TAMPERE UNIVERSITY OF TECHNOLOGY

JESSE HAKANEN

ACCELERATING IMAGE PROCESSING PIPELINE ON MOBILE DEVICES USING GPU

Master of Science Thesis

Examiner: Professor Tommi Mikkonen
Examiner and topic approved by
the Council of the Faculty of
Computing and Electrical Engineering
on April 9th, 2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
HAKANEN, JESSE: Accelerating Image Processing Pipeline on Mobile Devices Using GPU
Master of Science Thesis, 54 pages
June 2014
Major: Software Engineering
Examiner: Professor Tommi Mikkonen
Keywords: GPU, image processing pipeline, mobile camera, Windows Phone, software framework

Majority of current mobile devices include a camera. To meet the form-factor and price requirements, the camera is typically built from inexpensive components which causes defects such as noise, dead pixels and distortions. An acceptable image quality is achieved by processing algorithms which together form an image processing pipeline. Hardware implementations typically offer the best performance and the lowest power consumption, but software implementations can be used to cut costs and maximize the flexibility of the system. However, software implementations may be too ineffective and cause overheating. One alternative to pure hardware and software implementations is the GPU.

In this thesis, a generic framework for GPU-based image processing is implemented. The framework simplifies algorithm implementation and organization significantly, and hides some hardware limitations that current mobile GPUs have. The framework is evaluated by implementing an image processing pipeline which consists of seven typical algorithms, and by comparing its performance, memory consumption, power consumption and heat generation to an equivalent CPU implementation. This thesis also discusses optimizations that can be done for the GPU implementation especially on mobile devices.

The experiments show that the GPU implementation is able to process images over 40% faster than a multi-threaded CPU implementation. Biggest performance gains were seen in algorithms that were computationally heavy. The GPU is also able to process the same image with much less power consumption. On the other hand, the GPU proved to produce more heat in the test device. With the tested pipeline, also memory consumption was higher than with an optimized CPU implementation.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
HAKANEN, JESSE: Kuvankäsittelyliukuhihnan kiihdyttäminen mobiililaitteissa GPU:ta käyttämällä
Diplomityö, 54 sivua
Kesäkuu 2014
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Tommi Mikkonen
Avainsanat: GPU, kuvankäsittelyliukuhihna, mobiilikamera, Windows Phone, sovelluskehys

Useimmissa mobiililaitteissa on nykyään sisäänrakennettuna kamera. Jotta laitteiden koko- ja hintavaatimukset täyttyisivät, rakennetaan kamerat usein halvoista komponenteista, mikä aiheuttaa kuvaan virheitä, kuten kohinaa, kuolleita pikseleitä ja vääristymiä. Kelvollinen kuvanlaatu saavutetaan kuvankäsittelyalgoritmeilla, jotka yhdessä muodostavat kameran kuvankäsittelyliukuhihnan. Laitteistototeutukset mahdollistavat tyypillisesti parhaan tehokkuuden ja matalimman virrankulutuksen, mutta ohjelmistototeutuksilla voidaan saavuttaa pienempi hinta ja maksimoida järjestelmän joustavuus. Ohjelmistoon perustuvat ratkaisut saattavat kuitenkin olla liian tehottomia ja aiheuttaa laitteen ylikuumenemista. Yksi vaihtoehto pelkästään laitteistoon ja ohjelmistoon perustuville ratkaisuille on GPU.

Tässä työssä toteutetaan yleiskäyttöinen sovelluskehys GPU-kuvaprosessointia varten. Sovelluskehys yksinkertaistaa algoritmien toteutusta ja järjestelyä huomattavasti sekä piilottaa tiettyjä rajoituksia, joita nykyisten mobiiliaitteiden GPU:issa on. Sovelluskehystä arvioidaan toteuttamalla seitsemästä tyypillisestä algoritmista koostuva kuvankäsittelyliukuhihihna, jonka tehokkuutta, muistinkulutusta, virrankulutusta ja lämmöntuottoa verrataan vastaavaan CPU-toteutukseen. Työssä esitellään myös muutamia tapoja optimoida GPU-pohjaista prosessointia erityisesti mobiililaitteissa.

Työssä suoritetut mittaukset osoittavat, että GPU pystyy käsittelemään kuvia yli 40 % nopeammin kuin useaa ydintä hyödyntävä CPU-toteutus. Suurimmat tehokkuusparannukset saavutettiin algoritmeissa, jotka ovat laskennallisesti raskaita. Lisäksi GPU pystyy käsittelemään saman kuvan paljon pienemmällä virrankulutuksella. Toisaalta mittaukset osoittivat GPU:n tuottavan enemmän lämpöä testilaitteessa. Testatulla kuvankäsittelyliukuhihnalla myös muistinkulutus oli korkeampi kuin optimoidussa CPU-toteutuksessa.

PREFACE

This thesis was carried out while working in Nokia Smart Devices, Imaging organization in Tampere, Finland between October 2013 and May 2014.

I would like to thank my colleagues in the algorithm and middleware team for creating an inspiring atmosphere and providing help when needed. Special thanks to Markus Vartiainen who supervised the thesis on behalf of Nokia and assisted me throughout the whole process, and Christian Mäkelä who helped with algorithm implementation and GPU-related issues. I also want to thank professor Tommi Mikkonen from Tampere University of Technology for supervising the work and providing useful tips during the writing process.

Tampere, May 2nd, 2014

Jesse Hakanen

CONTENTS

TERMS AND DEFINITIONS

API	Application Programming Interface
CCD	Charge-Coupled Device
CFA	Color Filter Array
CMOS	Complementary Metal-Oxide-Semiconductor
CUDA	Compute Unified Device Architecture
GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
SoC	System on Chip

1. INTRODUCTION

Digital cameras are nowadays included in a wide range of mobile devices from entry level smartphones to the most high-end models. For many people, mobile devices have replaced a separate point-and-shoot camera as the main device for capturing images and videos. Therefore, image quality has become an important aspect where device manufacturers try to differentiate from others. To achieve an acceptable image quality, image processing is needed in all digital still cameras. The combination of algorithms is called an image processing pipeline which is a key part in construction of the final image.

To meet the requirements for an inexpensive consumer device, mobile cameras are typically really small and cheap. In addition, the requirement for capturing bigger and more detailed images has lead into a really small pixel size. These kind of cameras typically suffer from several defects such as noise, dead pixels and distortions which is why mobile image processing pipelines often require a large number of algorithms to compensate the errors. In addition, it is desired to be able process the image as efficiently as possible to allow a short latency between captures. Hardware implementations typically offer the best performance and the lowest power consumption, but they tend to be too expensive and limited when the target is getting the best possible image quality at low price. Software implementations are much more flexible and cost-effective, but their ineffectiveness may lead into poor performance and cause problems with overheating.

One option for implementing the image processing pipeline is using a graphics processing unit (GPU). The GPU offers an interesting option because it lies between pure hardware and software implementations: it is a separate chip that is available on all modern mobile devices, but it also has an adequate level of programmability. Originally, GPUs implemented a fixed graphics pipeline which made it appropriate for accelerating simple 3D graphics. During recent years, GPUs have evolved into highly programmable chips which has made them an interesting option for general computing as well. However, feature-wise mobile GPUs lag few years behind their desktop counterparts which adds some limitations for the implementation.

The purpose of this thesis is to study the use of a GPU as an image processor in mobile devices using Windows Phone platform. This includes studying how well currently available graphics hardware and interfaces suit for image processing, what

kind of limitations there are and how these limitations can be solved. The GPU is compared to a basic software implementation in terms of processing performance, memory consumption, heat generation and power consumption.

This thesis begins with an introduction to digital camera systems and image processing pipelines in chapter 2. Chapter 2 also introduces an example pipeline which will be later used when the GPU implementation is evaluated. Chapter 3 provides a short introduction to the current programmable graphics pipelines, concentrating on the image processing point of view. In addition, chapter 3 provides ways to implement different types image processing algorithms by utilizing the standard programmable parts of the graphics pipeline. Chapter 4 presents a generic framework for GPU-based image processing. The framework is used to hide some of the limitations that current mobile GPUs have. It also provides a higher abstraction level for implementing algorithms on the GPU. Chapter 5 shows the available parameters that can be used to tune the framework implementation. Chapter 5 also contains measurements and calculations that were made to optimize those parameters. In chapter 6, the implemented example pipeline on the GPU is evaluated by comparing it to a CPU implementation. Finally, chapter 7 concludes the thesis.

2. DIGITAL CAMERA SYSTEM

Digital camera system is a combination of hardware and software that captures light and converts it into a digital representation. This chapter familiarizes the reader into the subject by providing a brief overview of the hardware components and required processing algorithms in a digital still camera. The camera system represented in this chapter is based on [1], unless stated otherwise.

2.1 Hardware

Digital still cameras have evolved considerably after the first consumer grade digital camera, Casio QV-10, was released in 1994 [1, p. 9]. Today digital cameras are embedded in a wide range of consumer products such as cars, toys and smartphones. While digital cameras have become more and more affordable for average consumers, the demand for capturing high quality images on inexpensive consumer products, especially smartphones, has increased.

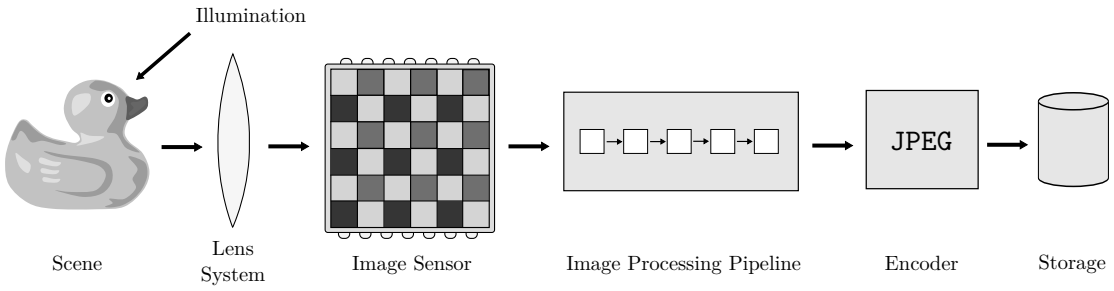


Figure 2.1: The components of a digital still camera.

The digital camera system is typically formed by a lens system, an image sensor and an image processing pipeline as illustrated in figure 2.1. When an image is captured, the scene is first illuminated by the camera flash or by ambient lighting. The light beams then travel through a lens system and reach an image sensor. The image sensor creates a digital representation of the captured light. Raw image data is read from the sensor and transferred into the image processing pipeline. Finally, the processed image is encoded into a compressed format, such as JPEG, and saved to a memory card or another storage device.

2.1.1 Lens System

The purpose of a lens system is to control how light reaches the image sensor. The lens system typically consists of multiple lens elements, a fixed or adjustable aperture and an infrared filter. Figure 2.2 shows the components in a typical mobile camera lens. The infrared filter is used to attenuate the entering infrared signals from reaching the sensor because the sensor typically has significant sensitivity in infrared range [1].

