CHALMERS


# Partial Image Decoding On The GPU For Mobile Web Browsers

*Master of Science Thesis in Computer Science - algorithms, languages and logic*

JIMMY MILLESON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, May 2014

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example apublisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Partial Image Decoding On The GPU For Mobile Web Browsers

JIMMY MILLESON

© JIMMY MILLESON, MAY 2014.

Examiner: ULF ASSARSSON

Chalmers University of Technology

University of Gothenburg

Department of Computer Science and Engineering

SE-412 96 Göteborg

Sweden

Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering

Göteborg, Sweden May 2014

## Abstract

This thesis will investigate how the GPU can be used to facilitate image decoding for mobile web browsers. In the mobile web browsers of today, the majority of computations are performed on the CPU, with only some parts offloaded to the GPU. This thesis will move computations from the CPU, by partially decoding JPEG images on the CPU, and offloading the rest of the process to the GPU. Both the decoding- and encoding process of the JPEG format will be investigated to determine which parts of the decoding process would be suited to be performed on the GPU.

The study will show that upsampling and color space transformation, which are the two last steps in the JPEG decoding process, are particularly suited for the GPU. The implementation will be done in the Opera web browser for Android. Opera for Android uses the Chromium back-end for handling decoding and rendering of images. In Chromium the Blink component handles image decoding and the Skia component handles rendering of images. The results of the implementation will, depending on how the encoding of the image was done, show memory savings of up to 62.5



## Sammanfattning

Denna rapport kommer undersöka hur GPUn kan användas för att avkoda bilder för mobila webbläsare. I dagens webbläsare så utförs majoriteten av arbetet på CPUn, där endast ett fåtal av beräkningarna är avlastade på GPUn. Denna thesis kommer att flytta beräkningar från CPUn, genom att endast partiellt avkoda JPEG bilder på CPUn, och avlasta resten av processen till GPUn. Både avkodning- och kodningsprocessen av JPEG formatet kommer att undersökas för att avgöra vilka delar av avkodningsprocessen som bättre skulle kunna utföras på GPUn.

Rapporten kommer visa att upsampling samt färg-transformation, vilket är de två sista stegen i JPEG avkodnings processen, är särskilt lämpade för GPUn. Implementationen kommer utföras i Operas Android webbläsare. Operas Android webbläsare använder sig av Chromium för avkodning och utritning av bilder. I Chromium så hanterar Blink komponenten bild avkoding och Skia komponenten hanterar utritning av bilder. Resultatet av implementationen i Chromium kommer, beroende hur bilden kodades vid komprimering, visa att det är möjligt att spara upp till 62.5

## Acknowledgements

I would like to thank Ulf Assarsson and Erik Sintorn for agreeing to be my examiner and supervisor despite their busy schedule. I also would like to thank Opera for giving me the opportunity to do my thesis for them, and a special thanks to Erik Möller and Peter Pettersson for helping me along the way.

Jimmy Milleson

Gothenburg, 15 May 2014

# Table of Contents

# 1

# Introduction

## 1.1 Background

The web has evolved tremendously in the last few years [Google Inc. 2012]. This can especially be seen with advance of mobile devices, where the browser has gone from only being able to handle very basic web-sites to being as capable as a desktop browser (see Figure 1.1). While the hardware of mobile devices has become more advanced at a fast pace, the majority of mobile devices out on the market today are not very capable. According to [Gartner and Gartner 2013], feature phones[1] still account for roughly half the cell phones sold in the second quarter of 2013.

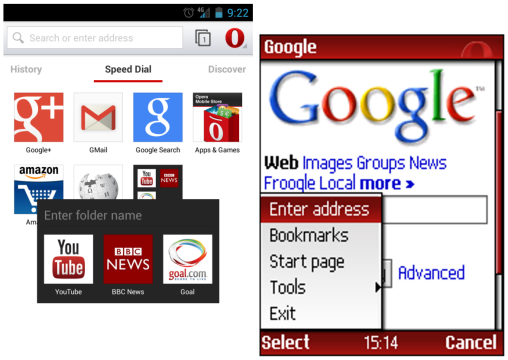



**Figure 1.1: Left:** Opera for Android. Image taken in 2014 from *http://www.opera.com/mobile/android*. **Right:** Opera mini. Image taken in 2006 from *http://mobhappy.com/blog1/2006/01/16/the-best-in-mobile-2005/*.

Opera is a company that was founded in the belief that any device should be able to experience the internet [Opera Software 2014a], from the latest and greatest all the way down to feature phones. Since Opera exists on such a varied amount of devices, with different hardware and software, it is important that the browser is capable on each and every device. To achieve this, Opera focus on creating optimized browsers that are capable on each and every device, with features like Off-road [Opera Software 2014c] mode and Texture compression [Pettersson 2013].

Opera have multiple different products for mobile [Opera Software 2014b], both for the Android platform and iOS platform. The most popular Opera browsers for Android are "Opera for Android" and "Opera mini". The Opera browser for Android uses the Google Chrome back-end, Chromium, to power their browser. Opera Mini on the other hand is a light weight browser with more than 250 million users, where most of the users are on feature phones.

Images are an integral part of the web and account for roughly 65% of the average bytes per page [Google et al. 2014]. In the last few years, new image formats have been introduced, see for example WebP [Google Inc. 2014a], and the amount of images that are displayed on the web have increased substantially the last few years with sites like Facebook, Reddit and Instagram. According to SEC [United States Securities And Exchange Comission 2012], as of late 2011, more than 6 billion images per month were uploaded to Facebook. Even with new formats like WebP available, the most popular image formats on the web are JPEG and PNG, which both date back to early 1990s. Combined, they accounts for roughly 75% of the images consumed on the web, where JPEG alone account for 47 % (see Figure 1.2). According to [Sodsonget al. 2013], 463 out of the 500 most popular websites contain JPEG images [Alexa 2014]. Because of this, the decoding of the JPEG format will be the focus of this thesis.

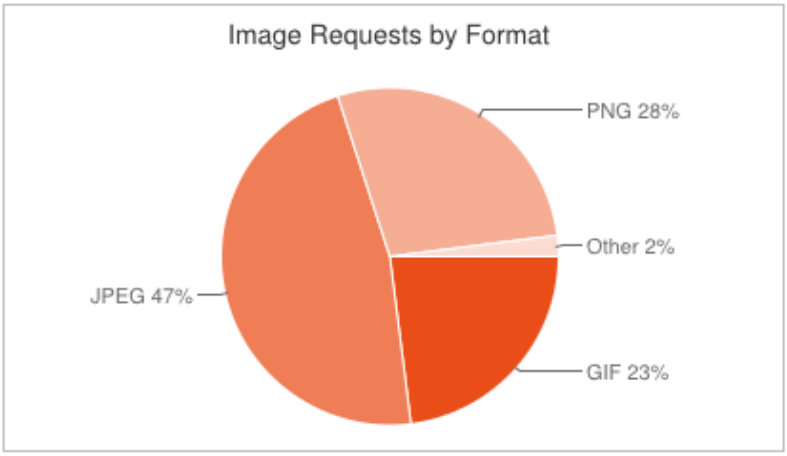



**Figure 1.2:** Chart shows which image formats are the most common by image requests. Chart taken from httparchive.org the 15th of April 2014.

[1] A phone which is not a smartphone, but is capable of running applications in some form (for example Opera Mine for J2ME).

## 1.2 Purpose

This thesis will investigate if it is possible to take advantage of the GPU, on mobile devices, to decode images efficiently. If the investigation shows that using the GPU is indeed efficient, the methods used to perform the decoding on the GPU will be implemented into the Opera browser for Android.

## 1.3 Problem

The JPEG encoding and decoding process will be fourthly investigated to determine which parts of the decoding process are most suitable for the GPU.

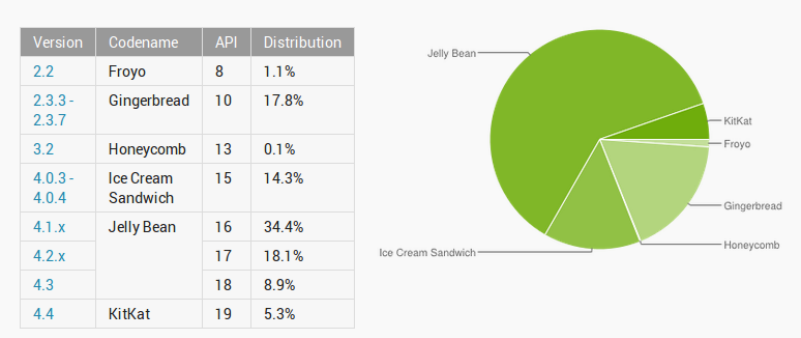
**The key focus and ultimate goal of this thesis is to:**

- see if it is possible to more efficiently decode images on the GPU compared to the CPU.
- implementing GPU decoding in Chromium.
- reduce:
  - memory usage on the CPU and GPU.
  - CPU load.
  - power usage.

## 1.4 Method

A standalone application will be created with the bare minimum features to be able to decode and display a JPEG image. Various different ways of decoding and rendering the image will be tried to determine if decoding on the GPU is possible and which methods produce the best results.

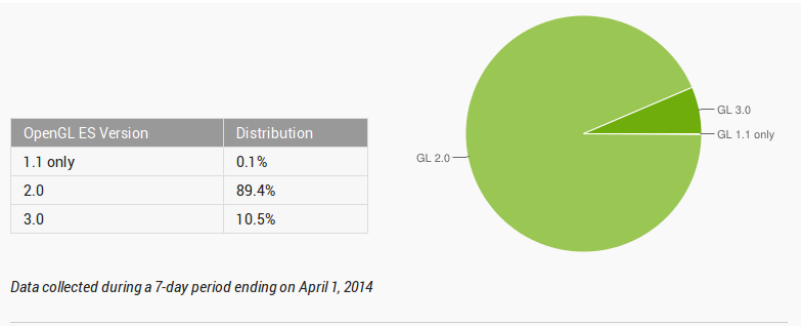
Since Android will be the platform used for this project, OpenGL ES 2.0 will be used for programming the GPU. OpenGL ES [Munshi et al. 2008] is an API that is used to interact with the GPU. The most current version of this API for Android is OpenGL ES 3.0, but this version is only available Android version 4.3 or higher. Therefore, OpenGL 2.0 will be the preferred choice (see Figure 1.3 and Figure 1.4). By using OpenGL ES 2.0 more than 99 % of Android devices are accounted for.



| Version | Codename | API | Distribution |
|---|---|---|---|
| 2.2 | Froyo | 8 | 1.1% |
| 2.3.3 - 2.3.7 | Gingerbread | 10 | 17.8% |
| 3.2 | Honeycomb | 13 | 0.1% |
| 4.0.3 - 4.0.4 | Ice Cream Sandwich | 15 | 14.3% |
| 4.1.x | Jelly Bean | 16 | 34.4% |
| 4.2.x | | 17 | 18.1% |
| 4.3 | | 18 | 8.9% |
| 4.4 | KitKat | 19 | 5.3% |



**Figure 1.3:** The chart shows how the different android versions are distributed. Chart was taken from developer.android.com the 15th of April 2014.



| OpenGL ES Version | Distribution |
|---|---|
| 1.1 only | 0.1% |
| 2.0 | 89.4% |
| 3.0 | 10.5% |

*Data collected during a 7-day period ending on April 1, 2014*



**Figure 1.4:** The graph shows how many of the devices support the different OpenGL ES versions. Chart was taken from developer.android.com the 15th of April 2014.

Once decoding on the GPU has been determined to be possible, memory usage, CPU load, and power usage will be measured. During the benchmarking period the work on implementing the method in Chromium will be started. This process will take up most of the thesis, since the code base for Chromium is huge and adapting already optimized code can take time.

