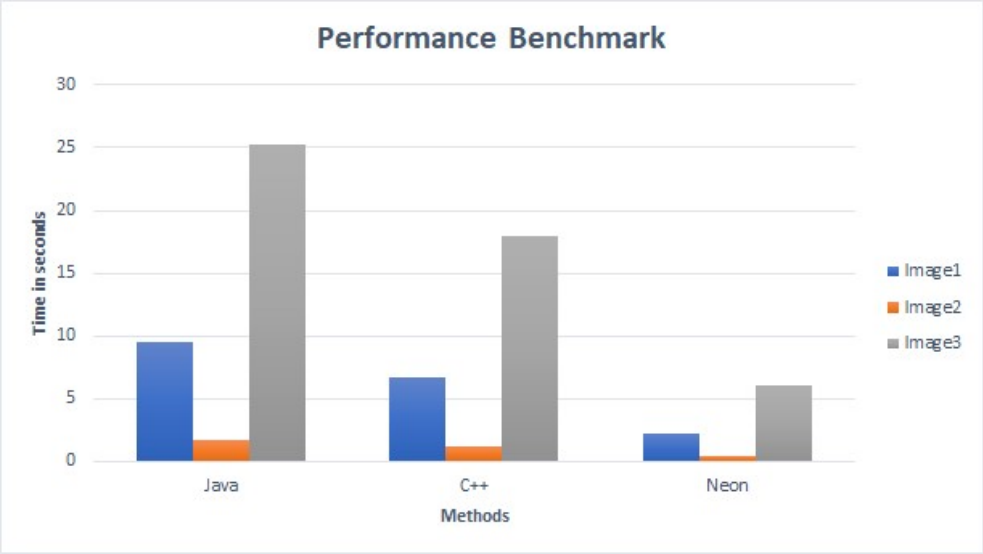

Figure: Bar chart showing performance of the 3 implemented methods.

CHAPTER 4: CHALLENGES FOR THE PROJECT

- Image was getting blurred at most segments but at few we observed weird distortion in the image. Here, we were missing the sigma value condition check of greater than 0.6 at some parts of the code.
- The tilt shift blurring effect was not visible at the edges of the image. Here the mistake made was to not consider the values outside the matrix as -1 or black. After taking in the values outside the matrix as -1 we were able to apply the tile shift on the entire image.
- In Java, the sigma values generated come out to be very low compared to c++. Hence, the tilt shift blurring effect is not as prominent as in c++ and at times not easily visible. To make the effect distinguishable when applied on an image the multiplication factor was made larger but the trade-off for which was that the computation time increased significantly. Currently, the challenge here is to find an optimization technique to reduce the computation time.
- At times, the application generated for c++ as well as java code was getting crashed when we computed the respective title shift blur feature on the image. This issue was solved by attaching the debugger and finding the error. The error over here was an Out of bound array case as the boundary conditions were miscalculated and hence the kernel tracing over the image went outside the boundaries for non-existent pixels. We redefined the boundary conditions to fix the issue.
- Initially, we were taking the gaussian kernel without normalization, which resulted in black stripes along the blur region of the image. Later, solved the issue by dividing each element of the kernel by the sum of all the kernels, which resulted in a total sum of the kernel to 1 and was able to get the correct blurring effect.
- At first, we started with the weight matrix approach of the gaussian blur, it was resulting in a lot of overhead and some pixels were getting skipped due to high computation time. Later, we improved on that issue by following the weight vector approach of the gaussian blur, which was much faster and efficient as compared to the weight matrix approach.
- In Neon, since we are computing 16 values of array at a time, managing out of bound values and choosing pixel vectors at near width and height conditions is challenging. This issue was solved by careful calculations of the edge conditions involved on the left and right width boundaries of the x axis as width needs proper attention and correct boundary condition as we are loading 16 values in row by providing a pointer to the first value of the required vector.

- While Multiplying float values to int pixels vector in Neon for the right amount of multiplication value needs to be selected for multiplier*value/value. Here the value is also right shifted in order to divide it later. Therefore the correct right shift amount in the range of 0 to 15 should be selected as it can affect the exposure of the image. At the end we selected 8 as the right shift amount and 2^8 for multiplying before as 8 is the center value in the range and shows the least change in exposure. We finalised the value after experimenting and playing around with different numbers.

- Another challenge we faced in neon was the saturation of the sum of all vectors while computing in the kernel. The saturation was not needed at times but only at the end of all computations in the vector.

- The brightness of the image was varied after the computations at times. This issue was solved by removing all computations done on A of the pixel which we by mistake added as we were doing computations for RGB.

REFERENCES

1. Ltd., A., 2020. *SIMD Isas | Neon – Arm Developer*. [online] Arm Developer. Available at: <https://developer.arm.com/architectures/instruction-sets/simd-isas/neon> [Accessed 17 October 2020].
2. En.wikipedia.org. 2020. *Gaussian Blur*. [online] Available at: <https://en.wikipedia.org/wiki/Gaussian_blur#:~:text=In%20image%20processing%2C%20a%20Gaussian,image%20noise%20and%20reduce%20detail.> [Accessed 17 October 2020].
3. SDK, F. and Favre, P., 2020. *Fast Bitmap Blur For Android SDK*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/2067955/fast-bitmap-blur-for-android-sdk> [Accessed 17 October 2020].
4. Android Developers. 2020. *Neon Support | Android NDK | Android Developers*. [online] Available at: <https://developer.android.com/ndk/guides/cpu-arm-neon> [Accessed 17 October 2020].
5. Arm.com. 2020. *Neon – Arm*. [online] Available at: <https://www.arm.com/why-arm/technologies/neon> [Accessed 17 October 2020].
6. Medium. 2020. *How To Blur An Image On Android*. [online] Available at: <https://medium.com/mobile-app-development-publication/blurring-image-algorithm-example-in-android-cec81911cd5e> [Accessed 17 October 2020].