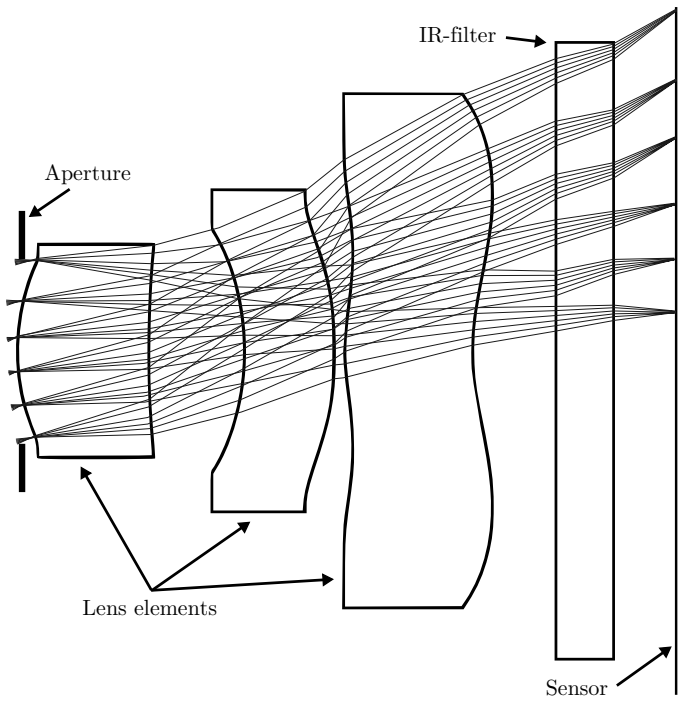


Figure 2.2: A typical mobile camera lens system [2].

The quality of optics in the lens system has a significant impact on image quality. Usually poor optics can be seen as softness in resulting images regardless of how many pixels the image sensor has. In addition to optics quality, the lens system has two other aspects that have an impact on the image: *aperture* and *focal length*. [1, p. 21]

The focal length of an optical system is the distance from the lens to the sensor when the lens is focused on infinity [1, p. 33]. When a light ray travels through a lens, it is refracted so that the object appears sharpest at the focal point. The focal length determines how big the angle of view of the lens is, and thus how big the magnification of the lens is. A shorter focal length leads into stronger refraction and therefore makes the object smaller on the sensor.

The aperture controls how wide the entering of the lens system is. This has an effect on how much light enters the lens system, but also affects how the image is

focused on the sensor. A depth of field is the area (depth) within which the captured object is in focus. With wide aperture, the depth of field becomes narrow and only the captured object is on focus, while with narrow aperture the whole scene can be in focus. The size of the aperture is defined as an F-number, smaller number designating wider aperture [1, p. 25].

Mobile cameras typically have a really short focal length because of the small form factor. A short focal length makes mobile phone cameras more prone to multiple distortions, including chromatic aberration and lens vignetting. These distortions must be corrected later in the image processing pipeline.

When a light ray enters the lens, it is refracted in an angle depending on its wavelength. This phenomenon causes a distortion called chromatic aberration. In chromatic aberration, the colors from the same source reach different pixels at the sensor. With short focal lengths the refraction is higher which increases the spreading of the rays. Chromatic aberration can be significantly reduced with multiple lenses, but it can never be completely removed. [1, p. 294]

Lens vignetting appears as a radial darkening and decreased sharpness of the image towards the corners of the frame. The biggest intensity is captured at the center because light reaching that part of the sensor has a right angle. The angle decreases towards the borders which causes radial dimming. With higher refraction on short focal lengths, the difference between angles of different light rays becomes bigger which has an increasing impact on lens vignetting. [1, p. 46]

2.1.2 Image Sensor

An image sensor is a device that converts light formed by the lens into measurable signals. The image sensor consists of an array of light-sensitive components called pixels. Each pixel is sensitive to light and able to convert the amount of captured photons into an electrical voltage. In digital still cameras, the sensor materials are chosen carefully, so that only photons with wavelengths of visual light (380 nm to 780 nm) enter the photosensitive parts [1, p. 54].

Color Separation

The sensor elements only detect the intensity of light falling into the detector. To measure the intensity of each color channel separately, the arriving photons must be somehow separated based on their wavelength. The predecessors of modern digital still cameras used three sensors and a beam splitter separate the light into three optical paths. [3, pp. 3-4]

However, the sensor is usually the most expensive component in a digital still camera, so three sensor systems are not used in consumer grade cameras [3, p. 4].

To reduce expenses, only one sensor covered with a mosaic of color filters is used instead. Each filter passes through light with a specific wavelength range. These color filter arrays (CFA) are usually based on red, green and blue components, although configurations based on more components exist [3]. Because each pixel does not contain the intensity of all colors, the missing information must be interpolated later in the image processing pipeline.

The most widely used CFA is the Bayer pattern which uses a symmetrical grid of four color components as demonstrated in figure 2.3. The filter array contains 25 % red , 25 % blue and 50 % green pixels. The dominant amount of green is selected because the human eye derives image details primarily from the green portion of the visible light spectrum [1, p. 63].

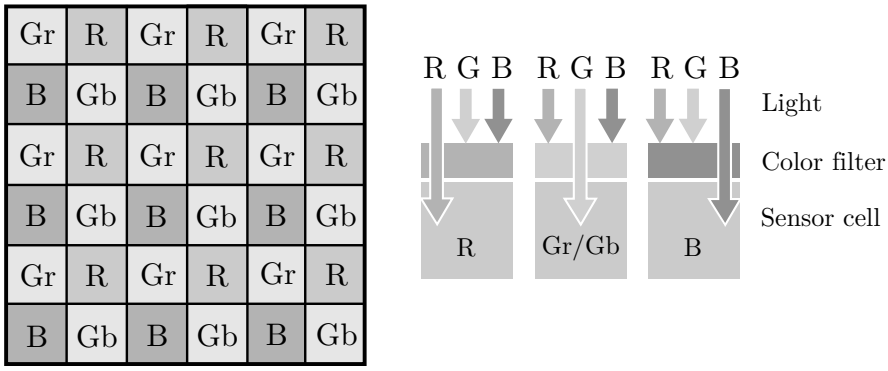


Figure 2.3: A Bayer pattern color filter array and the structure of different color pixels.

Sensor Types

Two major image sensor technologies, Charge-Coupled Device (CCD) and Complementary Metal-Oxide-Semiconductor (CMOS), are generally used in imaging devices. The difference between these technologies is how light is converted into voltage and how raw data is read from the sensor [4].

A CCD sensor transforms light into a simple analog signal while a CMOS sensor integrates multiple features, such as analog to digital conversion, into one chip. As a result, CMOS circuits are smaller, consume less power and are cheaper to manufacture than a CCD sensor combined with a separate analog to digital converter chip. CMOS sensors are popular especially in mobile devices and other low power consumer devices. [4]

In modern smartphones, the resolution of sensors typically varies between 5 and 20 megapixels, although mobile sensors up to 41 megapixels exist. Generally, more megapixels leads into bigger and detailed images, but it also means that individual pixels become smaller. This can have a negative effect in image quality especially in

low light conditions since smaller pixels gather less photons. Longer exposure times can be used to capture brighter images, and several technologies such as optical image stabilization have been integrated into mobile cameras to keep the device stable during capture. Long exposure times are mainly suitable for capturing objects with very little movement. Another way to capture brighter images with small pixels is to apply analog or digital gain to the image data. However, gaining the signal also increases noise which must be later compensated in the image processing pipeline [1, p. 89].

2.2 Image Processing Pipeline

To produce high-quality digital images from raw image data, a large amount of processing is needed. The purpose of an image processing pipeline is to fix or minimize the errors and distortions such as noise, dead pixels and chromatic aberration that exist in raw data read from the sensor. Also, the difference between the way image sensors capture light and the human visual system sees things must be compensated to make the captured scene look natural to the human eye.

2.2.1 Image Formats

In digital image processing, different image formats are useful for different types of filtering and processing algorithms. For this reason, the image usually goes through multiple format conversions during the image processing pipeline. This section introduces some of the most widely used formats and provides information about where they are used and how they are usually stored in the system memory.

Raw Bayer

Raw Bayer is the initial format of the image data when it has been read from the sensor. In figure 2.3, the top left corner has a Gr color component, but the order of the components can be also different. For example, in RGrGbB Bayer order, the first line contains red and green components and the second line contains green and blue components. Terms Gr and Gb are used to identify whether the green pixel is in RG or GB line.

Raw Bayer data typically has a bit depth of 6, 7, 8, 10 or 12 bits [5]. A naive way to store the raw data into memory would be by adding extra padding to each value. For example, 10 bit data could be padded with 6 bits to make the values follow the 8-byte boundaries and therefore easier to be stored to the system memory. However, this would mean that 6 bits of memory would be wasted per pixel.

The Compact Camera Port 2 (CCP2) specification defines an interface between a digital camera sensor and a mobile phone engine. In CCP2, the Raw Bayer data

is packed to keep memory usage and bandwidth as low as possible. For example, 10-bit data is packed into continuous memory so that four pixels are stored into five bytes. The eight most significant bits of each pixel are stored into the first four bytes, and the remaining two bits of each pixel into the last byte. This packing scheme is shown in figure 2.4. [5]

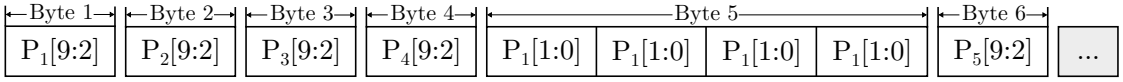


Figure 2.4: Packing 10-bit Raw Bayer data into memory [5].

RGB

RGB is an additive color space where red, green and blue are added together to reproduce a single color. Typically full-color images are stored in 24 bits per pixel where the intensity of each color channel is presented in eight bits. This image format is also called RGB888 [5]. Typically color components in RGB888 are stored in interleaved format, so that all three colors of each pixel are stored in adjacent memory addresses. Another way to store RGB888 data is to use a separate buffer for each color.

YCbCr

In YCbCr image format, the image information is divided into luminance component (Y) and two chrominance components (Cb and Cr). Y contains a grayscale presentation of the image while chrominance contains the color information.

The human eye is much more sensitive to differences in luminance than in chrominance information [1, pp. 233]. This can be utilized to reduce the bandwidth and memory usage when the image is processed or stored in YCbCr format. For example, the chrominance components can be presented with a smaller resolution than the luminance component [1, pp. 233]. There are three common chroma subsampling patters used: 4:4:4, 4:2:2 and 4:2:0. In 4:4:4 sampling, each channel has the same resolution. In 4:2:2 sampling, the horizontal resolution of the chrominance data is half of the resolution of the luminance data. In 4:2:0, the chrominance resolution is half of the luminance resolution in both vertical and horizontal directions. These chroma subsampling patters are illustrated in figure 2.5.

YCbCr is also used by the JPEG image format [7]. Since images captures by digital still cameras are usually stored in JPEG, the output of an image processing pipeline is typically YCbCr. YCbCr is also useful for processing algorithms that operate only on luminance or chrominance information. For example, there could

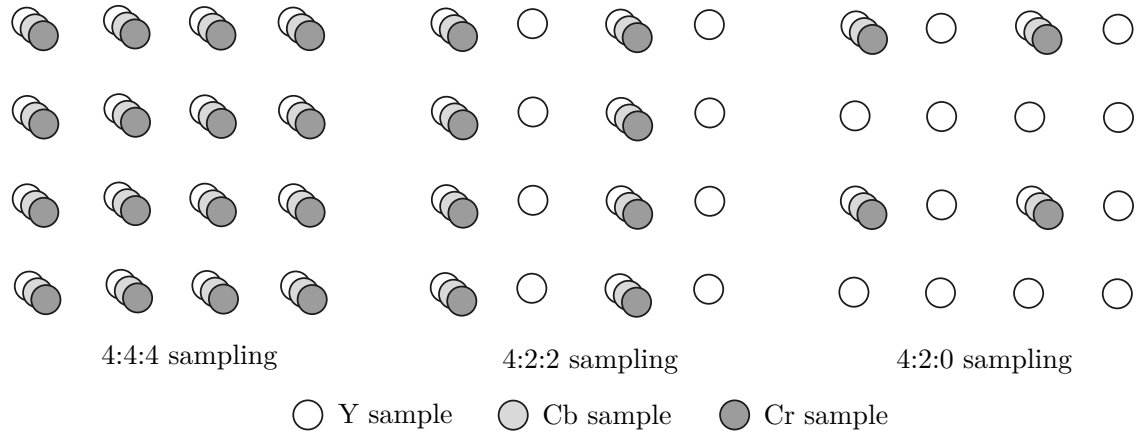


Figure 2.5: Different YCbCr chroma subsampling patterns [6, p. 18].

be separate noise reduction algorithms for luminance and color data and YCbCr provides one way to divide the information.