## 1.5 Limitations

On the web today, there exist three major image formats (See Figure 1.2), and providing an implementation for each of them is not feasible in the amount of time that is available. Therefore this thesis will mainly focus on the decoding of JPEG images. The reasoning behind choosing JPEG as the format is that it is the most used image format on the web, and the fact that some parts of the JPEG decoding process are suitable for the GPU.

## 1.6 Outline

Section 2 of the thesis will cover digital image compression, which will build up the foundation for understanding JPEG compression. In this section, JPEG (2.2), PNG (2.3), and WebP (2.4) will be described, with focus on the JPEG format since it is the format that will be used for decoding on the GPU. Section 3 will focus on explaining the GPU pipeline (3.1), GPGPU (3.2) and OpenGL (3.1.1) in a way that is relevant to image decoding. Next in Section 4, the Chromium components, Blink (4.1) and Skia (4.2), will be described with focus on decoding and rendering of images. Parallel decoding will be discussed in Section 5, which will go in depth on some methods used to decode parts of the JPEG process on the GPU. In Section 6 the standalone implementation of decoding the JPEG image on the GPU will be given. Then in Section 7 the actual implementation of the GPU decoder in Chromium will be described. Next, Section 8 will discuss benchmarking and results from Section 6 and 7. Section 9 will be a discussion about the findings and possible future work that can be done. Lastly, Section 10 will be a conclusion of the thesis.

# 2

# Digital Image Compression

The amount of images transferred over the web, on average, stands for more than 60%, see Figure 2.1, of the total web content transferred. The amount of storage and transfer needed for this data is huge. This is only possible because the images are compressed, without compression storage and transfer of this amount of data would not be feasible.

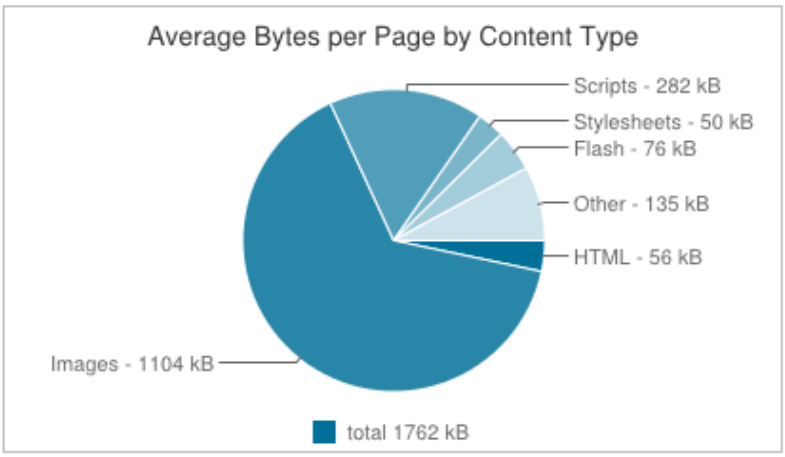



**Figure 2.1:** This chart shows the distribution of the formats for a average page load. That is, how much of the page data is of a certain format. The chart was taken from httparchive.org the 15th of April 2014.

An image with the dimensions 2832*2832 pixels[2], which in today's standards isn't that big[3], would, uncompressed, require around 32 Mb[4] of storage. Imagine a web page containing this single image, then an average internet connection in Sweden (~890 KB/s) [TestMy Net LLC 2014] would require

more than 32 seconds just to download the image. When storing and transferring images over the web, it is necessary to compress them in order to make the web sustainable.

In the field of digital image compression, researchers study the various ways to reduce the amount of data that is needed to represent an image. The main way, of reducing the size of an image, is by reducing the amount of redundancy in the image [Li and Kim 2000]. There are two distinct categories an image compression algorithm can be placed in : either lossless or lossy. Lossless is when the original image can be reproduced from a compressed image without any loss of information. But the amount of information that can be reduced in a lossless format is finite [Gonzalez and Woods 2001]. In lossy compression it is not required that the exact original image should be obtainable from the compressed image. The amount of compression can be varied against the quality of the image; the higher the compression the more the quality will suffer. Images that are rich in redundancies can be compressed more than those who are not.

Lossless and lossy compression are both useful in different scenarios. Lossless images are often used in scientific, medicinal [Yang and Welch 2002], and other fields where the exact recovery of an image is necessary. A lossy format is often preferred over a lossless format when transfer speed is of importance because of the additional compression. On the Internet today, the majority of images are compressed using lossy image compression algorithms (see JPEG in Figure 1.2)

## 2.1 Redundancy and Irrelevancy

Most images contain redundant information. Imagine an image with a clear blue sky, most pixels in the image can be described by the same color. In compression, the aim is to remove all redundancy from the original image. There are multiple forms of redundancy:

- Coding redundancy, exploit that some pixel values are more common than others.
- Interpixel redundancy, pixels close to each other have similar values.
- Psychovisual redundancy, exploit the weakness in the human eye.

An uncompressed image is usually coded in fixed length , i.e each pixel in the image is represented by the same amount of bits. Using methods like Huffman encoding (2.2.9) [Cormen et al. 2001] code-words are used with varying length to losslessly represent the image data. If a value in the stream occurs in multiple places, the value in all the places may be exchanged into a code-word which can be represented by fewer bits. The more a value occurs in the data the fewer the bits the code-word will have for representing the value. A table will then be used to find the original value when decompressing.

[2] A pixel describes the color of one point in the image. An image is made up of an array of pixels.

[3] 2832*2838 resolution represents a 8 megapixel camera, which in today's standard is pretty common.

[4] 2832*2838*(4 bytes per pixel)/1000000 ~= 32 Mb.

Interpixel redundancy can be defined as the "failure to identify and take advantage of data relationships". Informally interpixel redundancy is when the neighboring pixels of some pixel can accurately predict the value of said pixel. The image resolution usually correlates with how much interpixel redundancy an image contains, since the probability that neighboring pixels are similar is higher.

The last form of redundancy is slightly different than the previous two. While coding- and interpixel redundancy take advantage of the data representing the image, psychovisual redundancy takes advantage of the visual defects of the human eye. This stems from the observation that the human eye doesn't respond to all visual information in the same way. The human eye constructs an image differently than a computer screen. The eyes search for distinct features and then constructs something called recognizable groupings. When constructing these groupings certain information is not as important, and is referred to as psychovisually redundant. This process is often referred to as Quantization, later described in JPEG - Quantization. This reduction is dependent on that the recipient has human eyes. If the image is intended to be viewed by computer, to for example detect edges in the image, the loss of information may make this operation more difficult.

A lot of information in the image can be removed since the human eye cannot pick up the abundance of information. This operation can only be performed in a lossy environment since the information that is discarded cannot be recovered. Research has shown that small changes in hue are less noticeable than changes in luminance[5] [Smith 1997]. It is also known that low frequency changes in images are more noticeable than changes in high frequencies. Low frequencies usually describe the smooth variations in images while the high frequencies describe fine details like edges.

There are three types of irrelevance redundancy:

- Spatial redundancy, correlation between neighboring pixels in an image.
- Spectral redundancy, correlation between color planes in an image.
- Temporal redundancy, correlation between neighboring frames in a sequence.

Spatial, and spectral, redundancy is applicable to still images, while temporal is applicable to animated images and video.

Image compression is fundamentally different from compression of for example text data. This is mainly because the statistical properties of an image can be exploited. Text compression is always lossless, since disposing of information may most likely lead to undesirable consequences; while in image compression it is possible to throw away information at the cost of quality.

[5]Luminance is the brightness in the image, similar to a gray-scale image.

## 2.2 JPEG

JPEG stands for the "Joint Photographic Experts Group", which is what the group that created the standard was called. The JPEG standard defines how an image, an array of pixels where each pixel is a fixed amount of bytes, can be compressed and decompressed [Wallace 1991]. The JPEG format is designed to handle still images, but a format similar to JPEG, called MPEG, can be used for video. JPEG can compress both color- and gray-scale images, and is suitable for most images with the exception of images with lettering and images with fine detail. In those cases a lossless format like PNG (2.3) is more suitable.

JPEG supports the truecolor and grayscale image types. Grayscale images describe each pixel with a single component. Truecolor use multiple channels to describe each pixel. A channel that is not supported in JPEG is the alpha channel, which can be used in other image formats, like PNG, for transparency.

To achieve good compression JPEG utilizes the fact that the intended viewer of the image is a human. With this fact in mind, JPEG can optimize the compression mechanism to make use of the inherent limitations of the human eye. The human eye can more easily perceive changes in the brightness of an image than the change in color. This loss of information is what makes JPEG a "lossy" image format.

### 2.2.1 JPEG Compression

JPEG encoding achieves good compression by discarding information from the image. Depending on how much the image is compressed, most human eyes will not be disrupted by the loss of information. Lossy compression in JPEG follows the general algorithm that can be seen in Figure 2.2.

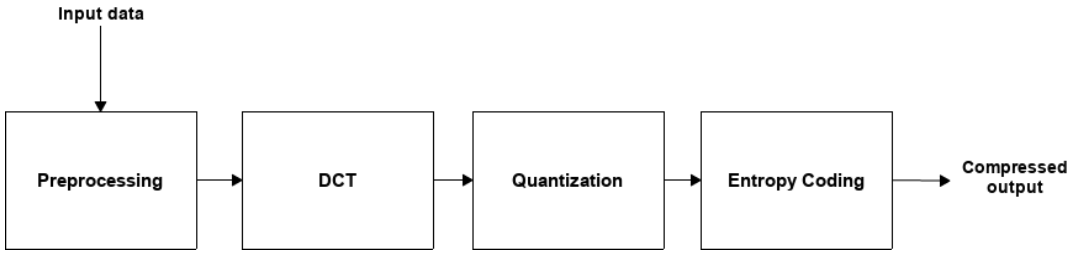



**Figure 2.2:** General compression algorithm for lossy compression.

The steps will be explained in detail after this short overview:

1. Pre-processing (2.2.2): Change color space from RGB to YUV.

2. Pre-processing (2.2.3): Reduce, often referred to as subsample, the size of the chrominance components. This step is sometimes skipped.
3. Pre-processing (2.2.4): Segment the image into blocks of fixed size, which depends on the subsampling selected in step 2. If no subsampling is done the blocks will be of size 8x8.
4. For each block starting with the top left block and ending with the block on the bottom right, perform:
   a) DCT (2.2.5): Apply the DCT on the block, transforming from spatial domain to frequency domain.
   b) Quantization (2.2.6): Divide block by matrix and round the resulting values in the matrix to integers.
   c) Entropy coding (2.2.7): Perform zigzag scanning to the block, which will structure the block to prepare for next step.
   d) Entropy coding (2.2.8): Apply run-length encoding, which will remove redundancy and prepare for the next step.
   e) Entropy coding (2.2.9): Run Huffman encoding, which will reduce the required bytes for each value.

### 2.2.2 Color Space Conversion

The first step in the encoding process is the conversion from one color space to another. The input data to the encoding process will, if the image is in color, contain three bytes per pixel, each representing a color component. These components are red (R), green (G), blue (B), which often just is referred to as RGB. For each pixel, 24 bytes will be required to describe the color. As the human eye has difficulties perceiving changes in the hue the color space needs to be adjusted to more easily enable the algorithm to make use of this human error.

The color space that will be used is the Y'CbCr space (but often referred to as YUV). The luminance (Y) is the brightness intensity for each pixel, essentially a gray-scale image, and the two chrominance parts represent the blue intensity (U) and the red intensity (V). The process of transforming from RGB to YUV can be described with the following equations:

- Y = 0.299 * R + 0.587 * G + 0.114 * B
- U = (B - Y) / (2 - 2*0.114)
- V = (R - Y) / (2 - 2*0.587)

[6]Depending on the image it is possible to achieve different ratios of compression. The size can further be compressed by selecting how much of the data should be discarded.

A visual representation is shown in Figure 2.3.This process will not reduce the size of the image. It will instead transform the data make it easier to discard the irrelevant information.

**Figure 2.3: Left** Image in RGB format **Second from left** Luminance (Y) component of image **Second from right** Blue chrominance (Cb/U) component **Right** Red chrominance (Cr/V). Image taken from `http://en.wikipedia.org/wiki/File:Barns_grand_tetons_YCbCr_separation.jpg`, by Mike1024.

### 2.2.3 Subsampling of chrominance

Now that the image is in the YUV color space the limitations in the human eye can be exploited. As previously stated, human eyes are more sensitive to changes to luminance than chrominance. This fact can be exploited. Chrominance subsampling is the operation of discarding detail in the chrominance components and still keeping an acceptable final image.

In JPEG, subsampling is described as a three part ratio, J:a:b. The parts are:

- **J**: Horizontal sampling reference.
- **a**: Number of chrominance samples in the first row of J pixels.
- **b**: Number of chrominance samples in the second row of J pixels.

The four most common ways of performing the subsampling on the chrominance are:

- **4:4:4** - No subsampling of either horizontal or vertical resolution.
- **4:4:0** - Half the vertical resolution of the U and V components are discarded.
- **4:2:2** - Half the horizontal resolution of the U and V components are discarded.
- **4:2:0** - Half the horizontal and vertical resolution of the U and V components are discarded.

Figure 2.4 demonstrates visually how each chrominance maps to the luminance. This is the first lossy operation, if 4:4:4 *subsampling* is not selected, in the algorithm.

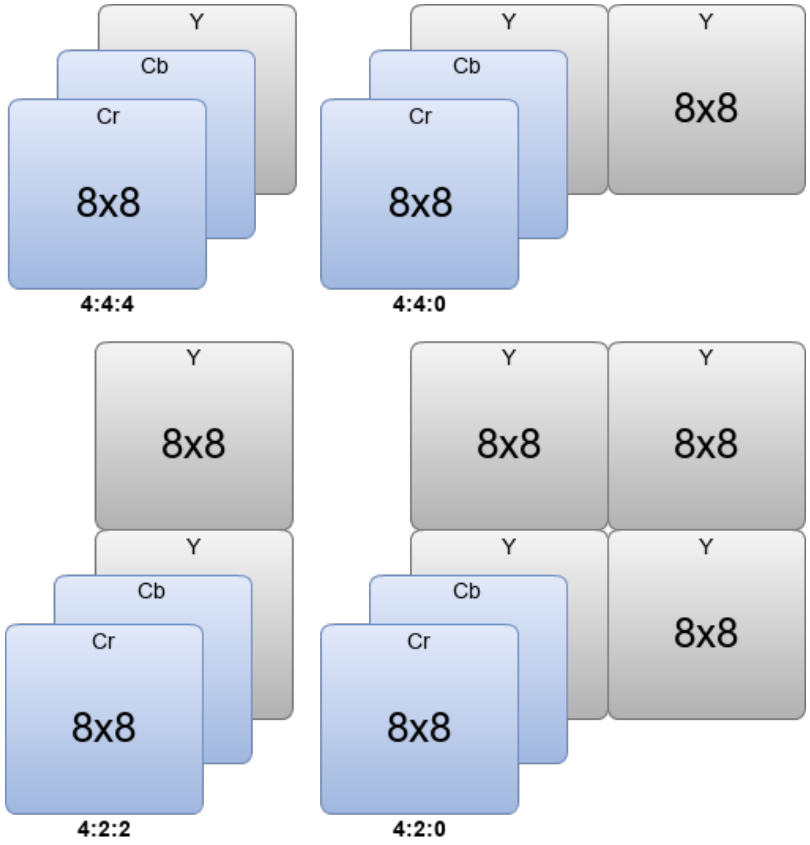



**Figure 2.4:** Show how the chrominance, after subsampling, map to the luminance. For example in 4:2:2 where the horizontal resolution in the chrominance is halved, two 8x8 chrominance blocks will map to a 8x16 luminance block. When the upsampling is performed, these three blocks will be used together to create a 8x16 RGB block.

### 2.2.4 Segmentation Into Blocks

This step will split the image into blocks of a fixed size. Each block is referred to as a Minimum Codec Unit (MCU), and the reason for this split is to make the next steps easier. The block sizes varies depending on how the image is subsampled, i.e:

- **4:4:4** - 8 x 8 block.
- **4:4:0** - 8 x 16 block.
- **4:2:2** - 16 x 8 block.
- **4:2:0** - 16 x 16 block.

Since the image will be divided into blocks of fixed size, the image dimensions must be dividable by the block size. If this is not the case, the image will have to be padded with extra pixels until the dimension is a factor of the block size. The extra pixels will often repeat value of the edge pixels to avoid artifacts. Next, each value will also need to be centered around zero. Each component uses values ranging from 0 to 255, so each value needs to be offset by subtracting 128. This step is necessary to reduce the dynamic range of the values in the following step.

### 2.2.5 Discrete Cosine Transform

After creating the MCU blocks, each block will be converted from the spatial-domain[7] into a frequency-domain. This transformation is called the Direct Cosine Transformation (DCT) [Ahmed et al. 1974] [Rao and Yip 1990]. The DCT is similar to the discrete Fourier transform (DFT), which transforms a finite list of sampled discrete-time signals to a discrete number of frequencies. The purpose of performing the DCT is to divide the MCU blocks into segments of varying frequencies.

The DCT can be written either as one- or two-dimensional. Since the DCT is to be run on a block, which is represented as a matrix of a fixed size, the two-dimensional equation (2.1) will be the one that is described.

$$D(i,j) = C(i)C(j)\sum_{x=0}^{N-1}\sum_{x=0}^{N-1} p(x,y)\cos\left[\frac{(2x+1)i\pi)}{2N}\right]\cos\left[\frac{(2x+1)j\pi)}{2N}\right] \tag{2.1}$$

$$C(u) = \begin{Bmatrix} \sqrt{\frac{1}{N}} & if u = 0 \\ \sqrt{\frac{2}{N}} & if u > 0 \end{Bmatrix} \tag{2.2}$$

The function p(x,y), from equation (2.1), is a function which gets the value from the MCU block, i.e. the normalized pixel value from the previous step. N is the size of the block, which will depend on which subsampling is used. For example if no subsampling is used, N should be 8.

The DCT equation in the from (2.1) can also be written as the product of a matrix, the input MCU block, and the inverse of the matrix. The matrix is described in equation (2.3).

$$T_{ij} = \begin{Bmatrix} \frac{1}{\sqrt{N}} & \text{if i} = 0 \\ \sqrt{\frac{2}{N}}\cos\left[\frac{(2x+1)i\pi)}{2N}\right] & \text{if i} > 0 \end{Bmatrix} \tag{2.3}$$

$$D(i,j) = TMT' \tag{2.4}$$

[7] The normal image space, i.e. values in block describe the color of the pixel in the image. In the frequency-domain the values will describe frequencies.

M is the MCU block from the previous step. The block after the DCT will concentrate the low frequencies, which our eyes are good at perceiving, in the upper left corner. The closer we get towards the bottom right, the higher the frequency will be. The value at the top left is called the DC coefficient, while all other values are referred to as the AC coefficients.

### 2.2.6 Quantization

As previously stated, the human eye is more sensitive to the low-frequency information than the high-frequency. The eyes are good at spotting differences in brightness over large areas, but have a more difficult time of spotting the strength of high-frequency variations. By exploiting this fact it is possible to discard most of the high-frequency information. This discarding of information is performed by dividing each coefficient from the block by a constant. The constant varies depending on the index of the coefficient. Usually the quantization matrix has higher values for the higher frequency components, and lower values for the low frequency components.

When the division has been performed each value will be rounded to the closest integer, and this is the second lossy operation in the encoding process. Compared to the first lossy operation, this step is not optional and data will always be discarded. The division will, in many times where the frequency is high, round coefficients to zero. The matrix that is used varies depending on the desired quality. The quality varies from 1 to 100, where 100 gives the best possible quality, and 1 the worst quality. What determines the quality is how large the numbers in the quantization matrix are. The higher the values, the more coefficients in the block will be rounded to zero which will mean a greater loss of information. Note, even if all coefficients are divided by 1 information will still be lost since all coefficients will need to be rounded, from floating values, to integers.

### 2.2.7 Zigzag Scan

With the block possibly containing many zeroes, from the Quantization step, the block can be restructured to more easily allow discarding of the zeroes. The zigzag pattern can be used on the 8x8[8] block to construct a 64 element vector. Because of how the DCT structures the frequencies in a block, performing the zigzag scan will sort the values from low- to high-frequency. Subsequently, because of the Quantization rounding many coefficients to zero, the tail of the vector will in most cases consist of a series of zeroes. Figure 2.5 shows in which order the elements are chosen to create the frequency sorted output array

[8]Depend on subsampling, to make the explanation easier 4:4:4 subsampling, resulting in a 8x8 MCU, is used.

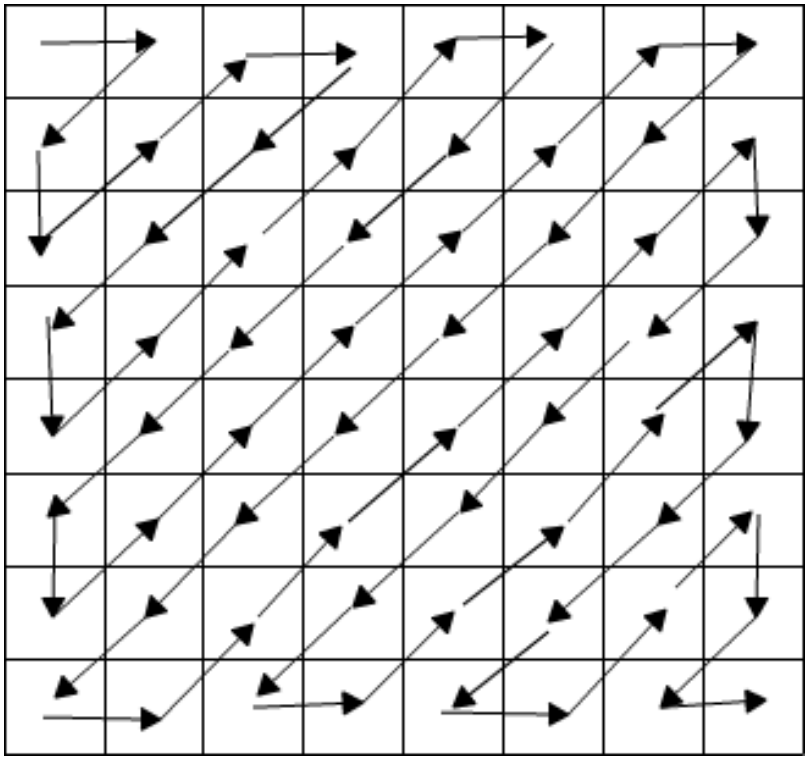


**Figure 2.5:** Zigzag pattern in which the block will be traversed to build the frequency-sorted array.

### 2.2.8 Run Length Encoding

Run length encoding (RLE) is an intermediate step, in which each non-zero AC value will be represented with a pair of symbols:

- Symbol 1 : (RUNLENGHT, SIZE)
- Symbol 2 : (AMPLITUDE)
    - RUNLENGHT - The number of skips.
    - SIZE - The number of required bits to represent the value.
    - AMPLITUDE - The amplitude of the non-zero AC value, i.e. the bit representation of AC.

The number of skips refer to the number of preceding zeroes. If the zigzaged vector contains many series of zeroes, the amount of bytes to represent the vector may be much smaller after the RLE.