Since chrominance data can be subsampled, luminance and chrominance data are usually stored in separate buffers. To use only one continuous memory buffer, these buffers can be located next to each other in memory. For example, NV12 is a 4:2:0 format where all Y samples appear first in memory, followed by interleaved Cb and Cr samples [8]. The memory layout of an NV12 image is illustrated in figure 2.6.

Increasing memory addresses →

0x00	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	0x07
0x08	Y_9	Y_{10}	Y_{11}	Y_{12}	Y_{13}	Y_{14}	Y_{15}	Y_{16}	0x0F
0x10	Y_{17}	Y_{18}	Y_{19}	Y_{20}	Y_{21}	Y_{22}	Y_{23}	Y_{24}	0x17
0x18	Y_{25}	Y_{26}	Y_{27}	Y_{28}	Y_{29}	Y_{30}	Y_{31}	Y_{32}	0x1F
0x20	Cb_1	Cr_1	Cb_2	Cr_2	Cb_3	Cr_3	Cr_4	Cr_4	0x27
0x28	Cb_5	Cr_5	Cb_6	Cr_6	Cb_7	Cr_7	Cr_8	Cr_8	0x2F

Figure 2.6: The memory layout of an NV12 image.

2.2.2 Digital Image Processing Techniques

Digital image processing can be performed either in spatial domain or frequency domain [9, p. 269]. Spatial domain processing means that the output pixel is calculated based on values in the original image color space. Algorithms that operate

on frequency domain use Fourier or wavelet transform to convert the image into frequency domain where the actual processing is performed. Spatial domain algorithms are more common in digital still cameras since the transform to frequency domain can be too complex for low power processors.

In spatial domain, the output pixel can be calculated either from a single pixel or from multiple surrounding pixels. Single pixel algorithms have the basic form

$$s_{x,y} = T(r_{x,y}), \tag{2.1}$$

where $r_{x,y}$ is the input pixel value, T is a transformation function and $s_{x,y}$ is the output pixel at the same position as the input pixel [10, pp. 77]. Images captured by digital still cameras typically have tens of millions of pixels, so using a complex transformation function can be slow. In single pixel algorithms, a pre-calculated *lookup table* can be used instead. A lookup table is a one-dimensional array containing an output value for each possible input value. For example, with 10-bit data the calculated lookup table would contain $2^{10} = 1024$ values. In the lookup table, all possible output values are calculated in advance. The input value is used as an index to lookup the actual value which can be much faster than calculating the value of the transformation function for each pixel separately.

Instead of using only one pixel as an input, the output pixel can be also calculated from multiple surrounding pixels. The selection of surrounding pixels is specified by a *filtering kernel*. The currently processed pixel is typically located at the center of the filtering kernel. The output is calculated by taking each surrounding pixel, multiplying it by a coefficient defined in the kernel and accumulating the multiplied values. The use of a filtering kernel is illustrated in figure 2.7.

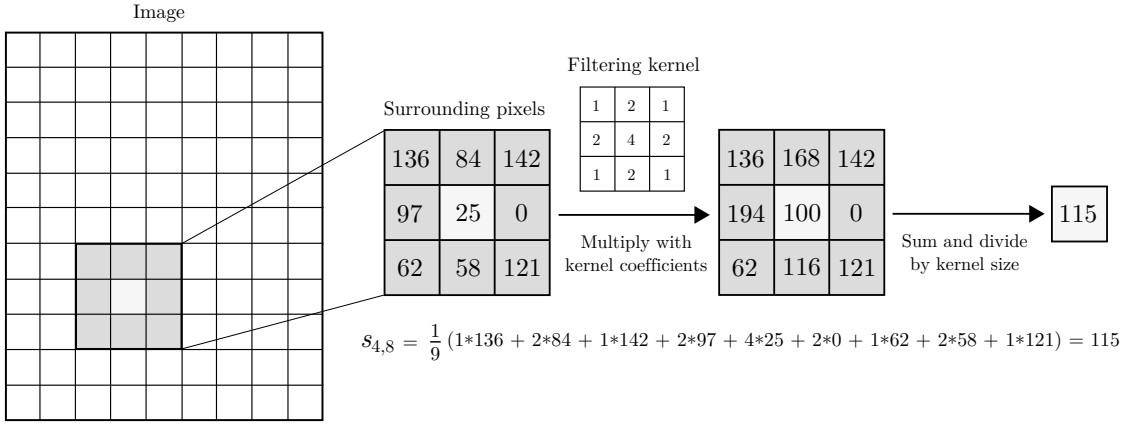


Figure 2.7: A simple blur effect using a 3x3 filtering kernel.

An important feature to take into account when using a filtering kernel is how to process the output pixel when the center of the filter approaches the border of the

image. A simple solution would be cropping border pixels after processing. If the maximum kernel size used in the processing pipeline is $N*N$, the required crop from each side of the image would be $(N-1)/2$ pixels [10, p. 119]. However, cropping is not often a desired effect since the size of the output image would be reduced. Another way to handle border pixels is adding padding pixels outside of the image area. After the image has been processed, the padding will removed which means that the size of the image stays intact. A simple way to add padding is to use zeros or other predefined values. The padding could also be clamped to value of the border pixel. Even more versatile way is to mirror the border pixels which can be done by mirroring on the border of the image or by mirroring on the border pixel. The different border processing methods are illustrated in figure 2.8.

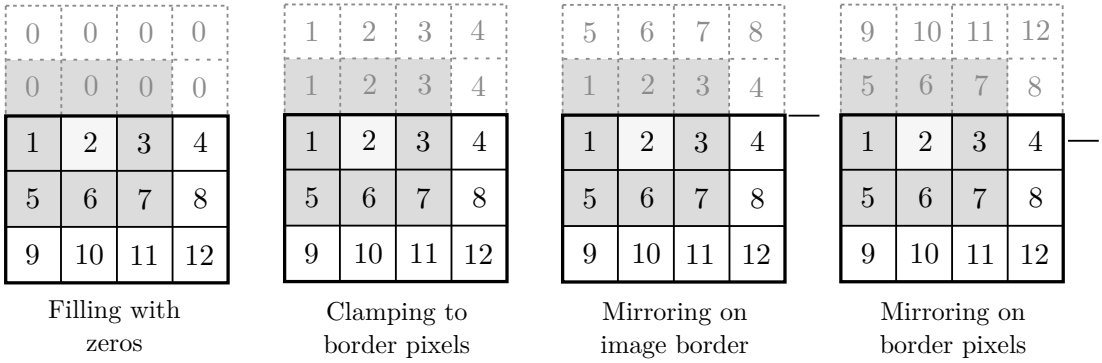


Figure 2.8: Different methods for processing border pixels when a processing kernel is used.

The applied border processing method depends on the image format. For example, in RGB, any method can be used, although some methods may lead into better results than others. However, in raw Bayer format any method except mirroring on the border pixel would break the Bayer order and therefore cause pixels with different colors to interact with each other. This could lead into artifacts in the final image.

2.2.3 Processing Algorithms

Although image processing pipelines have existed in digital still cameras for years, the actual processing algorithms are usually kept secret by the camera manufacturers [11]. However, there are many algorithms that typically exist in some form in all processing pipelines. Some of these algorithms are illustrated in figure 2.9.

In addition to algorithms in figure 2.9, image processing pipelines typically include algorithms for noise reduction, defective pixel correction and edge enhancement [11][13]. Because these algorithms usually require advanced processing and the

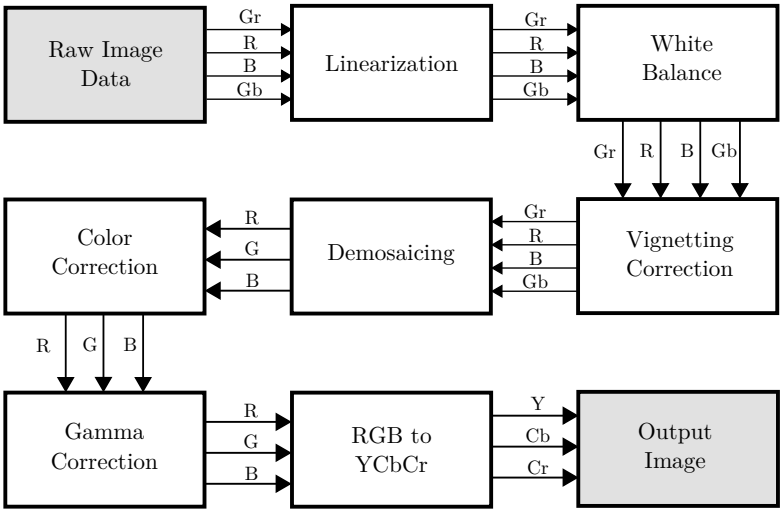


Figure 2.9: Algorithms in a typical image processing line [11].

actual implementations vary by manufacturer, we will overlook them in this thesis and focus only on the most necessary algorithms that are typically used in mobile devices. Also, the order of the algorithms can be different from the order above. For example, white balance or vignetting correction could be done after demosaicing.

Linearization. Even though there is no light falling into the image sensor, the measured intensity in a pixel is greater than zero. This effect is called *dark current* and is caused by charge leakages that occur due to thermal effects in sensor cells [1, p. 68]. The measured pixel value in completely dark environment is called the *black level*. Similarly, the *white level* denotes the maximum intensity that a pixel can reliably measure.

Linearization, or auto level, compensates the dark current by stretching the data between black level and white level to fill the entire input range. This can be done by using a histogram calculated from the raw data and finding points α and β that correspond to 0.1% and 99.9% of the histogram, respectively. The linearized value of each pixel can then be calculated from

$$s_{x,y} = (r_{x,y} - \alpha)\frac{2^b - 1}{\beta - \alpha}, \tag{2.2}$$

where b is the bit depth of the image, $r_{x,y}$ is the input value and $s_{x,y}$ is the linearized output value. After linearization, α represents black and therefore has a value of 0, while β represents white and has a value of $2^b - 1$. [11]

White Balance. The human eye automatically adapts to lightning conditions and recognizes the color of objects as they were observed in typical lightning conditions. For example, a white paper looks white to the human eye even though there

is a tungsten light source giving a yellowish color cast to the paper. This feature is called chromatic adaptation [1, p. 215]. An imaging sensor does not have this feature, which means that the colors in a scene captured by the sensor look different to what our eyes see. For raw Bayer data, white balance is typically fixed with the following diagonal transform [14, p. 94]:

$$\begin{bmatrix} R \\ Gr \\ Gb \\ B \end{bmatrix}_{out} = \begin{bmatrix} r_{wb} & 0 & 0 & 0 \\ 0 & g_{wb} & 0 & 0 \\ 0 & 0 & g_{wb} & 0 \\ 0 & 0 & 0 & b_{wb} \end{bmatrix} \begin{bmatrix} R \\ Gr \\ Gb \\ B \end{bmatrix}_{in} . \tag{2.3}$$

There are multiple approaches for calculating parameters r_{wb}, g_{wb} and b_{wb}. The optimal algorithm may be a combination of multiple algorithms based on the current lightning conditions. One thing that makes white balance parameter estimation particularly difficult is the fact that real life scenes usually contain multiple different light sources. [1, p. 216]

Vignetting Correction. Vignetting correction is needed to correct the gradual attenuation of brightness towards the border of the image. To estimate the vignetting effect on each pixel, a mathematical presentation, called vignetting function, can be calculated. The most straightforward way to calculate the vignetting function is to capture a calibration image. However, the calculated function would only be valid for similar conditions and camera settings [15].