Symbol 1 is defined by one byte, i.e. 8 bits. Since symbol 1 is split, each component is represented using four bits. The maximum value RUNLENGHT and SIZE can have is 15, since that is the highest number that can be represented with four bits. (15,0) can be used as a special code for symbol 1 when there are more than 15 preceding zeroes. This code can be repeated a maximum of three times. Another special case is when the remaining AC values are zeroes, then (0,0) is used as symbol 1 to indicate end of block (EOB). For example, the following sequence of values:

- (0)(0)(0)(15)(0)(0)(0)

would look like this after RLE:

- (3,4)(15),(0,0)

The DC coefficient will be encoded differently. Instead of encoding the value of the coefficient, the difference with the previously quantized DC coefficient will be used. If no previous DC value is available, the value will be 0. Two symbols will be used here as well:

- Symbol 1 : DC difference category, i.e. SIZE of the difference value.
- Symbol 2 : DC difference value.

### 2.2.9 Huffman coding

The Huffman encoding step is optional, but could be used to further reduce the compressed size. The output data is reduced by replacing the characters generated from the RLE with a, smaller, variable-length code according to a frequency table. The more frequent a character is, the smaller the code will be. To retrieve the original character a simple look-up in the Huffman table is necessary. The frequency of the characters can either be determined by dynamically checking which characters occur most in the array of characters and form a table from that, or statistically pre-calculated tables can be used. Using the pre-calculated tables are faster, but the dynamic tables will yield higher compression. At least four tables will have to be used, two for luminance DC and AC components and two for chrominance AC and DC components.

This operation does not discard or change any data. However, it should be noted that faults due to transmission of the JPEG image could cause severe degradation in the quality of the image. [Nguyen and Redinbo 2002] explore the fault tolerance of Huffman encoding, and show examples of repeated error in transmission can cause noticeable degradation.

### 2.2.10 JPEG Decompression

Decompression is performed by traversing backwards in the encryption process, as can be seen in Figure 2.6.

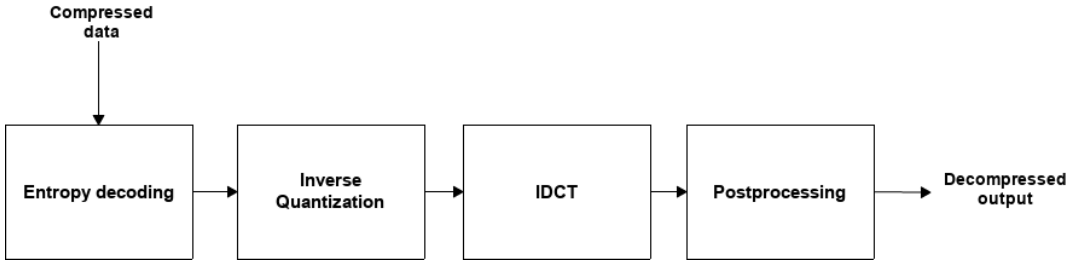



**Figure 2.6:** General decompression algorithm for lossy decompression.

The first step is to decode the entropy data, which is done by sequentially using the Huffman look-up table to re-construct the data. The data will then be in the intermediate RLE form. It is trivial to go back to DCT form since all that needs to be done is read from the start and append the appropriate amount of zeroes. Then the block can be restructured by placing the elements from the array in a matrix following the zigzag order.

Depending on which quality was selected, the appropriate Quantization matrix will now instead be used to multiply with the block. Here it will not be possible to get back all the data that was rounded to zero from the compression. Next, instead of the DCT the inverse DCT (IDCT) is used to transform the blocks from a frequency-domain into the spatial-domain. The algorithm for IDCT is very similar to the algorithm for DCT.

$$ID(i,j) = \sum_{x=0}^{N-1}\sum_{x=0}^{N-1} C(i)C(j)D(x,y)\cos\left[\frac{(2x+1)i\pi)}{2N}\right]\cos\left[\frac{(2x+1)j\pi)}{2N}\right] \tag{2.5}$$

Once the values lie in the spatial domain they have to be offset by adding 128 to each value to return them to a range between 0 to 255. If a value lies outside this region they are clamped, since only 256 shades are used for each component. Before the values can be transformed back to RGB the chrominance components might have to be upsampled. The weighted average of the neighboring pixels will be used to upscale the chrominance. When the values are back in YUV color space the values will have to be transformed back to RGB. The following equations will be used for the transformation:

- R = Cr * (2 - 2 * 0.299) + Y
- G = Cb * (2 - 2 * 0.587) + Y
- B = (Y - 0.114 * B - 0.299 * R) / 0.587

## 2.3 PNG

The PNG, Portable Network Graphics, format is a loss-less format which mainly was introduced to replace the patented GIF, Graphics Interchange Format [CompuServe Incorporated 1990]. Since PNG is loss-less it is especially suitable for image editing; compared to JPEG, saving the file multiple times will not contribute to loss of information since no information is lost in compression. If an image is in PNG format, the image is guaranteed to look the same between applications and devices.

Like JPEG, PNG supports truecolor and grayscale images. On top of that, something often referred to as the palette-based color format is also supported. Here each pixel will be described by an index in a palette, where the palette is a look-up table containing all the colors of the image.

There are two stages to PNG compression. Preprocessing is the first stage, referred to as the filtering stage, and the second stage is the main compression which uses an algorithm called the deflate algorithm.

### 2.3.1 Filtering

Filtering transforms the image information so that the deflate algorithm better can compress the image. There exist multiple ways to perform this transformation, these are:

- None: No changes to the image
- Sub: Each byte in the image will be replaced by the difference between the byte to the left and itself.
- Up: Same as sub filtering, but will use the byte above instead of the byte to the left.
- Average: Each byte in the image will be replaced by the difference between itself and the average of the byte to the left and the byte above.
- Paeth: Same as average filtering, but will include the byte in the upper left in the average.

Filtering is rarely useful when the image is in palette-based form, or when each pixel is described with less than 8 bits [Commander 2014]. It is when the image is in truecolor, or grayscale, with more than 1 byte per pixel that filtering will start being beneficial.

### 2.3.2 Deflate Algorithm

The deflate algorithm was defined by PKWARE in 1991 [Deutsch 1996], and it is a combination of LZ77[] and Huffman encoding [Cormen et al. 2001]. The algorithm is based on the premise

that the information in images, and other types of data, are repetitious to a varying degree. LZ77 achieves compression by replacing parts of the data with matching already passed through data. The structure in which this data is stored is called a sliding window. The idea with the window is to use it as a sort of look-up table for subsequent encoding of data.

The LZ77 algorithm is also applicable to other file types, for example text. To demonstrate how the compression is done, this simple array of text will be used as input:

- 123456789
- ABCBBABCC

LZ77 will encode the input, and output an array in the form of a series of (B,L)C. B stands for the amount of steps back in the window that needs to be taken until a match has been found, and L for how many of the characters should be copied. C is the next character after the match. For example encoding AAAA would first output (0,0)A, since A isn't yet in the window, and add A to the window. The second output would be (1,2)A, since we would have to go back a step in the window to get A, then copy it two times and then end with an A.

The input after encoding will form the following output:

- (0,0)A
- (0,0)B
- (0,0)C
- (1,1)B
- (5,3)C

Huffman encoding will then be used on this output, see the JPEG section for how this can be done.

The encoding process can be time-consuming since for each character the window may have to search through many possible candidates. Although the encoding can be slow, the decoding is simple and fast since there is no need for searching.

## 2.4 WebP

WebP is an image format that has methods for compressing images both lossy and losslessy. It was introduced by Google 2010 [Google Inc. 2014a] to be a modern format, making use of well established compression algorithms and be free from patents. According to studies made by Google, WebP images are about 30% smaller, while still maintaining the same visual quality, than JPEG and PNG.

Lossy compression is similar to compression in JPEG 420, but with some significant changed. WebP makes use of a technique called block prediction [Bankoski et al. 2011]. The image is divided into smaller segments, caller macroblocks, and within each of these blocks the encoder predicts redundant information based on previously processed blocks. This process is referred to as predictive coding and is lossless, which the JPEG counterpart is not.

Next, similar to JPEG, the output from the previous step will be transformed using the DCT, then quantized and entropy-coded. Entropy coding makes use of an arithmetic entropy, which achieves better compression than that in Huffman encoding used in JPEG. Arithmetic encoding was available when the JPEG format was created, but was at the time guarded by copyright claims which have all but expired.

Another significant difference compared to JPEG is quantization step. WebP makes use of something called adaptive block quantization. What this does is to create segments of visually similar looking areas in the image. For each of these segments the parameters for the quantization can be adjusted to give the best possible quality.

A major advantage with WebP over JPEG, except from better compression, is that Lossy WebP supports transparency. Adding transparency adds approximately 22% [Alakuijala 2012] to the total size compared to lossy WebP without transparency. Compared to PNG, WebP + Transparency give approximately 60% size savings.

Lossless compression make use of multiple different techniques to achieve the best possible compression. This thesis will not delve into the details of these techniques.

# 3

# Mobile GPUs

On a mobile device the GPU is usually integrated into the SoC, socket-on-a-chip, together with the CPU and other components. The GPU memory is usually shared with other SoC components, which will result in a lower memory bandwidth [Akenine-Moller and Strom 2008]. Some of the major manufacturers of SoCs for mobile are [Qualcomm Inc. 2014] with their SnapDragon SoC series, [Imagination Technlogies Limited 2014] with their PowerVR SoC series, and the Tegra SoC series by [NVIDIA Corporation 2014]

## 3.1 GPU Architecture

The GPU are originally intended to be used in reference to graphics, hence the name Graphics Processing Unit (GPU), which includes rendering of triangles, lines and points. To better understand the process of using the GPU the rendering pipeline of a triangle will be described, excluding the more complex parts and focusing on the core components.

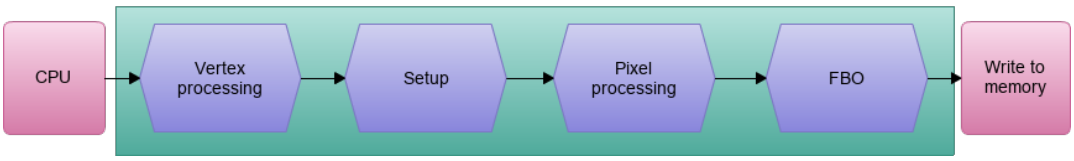



**Figure 3.1:** Overview of the rendering pipeline of a GPU.

First the CPU will provide the GPU with the triangle information that it wants to be rendered. A triangle is described using vertices, these vertices will be passed to the "vertex processing unit"

which will transform the triangle into the correct position. This stage is programmable, which means it is possible to tell the GPU how the transformation of the vertices should be done.

The vertices then enter a stage called "setup", which will create the actual triangles from three vertices and compute the constant data over each triangle. Next the "per-pixel processing" will perform per-pixel operations for visible fragments. This stage will determine the output color for each of the pixels. This stage is also programmable, i.e it is possible to tell the GPU what color each pixel should have.

The last stage will perform various "frame buffer operations", which for example includes depth testing and stencil test. Depth-test will make sure that the triangles closest to the screen will be shown, and the stencil test can be used to limit which pixels should be rendered.

### 3.1.1 OpenGL ES

To program the vertex- and pixel processing units, one possible language to use is GLSL [Khronos Group Inc. 2009]. In OpenGL the program running on the vertices is called vertex shader, and the program running per-pixel is called the fragment shader. The vertex shader will run on each vertice, and the fragment shader will run on each pixel.

OpenGL ES 2.0 is the currently most supported version of the OpenGL ES interface (see Figure 1.4) . OpenGL ES 3.0, and 3.1, is also available, but as of now only covers a little over 10% of total Android devices on the market. But OpenGL ES 3.0 is backwards capable with OpenGL ES 2.0, so with OpenGL ES 2.0 it is possible to target almost all Android devices. OpenGL ES is a subset of the OpenGL standard used on desktop graphics cards. There are differences in OpenGL ES compared to the desktop version of OpenGL, for example to reduce power usage precision qualifiers were introduced in OpenGL ES [Munshi et al. 2008].

When programming the shaders, the fragment shader will often take a texture as input to define the output color. A texture is a block of pixels, where each pixel is a set amount of bytes. The pixel format is defined before the texture is uploaded to the GPU. When the texture size does not match the output dimension, there are several parameters that can be used to set the behavior of the fragment shader. One possible option is to linearly interpolate the pixel values in the texture image before drawing. Another option is to perform nearest-neighbor filtering, meaning copying of the closest pixel. Linear interpolation will produce a more blurry effect, while nearest-neighbor filtering will be more blocky.

## 3.2 GPGPU

The structure of the GPU also proved useful for other areas. When non-graphics computations are performed on the GPU it is called GPGPU, which stands for General Purpose computations on

GPUs. The GPU of today is a powerful streaming processor with a programmable pipeline, this makes the GPU a good candidate for parallel computations [Goodnight et al.2005].

There are several ways to perform GPGPU on mobile devices, such as OpenGL [Khronos Group Inc. 2009], OpenCL [Khronos OpenCL Working Group 2014], or RenderScript [Google Inc. 2014c]. While OpenCL and RenderScript both are designed with GPGPU in mind, OpenGL was created with the intent to render graphics. This is not to say that GPGPU isn't achievable with OpenGL, just that it may be more difficult for certain kinds of computations. For example, when in OpenGL ES 2.0 the computations have to be mapped to the graphical structure of OpenGL (textures, primitives). In OpenCL it is possible to feed the GPU with arrays or blocks of data, and there is only a single stage.

CUDA is similar to OpenCL, as it is designed with GPGPU in mind, but it is only available on NVIDIA desktop devices [NVIDIA Corporation 2013a]. CUDA makes use of kernels. A kernel is executed on the GPU as an array of threads in parallel. Threads are grouped into blocks, where the blocks are grouped into grids. The kernel will run as a grid, of blocks, of threads. Threads within the blocks can share results with one another, and synchronize with one another. The blocks can either be run in parallel, or sequentially. A paper using CUDA will later be explored in section 5.2.

An advantage with OpenGL, over OpenCL and RenderScript, is that most devices support OpenGL. Roughly 99% support OpenGL ES 2.0, or higher, for all Android devices. OpenCL is only available on a select number of devices, and RenderScript is still not widely adopted and is only supported on Android 4.x devices. Compute shaders were introduced in OpenGL 3.1. Instead of using the pipeline designed for graphics, the compute shader is a single-stage machine. This is similar to that of OpenCL and CUDA.

# 4

# Chromium

Chromium is an open source project by Google [Google Inc. 2014b]. Chromium consists of several components, with Blink and Skia being the interesting ones in regard to image compression/decompression and rendering of images. Both components make use of third party libraries, such as libraries for decoding JPEG and PNG images.

## 4.1 Blink

Blink is the browser engine of Chromium. A browser engine, among other things, takes HTML content and formatting content, such as CSS, and processes this information into something that can be displayed on the screen. Blink is a fork of the WebCore component from the popular browser engine WebKit [Apple Inc 2014], which for example is used by Apple for their web browser Safari.

Before getting to the actual decoding of images, a brief description of how Blink is able to go from a HTML file to something that can be shown on a display will be described. When Blink receives a web page not all resources, needed to display the page properly, will be available (like images). When a HTML page is recieved, two pipelines will be created; one for loading the resources (images), and one for loading the page. The pipeline for loading the page essentially contains three stages:

- Policy stage: Determine whether the page is allowed to be loaded, or dismissed. If the page is opened in a new tab, or the domain is changed, a new process will be created to ensure that the new page is secure in a sandboxed environment.
- Provisional phase: Determine whether the page is a download or a commit[9]. If it is a commit, the current content will be replaced by the new page.

- Committed phase: Where the content will be parsed and prepared to be displayed.

The interesting pipeline, in regards to image decoding, is rather the pipeline for loading the resources. It will start by loading the URL of the resource to determining if the resource already is in the cache. If it isn't, the resource will be downloaded and cached. If the cached resource is an image, the header information of the image will be extracted by partially decoding the image. The actual decoding of the image will not take place until it is time to show it, therefore Skia needs to be introduced since Skia handles the drawing of the image.

## 4.2 Skia and Decoding

Skia, a 2D graphics library, is used for almost all graphics related operations for Chromium. These include, among others, rendering of images and text. Skia will take the information generated by Blink and perform the appropriate actions to display the content on the screen. The Skia library contains multiple different back-ends, one of which makes use of OpenGL. With the OpenGL back-end it is possible to run custom shaders on the textures we create with Skia. The currently default back-end will instead rasterize the entire page on the CPU and send the page as a texture to the GPU. With Ganesh it is possible to run different shaders for different parts on the page. In other words, it is possible to run a custom vertex and fragment shader for the JPEG images.

When Skia renders an image, Skia will check its cache to see if the image already has been decoded. If not, the decoding process will start. If the image is in cache, Skia will retrieve it from there instead of decoding the image again. When the decoding process of an image is started, the image will still be in the process of being downloaded. Therefore the image will be partially decoded as it is being downloaded. Blink used third-party libraries for handling the decoding, but this partial way of decoding is not the standard procedure of decoding. This means that Blink will have to setup the decoders in a way so that partial decoding is possible.

The actual process from downloading an image to displaying it on the screen is far more involved than what is being described here, but most of it is unrelated to image decoding and will therefore be left out of the thesis.

[9] A commit is when the page should be shown, i.e. when the page need to go to the committed phase and prepare to be displayed.

# 5

# Parallel decoding

When deciding which format to use, the JPEG format stood out among the others. It has been available since the early 90s, and is currently the most widely used format on the web, there have been several attempts to accelerate the process of compression and decompression of JPEGs. Therefore only research papers for the various JPEG phases will be explored for accelerating on the GPU. From the initial investigation of the JPEG format it can be seen that there are multiple phases could benefit of being performed on the GPU. Papers will be presented starting from the first step in the decoding process, Huffman decoding, ending with papers involving the IDCT. No research could be found for upsampling and color transformation, both of which will be the building blocks for the analysis in the next two sections.

## 5.1 Parallel Huffman Decoding

Huffman decoding is designed as a sequential process, i.e. it is complicated to divide the array of data into blocks until the whole Huffman encoding has been done. The reason behind this is because of the way the data is structured after compressing. The compressed image will be an array of codes, referred to as variable-length codes. These codes need to be computed in sequence, since there is no way to know where a block in the array starts and ends until the whole array has been decoded.

However, Klein and Wiseman [2003] made a keen observation. If you start decoding at a random location in the variable-length coding sequence, it is possible, after decoding for a while, to get correctly decoded output. Explained more practically, a *process i* would start decoding *block i*, then overflow into the next block until a synchronization point is reached. A synchronization point is a point where the decoded output of *process (i+1)* matches the output of the overflowing *process i*. Once this synchronization point has been found, *processor i* can stop and start on a new block.

Finding the synchronization point in JPEG may in some cases not happen, resulting in a process having to decode multiple blocks or more in sequence. A completely sequential process would be the worst case scenario. With the addition of multiple processes running at the same time, this may introduce unnecessary excess making the parallel implementation worse than the sequential.

A better way to allow parallelism would be to introduce code-word alignment at block boundaries. This would make parallelism trivial since it would be easy to separate the compressed data into their underlying blocks. However, this would require changes to the specification, which would not help for the images already compressed.

## 5.2 Parallel IDCT

IDCT, and DCT, are particularly suitable for the GPU since any MCU block can be computed independently of other MCU blocks. But even though the GPU is suitable, with languages like OpenGL where you have to conform to the graphics pipeline the overhead introduced may sometimes overshadow the parallelism. Two papers will be taken into consideration. One which conforms to the graphics pipeline, and another which fully utilize the properties of GPGPU with the help of CUDA.

Fang et al. [2005] proposed five different methods in which IDCT could be performed on the GPU. None of the methods outperformed the SSE2[10] CPU implementation, but three of them outperformed the MXX[10] CPU implementation. A SSE implementation to accelerate DCT can be seen in [Yam 2005]. libjpeg-turbo have the option to use SSE2 in their implementation to perform the DCT and IDCT.

Fang et al. [2005] made use of the graphics pipeline, i.e. mapped the IDCT computations to textures and vertices. Textures were used to input the IDCT matrix, and the output was rendered to a texture which could be later accessed on the CPU. The method that gave the best result is called Multi-channel and Multiple Render Target Multiplication (MCMRTM). The 2D-IDCT can be performed via splitting the matrix multiplication into two parts, here they call them LeftX and RightX. This is the same as equation (2.4) in the DCT section. To reduce the amount of times the texture needs to be read, four render targets are used in computing LeftX, and two for RightX.

Obukhov and Kharlamov [2008] proposed an implementation of the DCT in their CUDA language. Since this transformation is the DCT, not IDCT, the equation that will be used is equation (2.4). The implementation in the paper only operates on 8x8 MCUs. A set of blocks are referred to as a macroblock, which commonly have 4 or 16 blocks. The matrix multiplication shown in equation (2.4) is almost never used in CPU implementations, since more suitable methods for the CPU are better. But in CUDA the equation map nicely.

[10] SSE2 and MXX are single instruction, multiple data instruction set, meaning computations are performed in parallel on a set of data.

Obukhov and Kharlamov [2008] suggests two methods. The first method map directly to the equation (2.4), which will run 64 threads for one MCU block. Every thread will compute either a DC or AC coefficient, and to reduce computation each waveform, the cos functions in equation (2.1), will be pre-computed. The second method will split the image into a number of macroblocks. Instead of using one thread for each coefficient, 8 threads will be used for each MCU block. The macroblock must be a multiple of the number of threads in a warp. A warp is a set of threads that are executed physically in parallel. The warp size could for example be 32, then the macro block have to be 4 MCU blocks. The reason for only using 8 threads instead of 64, is to reduce the amount of redundancy for the matrix multiplications.

# 6

# Standalone Implementation

The standard way to decode an image in Android is to use the built in decoders. There is a facilitatory class which takes an input, either an array of data or a stream of data, and performs the whole decoding process. This class can take any image format that Android supports and perform the appropriate decoding and return the decoded image.

To actually be able to be more explicit in the decoding process you have to import a library that can decode the image. Using this library you are able to call functions directly and therefore be more specific how the decoding process is performed. Since JPEG is the format being decoded, a library called libjpeg-turbo [Commander 2014] will be used. libjpeg-turbo is derived from the library libjpeg [Joint Photographic Experts Group 2014], which is a library that has been available since 1991. libjpeg-turbo extends libjpeg with parallelism, which means that it can make better use of the multiprocessing structure of modern CPUs.

The standard way to display an image in Android is by simply creating a container in which the image should be shown in, and feeding this container with the output from the decoding process. It is not possible to run code on the GPU this way, instead you have to use OpenGL to be able write custom shaders for the GPU.

With the use of libjpeg-turbo it is possible to specify how much of the image that should be decoded on the CPU. As shown in the previous section, some of the steps in decoding JPEG can be done in parallel. Without altering the encoding of an image, Huffman encoding is hard to do in parallel since the complications introduced when jumping forward, i.e the decoding has to be sequential. Performing of the inverse Quantization and the IDCT on the GPU are both possible, which can be seen in Fang et al. [2005] where Fang managed to map the DCT to the graphics pipeline. Fang never got any speedups, compared to the parallel CPU implementation, but that was nine years ago and today it may be the complete opposite. However, considering the implementation should also be integrated with Chromium the methods by Fang et al. [2005] will introduce complications. See

the discussion section (9) for why this is. However, one part of the process that would be suitable to be performed on the GPU is the upsampling and color space transformation.

By not performing the upsampling on the CPU it is possible, theoretically, to save up to 50 % of the total amount of memory that is required to be stored on the CPU and GPU (texture memory). This number varies depending on the subsampling used when the image was encoded. See Figures 6.1 for how the dimensions of the components after subsampling. The way the components are placed in the image will later be very important in the next section (7).