Another way to calculate the vignetting function is to capture multiple overlapping images of the same scene [16]. In digital still cameras containing an electronic viewfinder, the images and image statistics captured from the viewfinder stream can be utilized in calculations that require multiple images. There are also methods based on single image only. Zheng et al. [15], for example, present method for calculating the vignetting function by doing a content-based segmentation for the image data.

Demosaicing. As already mentioned in section 2.1.2, raw data contains only one color component for each pixel. The purpose of demosaicing, or CFA interpolation, is to interpolate the missing color information so that each pixel contains all three color components that are typically red, green and blue. The most straightforward way to implement demosaicing is to use bilinear interpolation. In bilinear interpolation, the missing two color components are calculated as an average of the surrounding pixels having the specific color. However, a simple bilinear interpolation may increase noise, aliasing and other color artifacts [14, p. 131-132]. Therefore, more advanced demosaicing algorithms have been developed [17][18].

Color Correction. Color correction transforms the image data from sensor color space to a standard RGB color space, such as sRGB. This transformation is required

because the spectral response of the sensor does not usually match the desired output color space [14, p. 103]. Color correction also fixes cross-color bleeding that happens in the color filter array [1, p. 232]. Color correction is typically implemented as a 3x3 matrix operation

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix}_{out} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}_{in}, \tag{2.4}$$

where coefficients a_n, b_n and c_n are calculated based on a captured standard image such as the Macbeth ColorChecker [11][9, p. 553].

Gamma Correction. Gamma correction transforms the linear response of an image sensor to match the non-linear response of a computer monitor [1, p. 233]. As a result, dark areas in the image become brighter than they were captured by the sensor. The reason for storing data this way is to get more details to the dark areas of the image where the human eye has the biggest sensitivity to differences [9, p. 271]. Gamma correction can be implemented with a gamma curve which is defined by equation

$$s_{x,y} = r_{x,y}^{\gamma}, \tag{2.5}$$

where γ is referred to as gamma [10, pp. 80]. Lookup tables can be used to remove the excessive amount of processing required for the power calculations.

RGB to YCbCr Conversion. The final step in the image processing pipeline is to convert the processed sRGB image into YCbCr and therefore make it ready to be encoded by a JPEG encoder. Typically chrominance data is also subsampled after conversion and the data is stored into NV12 or other standard format.

3. IMAGE PROCESSING ON GPU

Graphics processing units were originally designed for drawing three-dimensional objects to the screen, the main purpose of the first GPUs being the graphics acceleration of 3D games. During recent years, GPUs have evolved into highly programmable chips which has enabled general-purpose computing on graphics processing units (GPGPU). One of the possible uses is image processing, which, because of its similarities to graphics rendering, appears to suit well for the GPU. This chapter provides a short introduction to the current programmable graphics pipelines and explains how they can be used for image processing purposes. The main focus of this chapter is in modern mobile GPUs, although the introduced concepts and methods are perfectly suitable for desktop GPUs too.

3.1 Motivation

Traditionally, image processing in mobile devices has been implemented in a separate image signal processor (ISP). The use of a dedicated hardware chip makes it possible to process images fast and with low relatively power consumption. However, since the processing is implemented in hardware, there is little room for customization. Typically, the available algorithms are hard-coded and can be only configured with pre-defined parameters. In addition, ISPs generally have really long development cycles which means that when an ISP is available for consumers, the provided algorithms are already few years old. Since image processing algorithms evolve continuously, the achieved image quality lags years behind the quality that could be achieved. Highly programmable ISPs exist, but adding a new chip to a device chip increases costs, so it would be tempting to implement image processing using components that exist in the device in any case.

An obvious alternative to the ISP is the central processing unit (CPU). Since the CPU is responsible for most of the general processing tasks in a mobile device, it needs to have a large amount of processing power. The trend in todays CPUs is to incorporate more and more cores instead of increasing the processing performance of a singe core. This suits well for image processing purposes since the image can be split into smaller parts that can be processed in parallel. Since the CPU is designed for general-purpose computing, there are no flexibility limitations for implementing new algorithms. The already available algorithms in a mobile device can be improved

and changed with an software update. Therefore, even a couple of years old device can be updated with the best available algorithms. However, the increased number of cores and processing performance typically leads into higher power consumption and bigger heat generation. Because there are millions of pixels to process in a single image, really intensive processing is needed. Since the CPU is used for most of the general tasks in the system, its excessive use can cause visible slowness in other processes as well. In addition, the increased heat generation can make the device overheat. The system typically resolves heating issues by limiting the processing performance of the CPU, which will not only make the image processing slower, but also has an effect on other processes running in the system.

The GPU generally has better processing performance and energy efficiency than the CPU which makes it a tempting choice for mobile image processing [12]. The GPU also acts as a co-processor, so it frees up the CPU to perform other tasks. And since the GPU is nowadays integrated into basically all mobile devices, there are no additional hardware costs for utilizing it. When the camera is running, the GPU is used very little which means that there is plenty of available processing power that can be utilized. Although the GPU is not as highly programmable as the CPU is, it has proven to have enough programmability for implementing advanced image processing algorithms [19][12].

3.2 Programmable Graphics Pipeline

The processing model of a modern GPU is described by an abstraction called graphics pipeline. The term *pipeline* is used because the transformations from geometrical models to pixels in screen require multiple stages that are performed in sequence. After one stage has finished processing, the output is passed to the next stage and so on. Finally, after the last step, the output is drawn on the screen. One of the advantages of such model is that data can be processed in small pieces simultaneously. When one stage has finished, it can immediately start processing the next piece of data. This kind of processing model enables very good parallelization which is one of the key things that makes modern graphics pipelines really powerful. [20, pp. 14-17]

In the past, graphics pipelines had a fixed set of stages that were controlled by parameters such as transformation matrices. Nowadays, these fixed function processing blocks have been replaced with small programs called shaders. Shaders allow the user to write really complex effects such as lightning and shadows. Programmability has also enabled totally new use cases for the graphics pipeline, such as image processing that will be discussed further in the next section. [20, p. 15]

The stages of a graphics pipeline are illustrated in figure 3.1. The pipeline is controlled through a graphics API. The purpose of a graphics API is to be the link

between CPU and GPU and to provide abstractions for concepts such as textures, vertices and buffers. Currently there are two widely used graphics APIs available: DirectX and OpenGL. Although some terms are different in these APIs and the shaders are programmed with a different language, the main concepts are the same. When a command is run using a graphics API, it is put on a queue waiting to be processed. This means that tasks done on a graphics API are generally asynchronous. In games, for example, the state, as seen on the CPU side, is typically few frames ahead of what is displayed on the screen. In addition to the input passed from the CPU to the GPU, the graphics API also provides ways to read data back [20, p. 17].

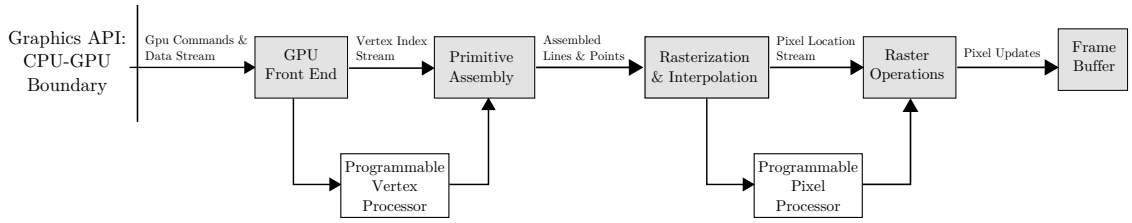


Figure 3.1: A conceptual image of the programmable graphics pipeline [12].

The geometric description of an object in the graphics pipeline is expressed as locations of vertices that form the object. When a vertex stream containing a scene arrives to the graphics pipeline, each vertex is sent to a programmable vertex processor. The purpose of a vertex processor is to transform the three-dimensional coordinate of a vertex into a two-dimensional coordinate at which the vertex appears at the screen. After vertex processing, transformed vertices arrive at the primitive assembly stage where primitives such as triangles, lines and points are formed [12]. Also, portions of the geometry that would fall outside of the screen are clipped.

In the rasterization stage, assembled triangles, lines and points are converted into pixels. Each pixel is then passed into a pixel processor which defines the color of the pixel [20, p. 18]. After the pixel processor stage, the output pixel goes through multiple operations before reaching a frame buffer. These operations include depth and pixel ownership checks that test whether the pixel should be visible or not [12]. Depth checks are needed because there might be multiple objects in the scene drawn at the same screen coordinate, but only the object closest to the camera should be drawn. Figure 3.2 shows how vertices are transformed during the pipeline.

3.2.1 Vertex Shader

To customize the function of the vertex processor, a small program called vertex shader is used. The vertex shader is run once per vertex and, in addition to the

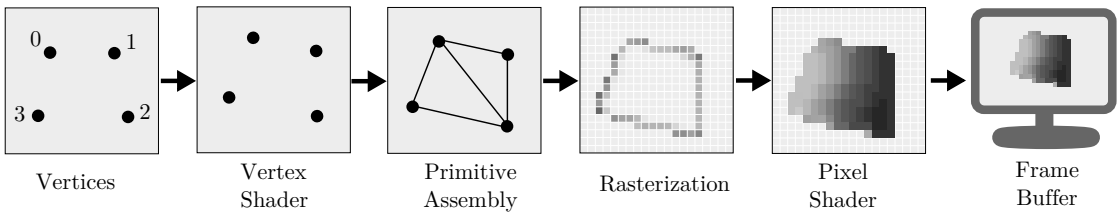


Figure 3.2: Vertices during different stages of the graphics pipeline.

position of the vertex, can also take user-defined variables as an input. These variables can be either *attribute variables* or *uniform variables*. Attribute variables are defined per vertex. They can contain texture coordinates, color data and normal information, for example. The vertex shader can pass attribute variables to the pixel processor stage where the actual coloring of the pixel is done. Uniform variables are constant for all vertices. Typically they contain different transformation and projection matrices that are used to transform the vertex from three-dimensional world coordinates to two-dimensional screen coordinates. Also, light properties such as direction and intensity, and material properties such as shininess are typically passed as uniform variables. [21, pp. 139-141]

The output of the vertex shader must contain at least the position of the vertex. Other typically used outputs include texture coordinates, vertex color and vertex normal information. The vertex shader can also pass any user-defined data that is needed in subsequent processing stages. For example, all variables that need to have a different value for each pixel in the pixel processor must be created in the vertex shader. After the vertex shader has been run for all vertices, the whole scene is located in a 2x2 cube with opposite corners at $(-1, -1, -1)$ and $(1, 1, 1)$. Any vertex located outside of these *normalized device coordinates* is either clipped based on the primitive object or removed completely. Although the transformation in vertex shader is made to two-dimensional screen coordinates, there is also the third depth component which is later used in the raster operations stage to check whether the pixel should be drawn or not. [21, p. 142] [20, p. 22]

3.2.2 Pixel Shader

A pixel shader, or fragment shader, is used to program the pixel processor. It is run for each pixel after rasterization. The purpose of a pixel shader is to define the color of the pixel. When a pixel shader is called, the output coordinate of the pixel is already defined which means the pixel shader cannot change the position of the pixel. In rasterization stage, the output variables of the vertex shader are interpolated for each pixel. Figure 3.3 shows an example of a pixel shader that

takes position and color as an input from the vertex shader and sets the output color of the pixel directly to the input color. Since the color value is interpolated automatically for each pixel, the result is a gradient containing the input colors at the corners. As seen in the example, pixel shaders define the color using four floating point components, red, green, blue and alpha. The range of color components is [0, 1], where 1 denotes the maximum intensity. Similarly, the input of a pixel shader could contain texture coordinates. A texture coordinate is defined by a pair of floating point numbers between 0 and 1 so that $(0, 0)$ is the top left corner and $(1, 1)$ is the bottom right corner of the texture. If vertices V_0, V_1 and V_2 in figure 3.3 contained texture coordinates $(0, 0)$, $(1, 0)$ and $(\frac{1}{2}, 1)$, respectively, the result would be a properly laid and scaled texture on top of the triangle.