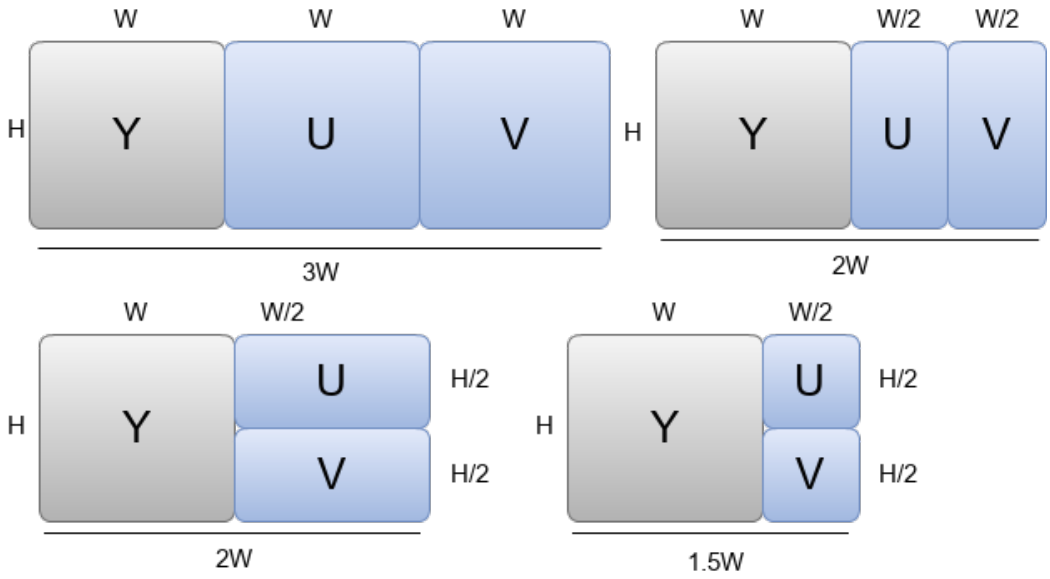



**Figure 6.1: Top Left** 4:4:4: Will result in no memory savings, since no information was discarded and no upsampling will be performed. **Top Right** 4:2:2: Will result in ~33% memory savings. **Bottom Left** 4:4:0: Will result in ~33% memory savings. **Bottom Right** 4:2:0 Will result in 50% memory savings.

It is possible to end the CPU decoding, in libjpeg-turbo, after the IDCT has been performed and the values have been normalized, but before the upsampling and transformation have been performed. The output of the decoding process will be an array firstly containing the luminance, followed by the blue chrominance, with lastly the red chrominance (see Figure 6.2).

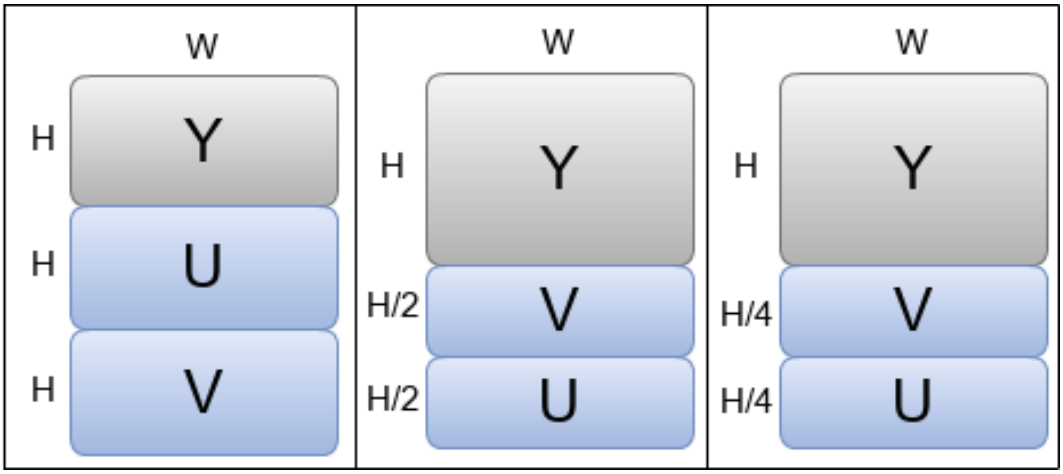



**Figure 6.2: Left** 4:4:4 **Middle** 4:2:2 or 4:4:0 **Right** 4:2:0

It may not be apparent at first, but after looking at 4:2:2 and 4:2:0 in Figure 6.2 one may be confused. For example, in 4:2:2 the height seem to be halved while the width remain the same as that of the luminance. But remembering back to the section about subsampling, 4:2:2 should discard in the horizontal plane not the vertical. The explanation for this is interleaving rows. Since the rows in 4:2:2 are half the size in the chrominance components, two of the chrominance rows occur on one texture row. The same idea holds true for 4:2:0, but with the addition that the height is also halved.

## 6.1 Multiple textures

There are various ways of performing this upscaling process on the GPU. One way is to feed the GPU with multiple textures where each texture representing one of the color components (YUV). So, one texture represents the luminance component, one the blue chrominance component, and one the red chrominance component. To achieve this, the data from libjpeg-turbo will have to be extracted into separate textures. Each of these textures will be sent to the GPU using OpenGL. Vertex and fragment shaders will be used to perform the appropriate upscaling and transformation.

When defining the textures to be sent to the GPU, the linear interpolation option will be selected for each texture. This option will change the way that the texture is read when the resolution output doesn't match the image dimensions, which is the case for subsampling. This option will take the weighted average of the surrounding four pixels, given a coordinate, to get pixel value. This linear interpolation will perform the upsampling for us. The vertex shader in listing 6.1 will run for all vertices.

**Listing 6.1:** Simple vertex shader

```
attribute vec4 a_position;
attribute vec2 a_texCoord;
varying vec2 v_texCoord;

void main()
{
  gl_Position = a_position;
  v_texCoord = a_texCoord;
}
```

The vertex shader receives the vertice position and texture coordinate sent from the CPU. The only thing the vertex shader does is set the position of the vertices, and pass along the texture coordinate to the fragment shader in listing 6.2.

**Listing 6.2:** Upsample and color transformation fragment shader

```
varying vec2 texCoord;
uniform sampler2D yTexture;
uniform sampler2D uTexture;
uniform sampler2D vTexture;
mat4 yuvMatrix = mat4(1.0,    0.0,  1.402, -0.701,
                      1.0, -0.344, -0.714,  0.529,
                      1.0,  1.772,    0.0, -0.886,
                      0,        0,      0,      0);
void main() {
  float yChannel = texture2D(yTexture, texCoord).r;
  float uChannel = texture2D(uTexture, texCoord).r;
  float vChannel = texture2D(vTexture, texCoord).r;
  vec4 channels = vec4(yChannel, uChannel, vChannel, 1.0);
  gl_FragColor = vec4((channels * yuvMatrix).xyz, 1.0);
}
```

The fragment shader will receive the texture coordinate from the vertex shader, and the three textures discussed earlier. Now, a texture look-up on each texture will be done with the supplied texture coordinate. This will return the correct, upsampled, component value for the pixel. The reason the correct subsampled value is returned is because of the linear interpolation. Before the fragment shader run the texture will, depending on the texture coordinate, linearly interpolate the value at the texture coordinate for each texture.

The transformation step from upscaled YUV to RGB is simply a matrix multiplication. The output color is set to the transformation, and alpha is set to a constant 1 since JPEGs don't support transparency.

# 7

# Chromium Implementation

With a working standalone implementation, of part of the decoding process on the GPU, it is possible to implement the upscale and color space transformation in the Chromium code-base. As mentioned in section 4.2, the decoding of images is handled a little differently than how images normally are decoded. The decoding process is started before the entire image has been downloaded. This is partly possible because of the way the block structure of the JPEG image is encoded.

First, support for converting an image partially to downscaled YUV needs to be done. The way the partial decoding is done is not particularly difficult, it is done with the help of a marker keeping track on which row in the image we currently are on. The rows of the image that is already decoded will not be decoded again, only the newly downloaded data will be decoded.

A restriction, that was not in the standalone implementation, is that the image needs to be decoded into a single bitmap. In Chromium a bitmap is used to contain an decoded image, among other things, on the CPU side. The way an image is represented, stored, and moved through the code only support for one bitmap per image. Extending a bitmap to contain more images, or sending multiple bitmaps through the code, would not be possible without major reworkings of the code.

If we look the figures at Figure 6.2 again, we note that the way the data was stored was by first adding the luminance component followed by the chrominance components. But the way the chrominance components were layed out was interspersed. This will pose a problem for Chromium later on in Skia. Skia will in some cases create something called tiles if the image is larger than some specified dimension. What Skia does is split the image into multiple tiles and render the tiles separately. What this means is that an image will essentially be split into smaller images, where each image will have to be passed to the GPU separately.

It is not possible to know how the tiles will be split, therefore it is necessary to store the data in a way that the structure accurately represents the way the image will be displayed on the screen.Storing

the data this way makes retrieving the components, even with tiling, easy. The way the data is stored is the same way the data was presented in Figure 6.1. In other words, the luminance is always stored as a block of w*h followed by the chrominance block to the left. Depending on how the image is subsampled the chrominance blocks will be represented differently, look at Figure 7.1 for an explanation of how a 4:2:2 image is stored in a bitmap.

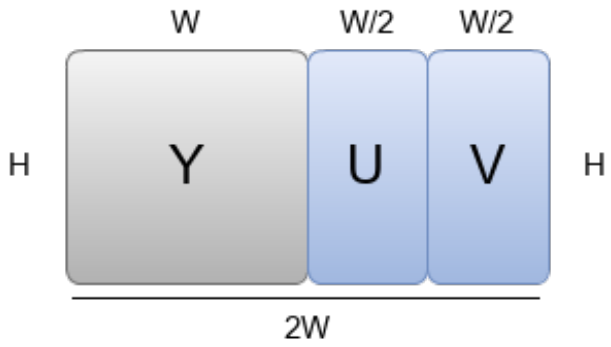



**Figure 7.1:** 4:2:2: The luminance component is stored to the left as a w*h block, and the chrominance parts are stored as (w/2)*h blocks followed to the left of the luminance. When the components are extracted from the bitmap, a simple operation that extracts a rectangle can be used to get each component (or tile of each component).

The bitmap that is used to represent the image introduces yet another restriction. The bitmap dimension must accurately represent the actual dimension of the image. The reason for this is because of the way Chromium handles rendering. If the dimension of the bitmap do not match with the dimension of the image, the image will not be rendered correctly. This is a problem since the dimension of the bitmap will not match the dimension of the image, see for example Figure 7.1 where the width of the bitmap is double that of the actual image. This problem can be circumvented by exploiting the definition of a pixel in a bitmap. A pixel in Chromium is defined by a set amount of bytes, for example RGB would require 3 bytes per pixel (bpp). The bpp varies depending on how the image is downsampled:

- 4:4:4 - 3 bpp.
- 4:4:0 - 2 bpp.
- 4:2:2 - 2 bpp.
- 4:2:0 - 1.5 bpp.

Exploiting the bpp it is possible to store all components in all subsampling formats in one bitmap.

Before the bitmap is sent as a texture to the GPU the components need to be extracted from the bitmap, since the shader expects one texture to represent one component. The tile splitting,

and extracting of components will be combined into one function which will return three bitmaps representing the components. The function which split the image take five inputs:

- **tx** and **ty**: The start position of the tile. If only one tile for the image is used, this value will always be (0,0).
- **tw** and **th**: The width and height of the tile. If only one tile for the image is used, the values will be the same as the actual width and height of the image.
- **iw** and **ih**: The actual width and height of the image.

To extract the components from the bitmap, a region will be defined for each component and for each subsampling. The region represent where, and how much, should be extracted to get the appropriate component. The regions, for each subsampling, are detailed in tables 7.1, 7.2, 7.3, and 7.4. The position, x and y, and dimension, w and h, in the tables are in the image dimension space. What I mean by this is that

**Table 7.1:** 4:4:4 split. Since the chrominance components are the same dimension as the luminance, each region is the same size but the start position is offset using the width of the image.

| **Components** | **x** | **y** | **w** | **h** |
|---|---|---|---|---|
| Luminance (Y) | tx / 3 | ty | tw / 3 | th |
| Chrominance (U) | tx / 3 + iw / 3 | ty | tw / 3 | th |
| Chrominance (V) | tx / 3 + (2 * iw) / 3 | ty | tw / 3 | th |

**Table 7.2:** 4:4:0 split.

| **Components** | **x** | **y** | **w** | **h** |
|---|---|---|---|---|
| Luminance (Y) | tx / 2 | ty | tw / 2 | th |
| Chrominance (U) | tx / 2 + iw / 2 | ty / 2 | tw / 2 | th / 2 |
| Chrominance (V) | tx / 2 + iw / 2 | ty / 2 + ih / 2 | tw / 2 | th / 2 |

**Table 7.3:** 4:2:2 split.

| **Components** | **x** | **y** | **w** | **h** |
|---|---|---|---|---|
| Luminance (Y) | tx / 3 + (2 * iw) / 3 | ty | tw / 3 | th |
| Chrominance (U) | tx / 4 + iw / 2 | ty | tw / 4 | th |
| Chrominance (V) | tx / 4 + (3 * iw) / 4 | ty | tw / 4 | th |

**Table 7.4:** 4:2:0 split.

| **Components** | **x** | **y** | **w** | **h** |
|---|---|---|---|---|
| Luminance (Y) | (2 * tx) / 3 | ty | (2 * tw) / 3 | th |
| Chrominance (U) | tx / 3 + (2 * iw) / 3 | ty / 2 | tw / 3 | th / 2 |
| Chrominance (V) | tx / 3 + (2 * iw) / 3 | ty / 2 + rh / 2 | tw / 3 | th / 2 |

If the original bitmap was not structured, as shown in Figure 6.1, this split would be more complicated. This is because of the way bitmaps in Chromium are designed, for example splitting out a region from a bitmap is already possible because of tiling. If the bitmap were structured like 6.2, i.e. the components are appended to the bitmap one after the other, it would be harder to extract the regions (partly because of the interleaving rows). It is certainly possible, but making big changes to Chromium could introduce unexpected behavior.

When the three components have been extracted from the bitmap they are sent to the GPU as textures. To apply shaders to the textures, something called effects in Chromium must be used. Effects are wrappers around shaders. The vertex shader will not have to be defined, since the position of the image on the page is not defined by the image. Rather, it is the fragment shader that will be needed. The fragment shader that is used is the same as the first version in the standalone implementation.

# 8

# Benchmarking and Results

The benchmarking will be performed on the NVIDIA Shield device. The Shield uses the Tegra 4 SoC [NVIDIA Corporation 2013b], which has a quad-core CPU, 2 Gb of memory, and a 72-core GPU. Out of the 72 cores, 24 of them are dedicated for the vertex shader and 48 cores to the pixel shader.

The upsampling process and color transformation process are not dependent on the information in an image. What determines how long upsampling and color transformation takes is the subsampling and the amount of pixels. First, a measurement will be done for how long it takes for an image to be fully decoded compared to partially decoded in libjpeg (see table 8.1). The measurement is started prior to telling libjpeg-turbo to fully decode, or partially decode, an image and ended when the decoded image is returned. The GPU computations are not included in this table (instead, see table 8.2).

**Table 8.1:** libjpeg-turbo decoding a 1280x720 image. The values, for each subsampling and decoding, are an average of ten runs.

| **Decode process CPU** | **CPU cycles** | **Diff** |
|---|---|---|
| 4:4:4 - Full decode | 47404033 | |
| 4:4:4 - Stop before upsample | 40513375 | 6890658 (~15% faster) |
| 4:2:0 - Full decode | 32588883 | |
| 4:2:0 - Stop before upsample | 22961441 | 9627442 (~30% faster) |

From the table we can gather that upsampling and color transformation, in this case, takes between

17% to 33% of the total decoding process. The GPU will be benchmarked by telling the CPU to wait until the GPU have finished rendering and measuring the amount of CPU cycles that have passed while waiting for the GPU to finish (see table 8.2). For upsampling and color transformation the fragment shader presented in listing 6.2 will be used. For the fully decoded image the simple fragment shader in listing 8.1 will be used.

**Listing 8.1:** RGB fragment shader. The output color is set to the RGB components gathered from the texture look-up.

```
varying vec2 texCoord;
uniform sampler2D texture;
void main() {
  gl_FragColor = vec4(texture2D(texture, texCoord).rgb, 1.0);
}
```

**Table 8.2:** Display a 1280x720 image using OpenGL.

| **Decode process GPU** | **CPU cycles** | **Diff** |
|---|---|---|
| 4:2:0 - RGB fragment shader | 7714609 | |
| 4:2:0 - Upsample fragment shader | 11441875 | (~50% slower) |

From table 8.2 we can gather that by performing upsampling and color transformation on the GPU is roughly 50 % slower. The combined CPU decoding and GPU decoding can be seen in table **??**.

| **Decode process CPU + GPU** | **CPU cycles** | **Diff** |
|---|---|---|
| 4:2:0 - CPU | 40303492 | |
| 4:2:0 - CPU + GPU | 34403316 | (~17% faster) |

In total, partially decoding the image on the CPU and performing upsampling and color transformation on the GPU is faster.

In Chromium, a minimum of 25% memory on the CPU and GPU will be saved by the proposed implementation in section 7. The reason for this is that instead of using four bytes per pixel, RGB and alpha, we instead use three bytes per pixel, YUV. Additional memory savings will depend on

the subsampling format of the image. The total amount of memory that can be saved in theory, on the CPU and GPU, can be seen in table 8.3

**Table 8.3:** Memory savings for different subsampling

| **Subsampling** | **Memory saved (%)** |
|---|---|
| 4:4:4 | 25% |
| 4:4:0 | 50% |
| 4:2:2 | 50% |
| 4:2:0 | 62.5% |

Benchmarking the memory in the standalone implementation is not very interesting, since the amount of memory saved on the CPU and GPU are exactly the same as the figures above. In Chromium, however, there is a different story. Bitmaps are used to store the image in Chromium and the way bitmaps are handled, for example caching, introduce overhead for images. So changing how much data is stored for an image could potentially give even higher memory savings.

The test case, that will be used in Chromium, will consist of loading a page with a single image. The reason for this is to avoid unnecessary complications with the benchmarking results. The image that is used for testing is a 3000x3000 image with subsampling 4:2:0. The memory measured for Chromium is the total amount of memory for the page containing the image.

**Table 8.4:** Memory on CPU in theory compared to Chromium for a 3000x3000 4:2:0 image. The Chromium values are an average over multiple runs.

| **Decoding** | **Memory CPU** | **Diff** |
|---|---|---|
| Fully decoded in CPU in theory | 36 Mb | |
| Partially decoded in GPU in theory | 13.5 Mb | 22.5 Mb (62.5 % smaller) |
| Fully decoded in CPU in Chromium | 162 Mb | |
| Partially decoded in GPU in Chromium | 70 Mb | 92 Mb (~55 % smaller) |

What can be gathered from this table is that there is a definite reduction in memory on the CPU. In theory 22.5 Mb should be saved for the image, but reality 4x as much memory is saved. However, if the image instead is 500x500 with 4:2:0 subsampling the overhead is not noticeable (see table

8.5).

**Table 8.5:** Memory on CPU in theory compared to Chromium for a 500x500 4:2:0 image. The Chromium values are an average over multiple runs.

| **Decoding** | **Memory CPU** | **Diff** |
|---|---|---|
| Fully decoded in CPU in theory | 1 Mb | |
| Partially decoded in GPU in theory | 0.375 Mb | 0.625 Mb (62.5 % smaller) |
| Fully decoded in CPU in Chromium | 39 Mb | |
| Partially decoded in GPU in Chromium | 38 Mb | 1 Mb (~2 % smaller) |

When it comes to measuring memory on the GPU it becomes more complicated. What Chromium does is set aside a set amount of GPU memory, which means there is no way to determine how much of the allocated memory is actually being used. However, on the GPU there should be less overhead required for the image. The image is sent as a texture to the GPU once. It is only sent to the GPU again if the GPU purged the texture. Therefore the memory saved on the GPU should better align to the memory savings that can be seen in table 8.3.

On the Tegra device it is possible to measure the power consumption. The measurement is from loading a single image in Chromium with the resolution 3000x3000. This image was loaded multiple times, and figure 8.1 show a comparison between decoding fully on the CPU and partial decoding on the GPU.

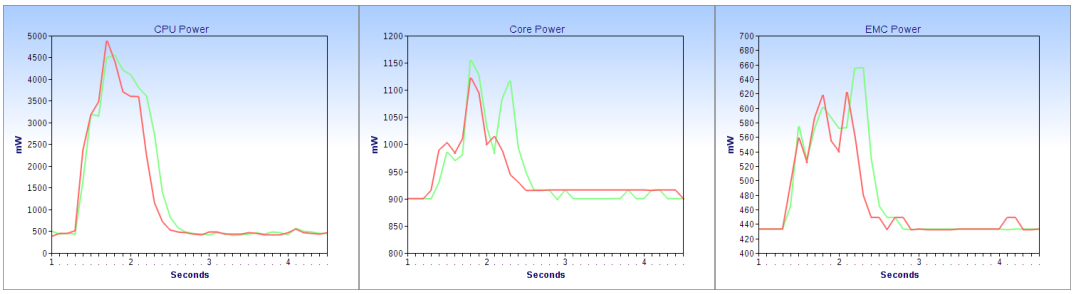



**Figure 8.1:** The red line is with upsampling and color transformation on the GPU, green is CPU only. **Left** The CPU load. **Middle** The GPU load **Right** EMC (Embedded Memory Controller).

It should be mentioned that the documentation for these values is non-existent, therefore the values should be with a grain of salt. For example, I'm unsure if Core and EMC power stand for what I think it does. From the image it can be seen that the changes in power consumption,

between the two implementations, are not extremely noticeable. Since less work is done on the CPU, for the GPU implementation, the CPU power should be lower, which it is not. And the GPU power consumption appear to be lower for the GPU implementation, even though the GPU implementation have a higher computational load. Looking at the EMC power, which I think is the memory controller, the power consumption should be lower since less memory is used, which it is slightly is. To better interpret the power consumption better documentation and tools are needed to make any significant observations.

# 9

# Discussion and Future Work

As can be seen in the benchmarking results in the previous section, there is the possibility for huge savings in memory by not upsampling on the CPU. It should however be noted that the huge savings that can be seen in table 8.4 only are possible if the page contains large enough images. Looking at table 8.5, where the image is 500x500, it can be seen that the amount of Mb that can be saved is almost negligible, which is a stark contrast from the previous table. Even if it is possible to see saving of 97 Mb, the norm[11] will probably be around just a couple of Mb. Nonetheless, as mobiles are getting higher resolution displays and internet bandwidth speed is increasing the quality of the web will increase. To increase quality, higher quality images will be needed, and then upsampling on the GPU will start showing more noticeable results.

The power consumption benchmarking, in section 8, is in all likelihood not statistically significant. For the values in figure 8.1 to mean anything, many more tests would be needed. I could see this new method both having lower power consumption, since less work is done on the CPU, but also higher power consumption because of the increased load on the GPU. Power consumption is a very interesting field, especially for mobile devices, and better measurements would be very interesting.