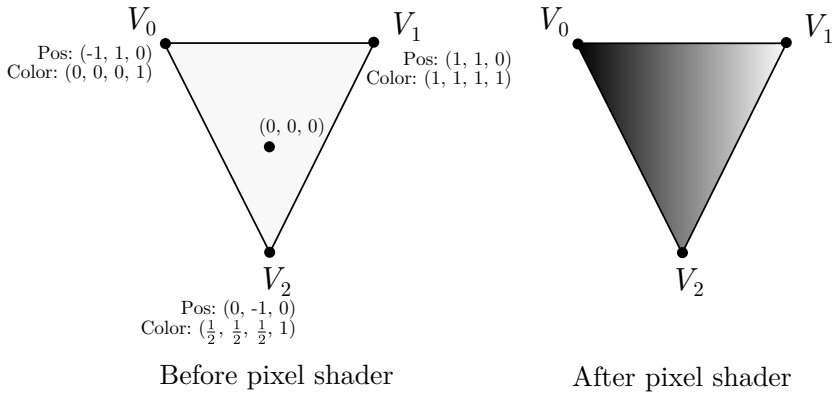


Figure 3.3: The output of a pixel shader that sets the color of the pixel directly to the color passed from the vertex shader.

In addition to input from the vertex shader, the pixel shader can also contain uniform variables just like the vertex shader. Variable types specifically designed for the pixel shader include textures and texture samplers. A texture contains a bitmap image that can be read and used when calculating the output color. To read a value from the texture, a sampler object is used. The sampler contains information about how the texture sampled, for example, whether bilinear interpolation should be used or what would be read if the texture is sampled outside of the texture coordinate range [0, 1]. [21, p. 159].

Typically pixel shaders have only one output variable, the four-component color. It is also possible to have multiple outputs, called render targets. The most common render target is the frame buffer, but it is also possible to declare a texture as a render target. This can be used in multipass rendering schemes. For example, a mirror could be implemented by first drawing the scene into a texture and then using that texture on top of the mirror surface. A texture can be also used as a render target if the output image needs to be read back from the GPU. [21, pp.

205-206]

3.3 Utilizing GPU for Image Processing

Although the programmable graphics pipeline was originally intended for rendering 3D scenes, there are some capabilities that make it possible to utilize the pipeline for image processing. These capabilities include reading a value from a texture into a variable, doing calculations based on the value and writing it back to another texture [21, pp. 239-240]. The ability to read the texture data back from the GPU instead of drawing into the back buffer makes it possible to save the result image or do further processing with it. A typical flow in GPU-based image processing is illustrated in figure 3.4. The graphics API is used to provide a scene description and a copy the input image into the GPU. The scene description is passed through a vertex shader, but the actual image processing takes place in a pixel shader. The input image is passed as a texture to the pixel shader which renders the output into another texture. The output texture is finally read back from the GPU and converted to a common image format.

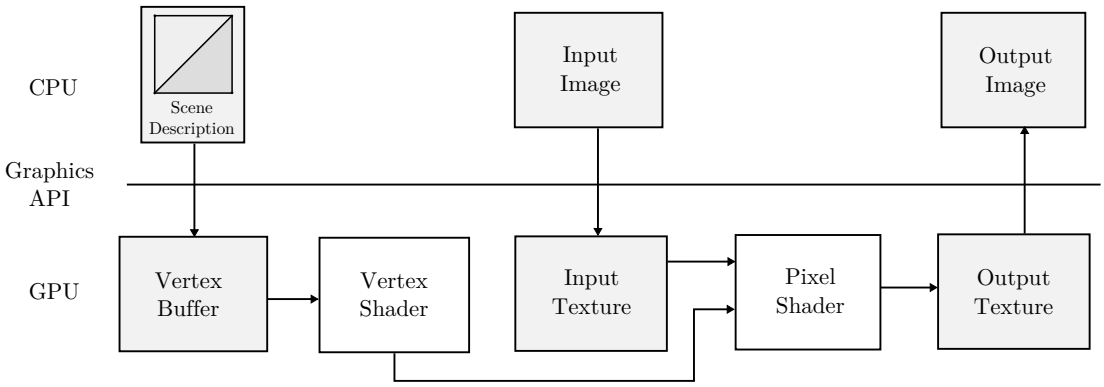


Figure 3.4: A typical processing flow in GPU-based image processing.

When a GPU is utilized for image processing, the scene description can simply contain coordinates and other attribute variables required for drawing a two-dimensional rectangle. Since triangles, points and lines typically are the only available primitives, the input rectangle must be split into two triangles as illustrated in figure 3.5. Each of the six vertices contain a scene position which is used by the GPU to define the position of each pixel in the pixel shader stage. In addition, each vertex contains a texture coordinate which can be used to read an exact pixel from the input texture. The vertex processing stage itself is typically quite simple among image processing tasks. For most algorithms, the vertex shader simply passes the scene description to the pixel shader.

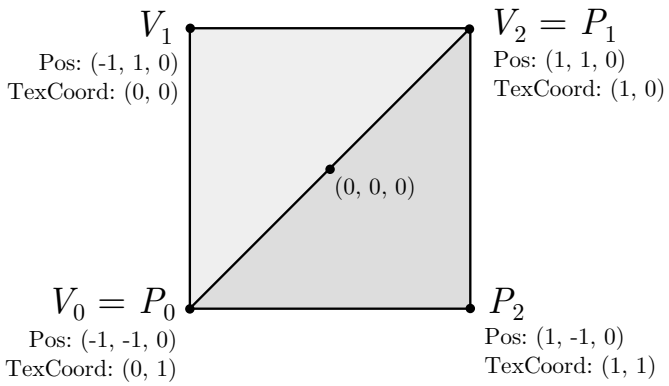


Figure 3.5: A scene description used for GPU-based image processing tasks. The first triangle is defined by vertices V_1, V_2 and V_3 and the second triangle by vertices P_1, P_2 and P_3.

To pass the input image into the GPU, it must be copied into a texture. There are multiple texture formats which can be used depending on the image format. There are generally three different component setups available in texture formats: one-component (R) textures, two-component (RG) textures, and four-component (RGBA) textures. Although terms R, G, B, and A are used to identify the channels, the actual data can be anything that fits the reserved space. Typical data types for color components include unsigned integers, floating point values and signed integers. Depending on precision requirements, the occupied space for each color component can be between 8 and 32 bits. Table 3.1 presents one way to store different image formats into a texture. Raw Bayer data can be stored into one four-component texture. Since all components are stored in one pixel, only one texture lookup is needed to process four pixels. Because both horizontal and vertical resolutions are half of the original image resolution, the number of pixel shader runs per image is also one quarter of the count required for processing a single-component, full resolution texture. Because of the available texture bit depths, the raw data must be padded to 16 or 8 bits. RGB and YCbCr data with 4:4:4 sampling can be stored similarly into a standard four-component texture. There is some overhead caused by the fourth alpha component which must be carried along since there are typically no three-component textures available. YCbCr data with 4:2:2 or 4:2:0 subsampling is problematic since chrominance information has lower resolution than lumimance information. One way to store such data is to use two textures: one single-component texture for luminance data and one smaller two-component texture for chrominance data.

Once the image has been copied into a texture, it can be bound as an input to the pixel shader. Since the texture coordinate was already passed as an attribute variable in the scene description, it is available as an input for the pixel shader.

Table 3.1: An example of texture formats for storing 800x600 image in different image formats.

Image format	Number of textures	Texture formats	Texture sizes
Raw Bayer	1	RGBA	400x300
RGB	1	RGBA	800x600
YCbCr (4:4:4)	1	RGBA	800x600
YCbCr (4:2:2)	2	R RG	800x600 400x600
YCbCr (4:2:0)	2	R RG	800x600 400x300

For single-pixel algorithms, this texture coordinate is the one used for sampling the texture for the input pixel at the current location. However, for kernel-based algorithms there must be a way to sample neighboring pixels. Since texture coordinates have range between 0 and 1, a simple increment to the coordinate does not work. Instead, the distance between two pixels in texture coordinates can be passed as a uniform variable to the pixel shader. To make the neighboring lookups as simple as possible, two uniform vector variables, **dx** and **dy** can be declared so that

$$\mathbf{dx} = \begin{bmatrix} 1/w \\ 0 \end{bmatrix} \quad \mathbf{dy} = \begin{bmatrix} 0 \\ 1/h \end{bmatrix}, \tag{3.1}$$

where w is the width and h is the height of the texture. These variables can be then used for jumping one pixel in texture coordinates to either direction. Listing 3.1 shows an example of of using

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{3.2}$$

as a processing kernel and **dx** and **dy** as uniform variables to implement a simple edge detection in High Level Shading Language.

As seen in listing 3.1, there is no special handling implemented for border pixels. Fortunately, sampling a texture with coordinates outside of the range [0, 1] is common in 3D rendering which is why graphics APIs have designed ways to handle the situation. DirectX, for example, defines following texture addressing modes: wrap, mirror, clamp, border and mirror once [22, p. 129]. The selected mode is defined when the sampler object is created. From image processing perspective, clamp, border and mirror modes match first three modes illustrated in figure 2.8, respectively.

```
Texture2D Image : register(t0);
SamplerState Sampler : register(s0);

struct PixelShaderInput {
  float4 pos : SV_POSITION;
  float2 texCoord : TEXCOORD0;
};

cbuffer constantBuffer : register(b0) {
  float2 dx;
  float2 dy;
};

float4 main(PixelShaderInput input) : SV_TARGET {
  float4 color =
      1 * Image.Sample(Sampler, input.texCoord - dy)
    + 1 * Image.Sample(Sampler, input.texCoord - dx)
    - 4 * Image.Sample(Sampler, input.texCoord)
    + 1 * Image.Sample(Sampler, input.texCoord + dx)
    + 1 * Image.Sample(Sampler, input.texCoord + dy);
   return color;
}
```

Listing 3.1: A simple edge detection implemented using High Level Shading Language.

However, mirroring on border pixel, which is required for raw Bayer data to keep the Bayer order intact, must be implemented manually in the pixel shader.

Using lookup tables on the GPU is as simple as creating an addition one-dimensional texture and using the input color to sample a value from it. Since colors have the same [0, 1] range as texture coordinates, no extra conversions are needed. With lookup tables it is also possible utilize the powerful bilinear interpolation that GPUs support for texture lookups. For example, with 16-bit data the required lookup table size would be 65536 items. However, for most cases a smaller table, such as 2048 items provides enough precision as far as there is a method for querying the missing values. If the lookup table sampler is declared to use bilinear filtering, the missing information is automatically interpolated when the lookup table is sampled. [23, p. 385]

As shown above, the GPU can be utilized for spatial image processing quite easily. However, there are certain image processing tasks that are difficult, too time consuming or even impossible to be efficiently implemented with the methods above. Algorithms that take an image in and process some kind of statistics out are generally one of the most difficult ones. One example is creating a histogram, i.e., calculating how many pixels in an image have different color values. The problem with implementing histogram calculation in a pixel shader is the fact that the output must be written to a predefined location in the output buffer. In general, pixel

shaders can flexibly gather information from different places at different sources, but cannot easily scatter the output into different locations, based on input values. [23, p. 394]

To allow even more general computing on GPUs, multiple platforms and frameworks have been developed. Compute Unified Device Architecture (CUDA) allows general purpose computing on NVIDIA GPUs [24]. Open Computing Language (OpenCL) provides similar functionality for GPUs from other manufacturers [25]. DirectX also have an equivalent functionality called DirectCompute which, unlike the others, is incorporated into the existing graphics API as a separate shader type [26]. All of these GPGPU APIs enable dynamic scattering of the output. They also add more control on defining the contents of input and output buffers without having to use predefined texture types only.

3.4 Special Considerations for Mobile Devices

Mobile GPUs have multiple limitations and features that need to be taken into account when implementing image processing algorithms on mobile devices. Although the processing performance of mobile GPUs is increasing continuously, there are significant differences in the architecture and design compared to their desktop counterparts [27]. The differences are mostly caused by the fact that mobile GPUs were designed for low power consumption. An easy way to reduce power consumption is to limit the number of cores and lower the memory bandwidth which obviously have a negative impact on performance [28].

Desktop GPUs typically have several gigabytes of video memory reserved for storing textures, shader code and other buffers. However, in mobile devices the memory is shared between CPU and GPU. In addition to the shared memory, also the memory bandwidth is shared. Since the graphics pipeline is mainly designed for drawing pixels on the screen, the bandwidth available for reading data back from the GPU may also be limited. This bandwidth bottleneck can be particularly visible in GPU-based image processing since filtering operations require multiple texture lookups per pixel. Although there is plenty of pure computing power available, the processing algorithm may not be able to utilize it efficiently since the processing is constrained by the memory bandwidth. Most high-end mobile GPUs have implemented on-chip caches to reduce memory transactions during rendering [28], but from image processing perspective these caches may be too small to significantly increase the performance because input and output textures are too big to fit entirely on caches. One way to reduce the number of memory transactions is to store data compressed into memory and decompress it on-the-fly in the GPU. [29][27]

Another thing that limits the potential of utilizing mobile GPUs for image processing is the absence of GPGPU APIs such as OpenCL and DirectCompute. Al-

though they are likely to be included in mobile GPUs in the future, currently image processing algorithms must be implemented with vertex and pixel shaders [12]. However, as covered in the previous sections, these standard shaders alone are suitable for implementing most spatial domain processing algorithms.

4. FRAMEWORK FOR GPU-BASED IMAGE PROCESSING

Graphics APIs are designed to give full control to the GPU which is why their abstraction level is fairly low. Implementing an image processing pipeline on top of built-in APIs would be time-consuming and tasks such as changing the order of the algorithms or adding new algorithms would be difficult. Also, in image processing, most of the available features are not used or are used very little which would cause plenty of overhead in implementation. This chapter introduces a DirectX-based framework designed solely on image processing use. The framework is designed so that it allows the user to dynamically construct the processing pipeline from separate processing blocks. It also provides a higher abstraction level for implementing algorithms, yet keeping the low-level APIs available for advanced algorithms that cannot be built with the APIs provided by the framework alone.

4.1 Direct3D and High Level Shading Language

Direct3D is the 3D-rendering API in the DirectX API family. It is designed by Microsoft for use in Windows-based products. Direct3D is implemented in a way that new versions of the API are backwards compatible with previous-generation GPUs. The available features are defined by a feature level. Desktop GPUs typically support the newest available feature level when they are released, but with mobile GPUs it can take years before a feature level becomes available. For example, in Windows Phone 8, the supported feature level is 9.3 although the Direct3D API is already in version 11.2. The framework introduced in this chapter will be built with Windows Phone usage in mind, so it will be based on the features available in the feature level 9.3.

An overview of the Direct3D 11 pipeline is illustrated in figure 4.1. Unlike the programmable graphics pipeline described in section 3.2 (p. 16), Direct3D runs the input assembly stage before vertex shader, although this does not have any effect on the shader implementation itself. After input assembly, the vertex shader is run as usual. In Direct3D 9 the next stages are rasterization, pixel shader and output merger which corresponds to the raster operations stage in figure 3.1 (p. 17). In Direct3D 10 and 11, however, few extra stages can be enabled in the pipeline. Direct3D 11 introduces a feature called tessellation which can be used to adjust the

details of a model dynamically. For example, when the camera is located close to an object, the object could have more vertices than when the camera is far away [21, p. 316]. After tessellation, the next stage in the pipeline is a geometry shader stage which was introduced in Direct3D 10. A geometry shader can be used to add new primitives such as points, lines and triangles to the scene. For example, a Bézier curve could be implemented by providing the control points as an input to the geometry shader and calculating actual the points of the curve dynamically in the shader [21, pp. 301-302].

Another new feature added in Direct3D 11 is the compute shader. It is implemented separately from the main rendering pipeline which is why it is not illustrated in figure 4.1. The compute shader is part of the DirectCompute API which adds better support for GPGPU in the DirectX API family. Compute shaders allow general-purpose calculations that are impossible or difficult to implement using the standard rendering pipeline, which is what makes it especially interesting from image processing point of view. The framework introduced in this chapter does not support DirectCompute because of the feature level limitations. However, compute shaders support the same texture formats that are used in our implementation, so adding the support will be pretty straightforward in the future. [30, p. 287-290]

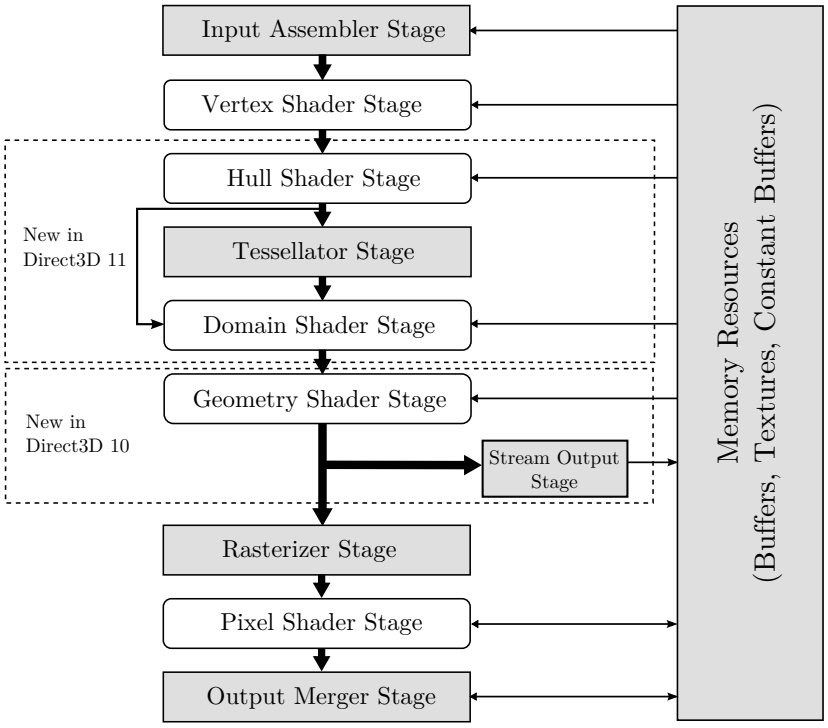


Figure 4.1: The Direct3D 11 pipeline and new features added to the API in recent versions [31, p. 4].

All shader types in Direct3D are programmed using a programming language called High Level Shading Language or High Level Shader Language (HLSL). In addition to basic arithmetic operations, HLSL provides a set of predefined intrinsic functions for operations such as rounding and calculating trigonometric values. HLSL is compiled into an optimized byte code which can be passed to the GPU.

There are some features in Direct3D that need to be taken into account when implementing an image processing pipeline. Direct3D separates textures into default, dynamic, immutable and staging textures depending on their access rights, as shown in table 4.1. Since there is no texture type which allows read and write operations on both CPU and GPU, the pipeline must be built from at least three textures. A dynamic texture is used as an input to the pipeline since it provides write support for the CPU and read support for the GPU. In processing algorithms, default textures must be used to be able to write into a texture and read it in the next algorithm. Finally, to copy the data back from the GPU, it must be first copied into a staging texture since default textures do not support read by the CPU. There is a separate call in the Direct3D that can be used to make that copy. During vertex shading, one more buffer copy of the data is made. Therefore, the total amount of required buffer copies when processing a single image through one algorithm is four as illustrated in figure 4.2. The bandwidth bottleneck is already a problem in basic rendering, but the large amount of buffer copies emphasizes it even further. Therefore, it is important to be able to do as much processing on the GPU at once before reading the data back. An implementation where one algorithm would require CPU processing in the middle of the pipeline would have a serious impact on performance.

Table 4.1: Available texture types and their access rights in Direct3D. The GPU write and read access to staging textures is provided by separate commands. Staging textures cannot be used as render targets in a pixel shader. [32]

Resource usage	**Default**	**Dynamic**	**Immutable**	**Staging**
GPU read	yes	yes	yes	(yes)
GPU write	yes			(yes)
CPU read				yes
CPU write		yes		yes

Several hardware limitations are another thing that need to be taken into consideration when implementing an image processing framework on top of Direct3D. One significant limitation is the maximum texture size which is 4096x4096 for feature level 9.3 [22]. In modern high-end mobile cameras, still image resolution is typically

bigger than that. For example, 5376x3024 pixels is a typical size for an image from a 20MP sensor in 16:9 aspect ratio. One way to avoid this limitation is to split the image into smaller segments called *tiles*. Another limitation is related to the available texture formats. Although Direct3D 11 API supports all possible component and bit depth combinations, the feature level 9.3 limits the availability of those formats. For example, there are only few texture formats that can be bound as a render target.

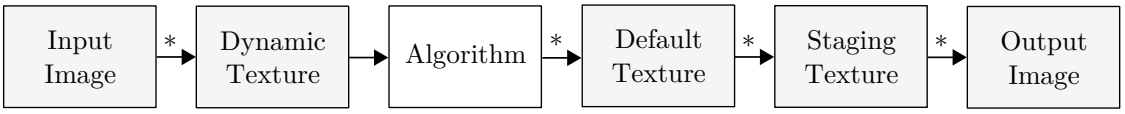


Figure 4.2: Required textures and other buffers when processing a single image through one algorithm using Direct3D. A buffer copy is created in stages marked with (*).