I mentioned in section 6 that DCT/IDCT is not suitable for Chromium. That statement is currently true, but could certainly in a few months be false. The implementation by Fang et al. [2005] required the data to be rendered to a texture and four render-targets. The way Chromium handles communicating with the GPU introduces some restrictions for what can, and cannot, be done. It certainly would be possible to rework the code to allow this, but I focused on the parts of the JPEG decoding process which would suit the design of Chromium. And the CUDA implementation, in section 5.2, is currently not possible. But when the OpenGL ES 3.1 user-base grows it would be possible to implement it with compute shaders.

[11] See for example the mobile site for bbc, m.bbc.co.uk/news. That site contains a lot of low-resolution images, resulting in just a couple (if even that) Mb of memory savings.

The upsampling that is done using linear interpolation may not produce the best possible upscaling. When we for example have downsampled with 4:2:2, each chrominance texture has half the width of the luminance texture. Now, to get an upscaled pixel between pixels, linear interpolation will use the four closest surrounding pixels to extrapolate the pixel value. Since this is not the same way that libjpeg-turbo implements the upsampling, the resulting image between the two versions may differ slightly. Since changes in the chrominance are hard to spot, I myself have not noticed any changes to images upsampled with linear interpolation. But if one were to compare the upscaled textures with a program there most likely would be slight variations. It would also be possible, instead of linear interpolation, to use the nearest pixel value to perform the upscale. This, however, introduces a block effect, because of color repetition, and is therefore not suitable for upsampling. Instead of using filtering it would also be possible to interpolate the value directly in the fragment shader. This would most likely be even more expensive than linear interpolation, but if exact reproduction the CPU implementation is wanted that is the way to go.

In the standalone implementation I also tried out only using a single texture for all components, instead of one for each component. I wanted to explore if sending only one texture, compared to three, would be beneficial in regards to texture upload (overhead with multiple textures instead). This implementation would require more work on the fragment shader, since for each fragments we had to find each components in the texture. The operation, required for finding the chrominance, introduced problems on devices with low float precision. On some devices, for example the Tegra 4, the float precision was too low which resulted in the wrong texture look-up for the chrominance components. In the end this implementation was never tried in Chromium.

I also tried using RenderScript to perform upsampling and color space transformation on the GPU. Since RenderScript is designed for GPGPU operation, this implementation was the easiest to setup since Android handled all communication with the GPU. But there were a couple of downsides to this implementation. Firstly, for upsampling and color space transformation to be beneficial on the GPU they need to be performed at the same time as the image is being rendered. In RenderScript, after the upsampling and color space transformation had been done the data was sent back to the CPU. So to render the image, the data first would have to be sent to the GPU, then back to the CPU and then to be handled by Android to render the image. Secondly, RenderScript is only supported for android version 4.0 and there is little to no proper documentation. Because of these facts, I gave up on the idea of using RenderScript.

With the introduction of the compute shader in OpenGL ES 3.1 it should be very possible to perform more steps of the JPEG decoding on the GPU. Since compute shaders will supposedly be similar to CUDA kernels, the possibility to reuse much of the work done in CUDA could show instrumental in increasing performance on mobile.

# 10

# Conclusion

This thesis have shown that it is possible to efficiently partially decode images on the GPU. A standalone and Chromium implementation have been implemented, where the upsampling and color transformation have been moved from the CPU to the GPU.

The implementation have shown to use less memory both on the CPU and GPU, and less computation on the CPU. In the case of big images, the Chromium implementation have shown savings of 4x the theoretical memory savings because of removed overhead. Power consumption was measured, which showed an overall lower power consumption, but this will need to be further tested.

# 11

# References


**[Ahmed et al. 1974]** Ahmed, N., Natarajan, T., and Rao, K. (1974). Discrete cosine transform. *Computers, IEEE Transactions on*, C-23(1):90-93.

**[Akenine-Moller and Strom 2008]** Akenine-Moller, T and Strom, J. (2008). Graphics processing units for handhelds. *Proceedings of the IEEE*, 96(5):779-789.

**[Alakuijala 2012]** Alakuijala, J. (2012). Webp compression comparison. `https://developers.google.com/speed/webp/docs/webp_lossless_alpha_study?csw=1#results` (15 May. 2014).

**[Alexa 2014]** Alexa (2014). Alexa top sites. `http://www.alexa.com/topsites` (15 May. 2014).

**[Apple Inc 2014]** Apple Inc. (2014). The Webkit open source project. `http://www.webkit.org` (15 May. 2014).

**[Bankoski et al. 2011]** Bankoski, J., Wilkins, P., and Xu, Y. (2011). Technical overview of vp8, an open source video codec for the web. In *ICME*, pages 1-6. IEEE.

**[Commander 2014]** Commander, D. (2014). Virtualgl - libjpeg-turbo. `http://libjpeg-turbo.virtualgl.org/` (15 May. 2014).

**[CompuServe Incorporated 1990]** CompuServe Incorporated (1990). Graphich Interchange Format. `http://www.w3.org/Graphics/GIF/spec-gif89a.txt` (15 May. 2014).

**[Cormen et al. 2001]** Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

**[Deutsch 1996]** Deutsch, P. (1996). Deflate compressed data format specification version 1.3.

**[Fang et al. 2005]** Fang, B., Shen, G., Li, S., and Chen, H. (2005). Techniques for efficient DCT/IDCT implementation on generic GPU. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, page 1126â1129. IEEE.

**[Gartner and Gartner 2013]** Gartner, R. and Gartner, J. (2013). Gartner says smartphone sales grew 46.5 percent in second quarter of 2013 and exceeded feature phone sales for first time. `http://www.gartner.com/newsroom/id/2573415` (15 May. 2014).

**[Gonzalez and Woods 2001]** Gonzalez, R. C. and Woods, R. E. (2001). *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

**[Goodnight et al.2005]** Goodnight, N., 0003, R. W., and Humphreys, G. (2005). Computation on programmable graphics hardware. *IEEE Computer Graphics and Applications*, 25(5):12-15.

**[Google Inc. 2012]** Google Inc. (2012). The evolution of the web. `http://www.evolutionoftheweb.com/` (15 May. 2014).

**[Google Inc. 2013]** Google Inc. (2013). Nexus 5. `https://www.google.se/nexux/5` (15 May. 2014).

**[Google Inc. 2014a]** Google Inc. (2014a). WebP. `https://developers.google.com/speed/webp/?csw=1` (15 May. 2014).

**[Google Inc. 2014b]** Google Inc. (2014b). The Chromium project. `http://www.chromium.org/` (15 May. 2014).

**[Google Inc. 2014c]** Google Inc. (2014c). Renderscript. `http://developer.android.com/guide/topics/renderscript/index.html` (15 May. 2014).

**[Google et al. 2014]** Google, Mozilla, New Relic, OâReilly Media, Etsy, Radware, dynaTrace Software, Torbit, Instart Logic, and Catchpoint Systems (2014). Http archive - interesting stats. `http://httparchive.org/interesting.php` (15 May. 2014).

**[Joint Photographic Experts Group 2014]** Joint Photographic Experts Group (2014). Independent JPEG Group. `http://www.ijg.org` (15 May. 2014).

**[Khronos Group Inc. 2009]** Khronos Group Inc. (2009). *OpenGL ES Common Profile Specification Version 2.0.24*.

**[Khronos Group Inc. 2014]** Khronos Group Inc. (2014). Opengl compute shader. `http://www.opengl.org/registry/specs/ARB/compute_shader.txt` (15 May. 2014).

**[Khronos OpenCL Working Group 2014]** Khronos OpenCL Working Group (2014). *The OpenCL Specification*.

**[Klein and Wiseman 2003]** Klein, S. T. and Wiseman, Y. (2003). Parallel huffman decoding with applications to jpeg files. *THE COMPUTER JOURNAL*, 46:487-497.

**[Lakhani 2004]** Lakhani, G. (2004). Optimal huffman coding of dct blocks. *Circuits and Systems for Video Technology, IEEE Transactions on*, 14(4):522-527.

**[Li and Kim 2000]** Li, R. Y. and Kim, J. (2000). Image compression using fast transformed vector quantization. In *AIPR*, pages 141-145. IEEE Computer Society.

**[Imagination Technlogies Limited 2014]** Limited, Imagination Technlogies Limited (2014). PowerVR. `http://community.imgtec.com/developers/powervr/` (15 May. 2014).

**[Munshi et al. 2008]** Munshi, A., Ginsburg, D., and Shreiner, D. (2008). *OpenGL(R) ES 2.0 Programming Guide*. Addison-Wesley Professional, 1 edition.

**[Nguyen and Redinbo 2002]** Nguyen, C. and Redinbo, R. G. (2002). Fault tolerance technique in huffman coding applies to baseline jpeg.

**[NVIDIA Corporation 2013a]** NVIDIA Corporation. (2013a). Cuda programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide` (15 May. 2014).

**[NVIDIA Corporation 2013b]** NVIDIA Corporation. (2013b). NVIDIA Tegra 4. `http://www.nvidia.com/object/tegra-4-processor.html` (15 May. 2014).

**[NVIDIA Corporation 2014]** NVIDIA Corporation. (2014). NVIDIA tegra mobile processor. `http://www.nvidia.com/object/tegra.html` (15 May. 2014).

**[Obukhov and Kharlamov 2008]** Obukhov and A. Kharlamov (2008). Discrete Cosine Transform for 8x8 Blocks with CUDA. `http://www.nvidia.com/content/cudazone/cuda_sdk/Image_Video_Processing_and_Data_Compression.html`

**[Opera Software 2014a]** Opera Software (2014a). About opera software. `http://www.opera.com/about` (15 May. 2014).

**[Opera Software 2014b]** Opera Software (2014b). Opera software mobile faq. `http://www.opera.com/help/mobile/faq` (15 May. 2014).

**[Opera Software 2014c]** Opera Software (2014c). Opera software mobile. `http://www.opera.com/mobile` (15 May. 2014).

**[Pettersson 2013]** Pettersson, P. (2013). Texture compression - google groups discussion. `https://groups.google.com/a/chromium.org/forum/#!topic/graphics-dev/6kX_UYXK10c` (15 May. 2014).

**[Ranganathan and Henriques 1993]** Ranganathan, N. and Henriques, S. (1993). High-speed vlsi designs for lempel-ziv-based data compression. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 40(2):96-106.

**[Smith 1997]** Smith, Steven W. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA.

**[Qualcomm Inc. 2014]** Qualcomm Inc. (2014). Mobile technology. `http://www.qualcomm.com/snapdragon` (15 May. 2014).

**[Rao and Yip 1990]** Rao, K. R. and Yip, P. (1990). *Discrete Cosine Transform: Algorithms, Advantages, Applications*. Academic Press Professional, Inc., San Diego, CA, USA.

**[Roelofs 1999]** Roelofs, G. (1999). *PNG - the definitive guide: creating and programming portable network graphics.* O'Reilly.

**[Sodsonget al. 2013]** Sodsong, W., Hong, J., Chung, S., Lim, Y.-K., Kim, S.-D., and Burgstaller, B. (2013). Dynamic partitioning-based jpeg decompression on heterogeneous multicore architectures. *CoRR*, abs/1311.5304.

**[TestMy Net LLC 2014]** TestMy Net LLC (2014). Sweden speed test. `http://testmy.net/country/se` (13 April. 2014).

**[United States Securities And Exchange Comission 2012]** United States Securities And Exchange Comission (2012). Form S-1 Registration Statement (Facebook Inc.). `http://www.sec.gov/Archives/edgar/data/1326801/000119312512034517/d287954ds1.htm` (15 May. 2014).

**[Wallace 1991]** Wallace, G. K. (1991). The jpeg still picture compression standard. *Communications of the ACM*, pages 30-44.

**[Yam 2005]** C.Y. Yam. (2005) Optimizing Video Compression for Intel Digital Security Surveillance applications with SIMD and Hyper-threading Technology. Intel Corporation.

**[Yang and Welch 2002]** Yang, R. and Welch, G. (2002). Fast image segmentation and smoothing using commodity graphics hardware. *J. Graphics, GPU, & Game Tools*, 7(4):91-100.