4.2 Architecture

The GPU processing framework is designed so that the pipeline can be built and configured without having to touch the framework code itself. A high level architecture of the framework and the general processing flow is illustrated in figure 4.3. The illustrated components can be separated into framework provided components and user provided components.

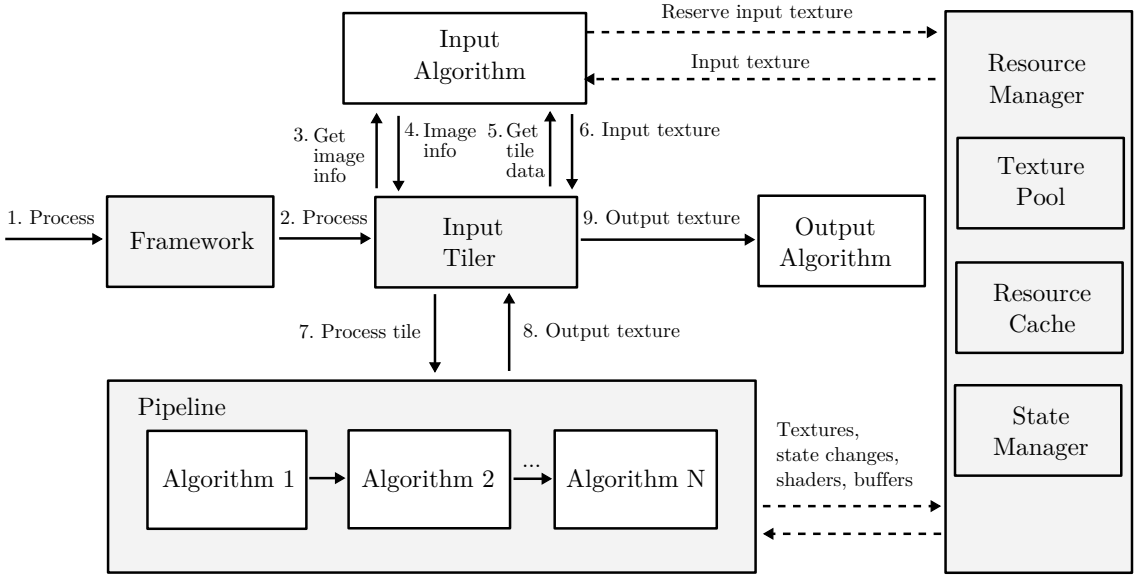


Figure 4.3: A high level structure and the processing flow of the framework.

User provided components are registered to the pipeline using a framework object

which is the only object that is visible to the framework user. In addition to registering algorithms, it contains a method for initiating the processing. There are three types of user provided components required. The actual processing algorithms are implemented as pipeline algorithms. There must be at least one pipeline algorithm, but the upper limit is not defined. The framework builds a pipeline automatically based on the order the pipeline algorithms are registered. In addition, there must be exactly one input and one output algorithm defined. The purpose of an input algorithm is to provide the initial data to the framework in a GPU-compatible format, i.e., as a texture. Similarly, the output algorithm is responsible for handling the output of the last pipeline algorithm. The operation of the framework can be divided into nine stages that are also illustrated in figure 4.3:

Stages 1 and 2. A process call is initialized on the framework object. The call is forwarded to the *Input Tiler* component.

Stages 3 and 4. *Input Tiler* asks *Input Algorithm* for image information such as size and format. Based on the information and framework configuration, *Input Tiler* calculates tiling parameters. These parameters include the required number of processing tiles and the required overlap between tiles. This functionality is described more closely in section 4.4.

Stages 5 and 6. *Input Tiler* asks for input data for single tile. *Input Algorithm* returns the requested tile data as a texture.

Stages 7 and 8. The input data is passed to the *Pipeline* component. *Pipeline* runs the data through each algorithm and returns the output of the last algorithm as a texture.

Stage 9. The output texture of a single tile is passed to *Output Algorithm*. *Output Algorithm* can copy the tile into a full output buffer, for example.

After the first tile has been processed, the processing will return back to stage 5, where the next tile will be processed. This continues until there are no more tiles to process. Each component in the pipeline has a reference to a component called *Resource Manager* which allows the framework and algorithms to reuse buffers, textures and other resources and minimize state changes on the GPU side.

4.3 Algorithm Interfaces

The use of input and output algorithms allow the user to define how data is stored in the GPU and how it is read back. Therefore, the framework itself does not have to decide which texture format to use for each input data format. In addition, the use of an input algorithm allows multiple pipelines to be bound together. For example, it would be possible to divide the pipeline into separate pre-processing and post-processing pipelines and maintain them separately. In such case the output algorithm of the first pre-processing pipeline could do some analysis based on the

output and then pass the output to the input algorithm of the post-processing pipeline.

The use of an input algorithm also makes it possible to implement scenarios where the input is a stream instead of a single image. For example, there could be an input algorithm which reads input from a movie file and passes each frame to the framework where some processing is done for each frame. The output algorithm in such case could pass the data into a video encoder which could re-encode the movie into another file. To implement one of the algorithm types, the user must implement an interface provided by the framework. These interfaces are given in figure 4.4.

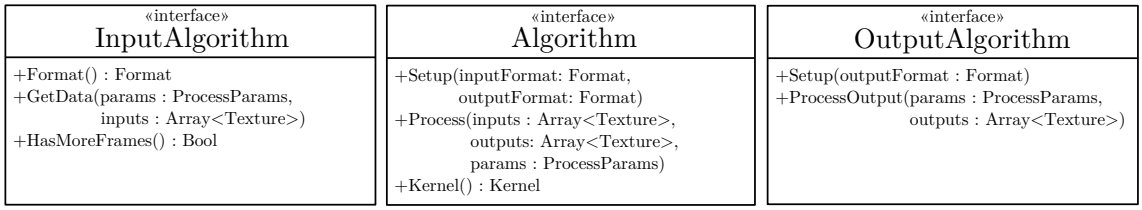


Figure 4.4: Algorithm interfaces provided by the framework.

InputAlgorithm interface contains methods for querying the input format and the actual image data. To support multi-frame scenarios, there is also a method for querying whether there are more frames available for processing.

Pipeline algorithms implement the *Algorithm* interface. There is a one-time setup method which can be used to do one-time initializations based on image size, for example. The process method takes output textures from previous algorithm as input. The output textures are also pre-allocated by the framework and passed as an argument to the process method. In addition, each algorithm has a method for querying the required processing kernel. The use of this information is described more closely in section 4.4.

OutputAlgorithm interface contains a similar Setup method as *Algorithm* interface. It also has a method for processing the output of the last texture. The output can contain multiple textures that are passed as an array to the method.

4.4 Tiled Processing

The main purpose of dividing the image into tiles is to bypass the hardware texture size limitation. When smaller tiles are used, also memory consumption is decreased because same textures can be reused for each tile. Also, the use of smaller textures can lead into better performance since internal caches on the GPU are able to accommodate a larger chunk of the input texture at once which decreases the amount of lookups to the system memory.

Since filtering algorithms require information about surrounding pixels, there

must be a way to handle the borders between tiles. The border processing methods described in figure 2.8 are not suitable for processing the borders of the tile since they would cause artifacts to the border areas. Instead, the division to tiles can be done so that there is a small amount of overlapping pixels on the tile border. After the tile has been processed, the unneeded overlapping pixels are discarded. To maximize the reuse of textures, the image is divided into equally sized tiles as illustrated in figure 4.5. Overlapping pixels o_x and o_y are added to one side of the tile only, depending on the position of the tile. Although there is some extra computation required when reading the data back from correct coordinates, this makes it possible utilize the built in texture mirroring and clamping features provided by Direct3D on the actual image borders.

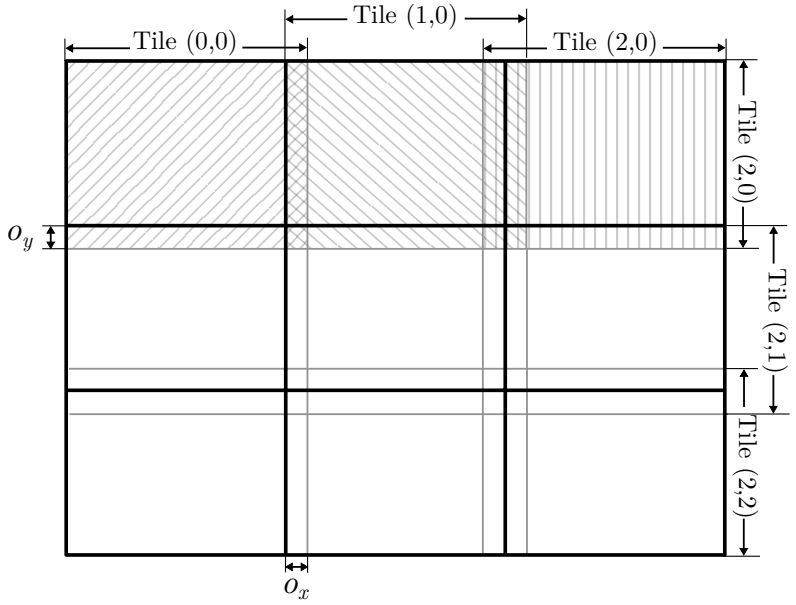


Figure 4.5: An example of dividing an image into three tiles horizontally and vertically.

In our framework implementation, each algorithm has a method for querying its processing kernel. The kernel is described by parameters k_w, k_h, k_x and k_y, where k_w is the width of the kernel and k_h is the height of the kernel. Parameters k_x and k_y are the x and y coordinates for the processed pixel in the kernel, (0, 0) being the top left coordinate of the kernel. The amount of overlapping pixels required for algorithm i can be calculated from

$$o_{x,i} = 2 * max(k_{w,i} - 1 - k_{x,i}, k_{x,i}) \tag{4.1}$$

$$o_{y,i} = 2 * max(k_{h,i} - 1 - k_{y,i}, k_{y,i}). \tag{4.2}$$

Since the amount of overlapping pixels must be the same for all textures to maximize texture reuse, the selected value is defined by the maximum kernel size in

the processing pipeline. When the output texture is read back from the GPU, half of the overlapping pixels are skipped from both textures. For example, when using a 5x5 kernel so that the processed pixel is located at the center, the value for both o_x and o_y is 4. Figure 4.6 illustrates how the pixels are read from the output textures in such case. To simplify the reading of correct pixels in the output algorithm, *Input Tiler* passes pre-calculated process parameters containing texture coordinates to the output algorithm.

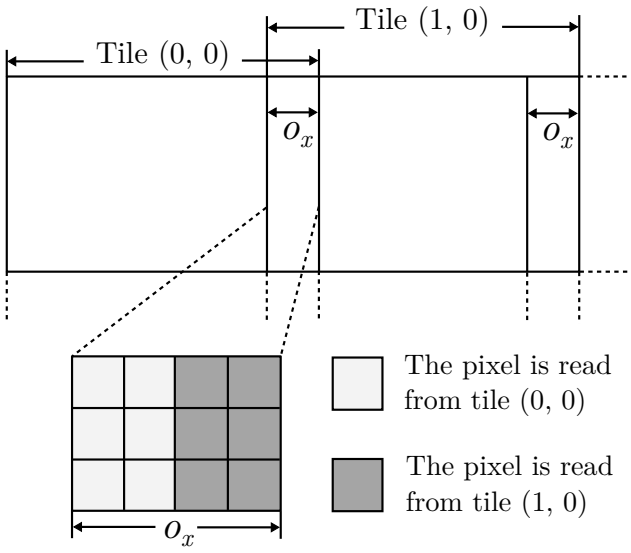


Figure 4.6: Processing output pixels when a 5x5 kernel is used.

4.5 Resource Management

The graphics pipeline is controlled by switching shaders, changing active buffers, and altering the current ones. These state changes should be generally avoided whenever possible, since they disrupt the pipeline throughput [20, p. 472]. When a state change is initiated, a new command is added to the command queue of the GPU. Each item in the queue causes extra overhead to the GPU when the queue is flushed. Also, buffers and shader objects consume memory, so reuse is required to keep the memory consumption low. When the amount of GPU resources is low, also the possibility that the resource is already available in GPU caches when the resource is needed increases. Reusing objects also decreases the framework initialization time since there is less data that needs to be transferred and possible converted to a GPU-compatible format. For example, the creation of shader objects takes a while because the GPU has to compile the shader byte code into its internal format. To maximize the reuse of buffers and to avoid unnecessary state changes, the framework implements a separate resource manager component. The resource manager object is

passed to all other components and algorithms, so that it can be used anytime. The resource manager contains separate subcomponents for managing textures, other resources and state changes.

4.5.1 Texture Pool

Since the pipeline is implemented by processing an image from a texture to another, some kind of texture handling must be implemented on the framework side. When a new input or output texture is needed, the framework queries the texture pool component for a texture buffer. If an available texture is found, it is returned, otherwise a new texture is created. When the texture is no longer needed, the framework can return it to the pool. In case an algorithm needs intermediate buffers, for example, the texture pool is visible to them through the resource manager.

Texture handling inside the pipeline is implemented using a scheme called ping-pong buffering. In ping-pong buffering, the output of an algorithm is passed directly as an input to the next algorithm. The next algorithm can then reuse the input of the first algorithm as an output buffer. On the GPU the same buffer can be reused as long as it has a compatible format, i.e. the same texture format and size. Figure 4.7 shows an example of ping-pong buffering with three algorithms. Algorithms 1 and 2 have the same input and output format, so textures 1 and 2 can be reused. Algorithm 3, however, has a different output format which is why an additional output texture is required. The same processing sequence is illustrated as a sequence diagram in figure 4.8.

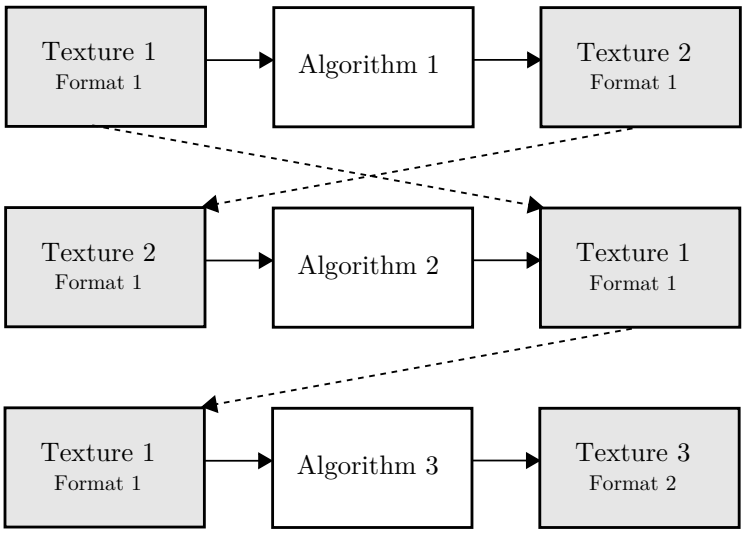


Figure 4.7: The ping-pong buffering scheme in GPU-based image processing.

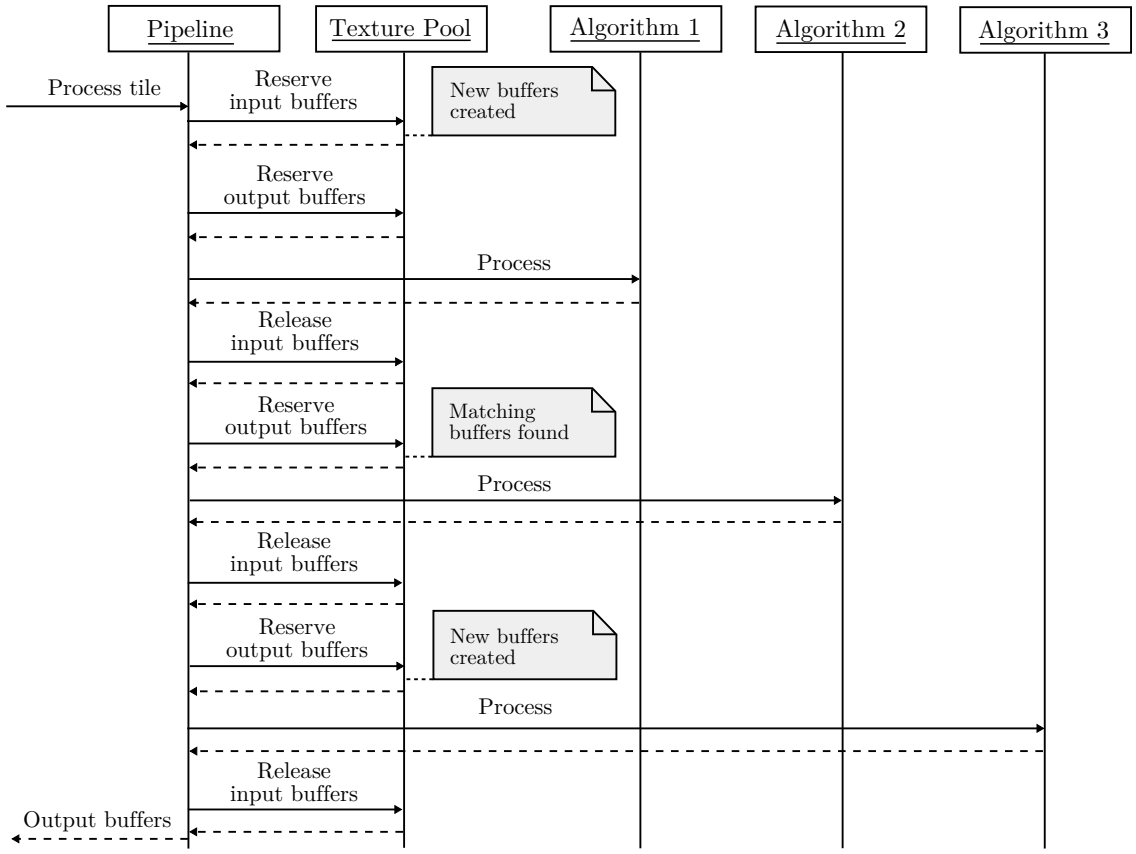


Figure 4.8: The processing flow inside the pipeline component.

4.5.2 Resource Cache

In addition to textures, also buffers and shaders can be reused. The scene is described by vertex and index buffers, which, for most algorithms, contain the exactly same description as illustrated in figure 3.5. An easy way would be using the same buffer for all algorithms, but that would break the generic nature of the framework. In some cases, an algorithm might need to use a different vertex buffer which is why a more sophisticated method for handling buffers must be implemented. The resource cache keeps reference to all created vertex and index buffers. When a new buffer is created, the resource cache is called with parameters containing information such as buffer size and formats of the individual items inside the buffer. If a buffer with same parameters already exists, it is returned, otherwise a new buffer is created. If an algorithm needs a buffer that contains dynamically changing data, it can create the buffer object without using the cache.

The resource cache also has a store for pixel and vertex shaders. Typically shaders are compiled into header files. The compiled header file contains a field in a universal byte code format, which is supported by both desktop and mobile GPUs. Since the byte code is available in a header as a byte array, the pointer to the byte array can

be passed to the resource cache when the shader object is created. If the resource cache finds a previous shader object created for the same byte buffer pointer, it can return that object.

4.5.3 State Manager

While texture pool and resource cache are able to maximize the reuse of buffers and other objects, they do not prevent algorithms or other components from changing active buffers and shaders unnecessarily. For example, when an algorithm sets the active vertex shader, it cannot know if the same shader is already active on the GPU. The graphics API can be queried for the current shader, but that adds some unwanted stress to the bus between the CPU and the GPU. In addition, it adds extra logic for each case when the state of the GPU is changed. A simple solution for reducing state changes is to keep a copy of the GPU state in the CPU memory, which is the purpose of the state manager component. Whenever an algorithm or another framework component wants to change the state of the GPU, it calls the state manager to do that. Since the state manager knows what the currently active shader or buffer is, it can either update the actual GPU state or skip the call in case the passed object is already active.

The actual cost of a state change depends on hardware platform, driver and the type of state variable being modified [20, p. 472]. In modern GPUs there is most likely already logic that handles the cases when the same vertex shader, for example, is set active multiple times. But since a generic processing framework may run in multiple different platforms, it makes sense to implement the state manager as a separate component. The required overhead on the CPU side to keep a reference to the active buffers and shaders is insignificant compared to the memory and processing requirements of other components. Since buffers and shaders are already cached and reused, the state manager can diminish a large amount of calls to the graphics API. It also keeps the command queue on the GPU as small as possible.

5. OPTIMIZATION FOR MOBILE GPUS

Although there is more and more raw computing power available in mobile GPUs, the limited memory bandwidth can have a significant impact on achieved performance. In image processing, memory bandwidth requirements are very high because the image data is passed through the graphics pipeline as textures. There are generally three types of methods for reducing the memory transfer between GPU and system memory: reducing the amount of transferred data per pixel, reducing the amount of memory lookups and optimizing the size of buffers so that the requested data is more likely available in caches when needed. This chapter provides some methods for reducing latencies caused by the limited memory bandwidth especially in mobile GPUs.

5.1 Test Hardware and Setup

The specifications of a test hardware used in optimization and measurements are listed in table 5.1.

Table 5.1: The test hardware and image formats used in optimizations and measurements.

SoC	Qualcomm Snapdragon 800
CPU	4x Krait 400 (Up to 2.3 GHz per core)
GPU	Adreno 330 (450 MHz)
Input image	Raw Bayer (5376x3024, 10 bits per pixel)
Output image	YCbCr 4:4:4 (5376x3024, 24 bits per pixel)

The test pipeline contains seven algorithms as illustrated in figure 2.9. Each algorithm is run in a separate pixel shader. The input is a 5376x3024 raw Bayer image captured by a 20 megapixel sensor. The input is passed as 16-bit fixed point data to the GPU texture and read back as 8-bit to reduce bandwidth costs. The input image contains a typical studio scene used in camera measurements and the output of the pipeline is an YCbCr 4:4:4 image which is illustrated in figure 5.1.

5.2 Fixed-Point versus Floating-Point

Most variables, such as colors, texture coordinates and scene coordinates, are passed to shaders as floating point numbers, which means that majority of the actual cal-

Figure 5.1: The scene used as an input image in measurements.

culations are done using floating point arithmetics. For this reason, arithmetic units inside a GPU are designed to be really efficient on floating point calculations. When algorithms are implemented on the CPU, the best performance is typically achieved with fixed-point arithmetics. However, on the GPU there is an extra conversion required when a single point value read from the texture is passed to a shader. This can be avoided by using floating point textures.

Both 32-bit and 64-bit floating point sizes are generally used in applications outside of the GPU scope, but GPUs also have support for half-precision, 16-bit floating point. The 16-bit version provides an interesting option from image processing perspective since it requires only half of the space and bandwidth compared the regular 32-bit floating point. However, the limited precision of 16-bit values can have a visible impact on the output image. To measure the impact, some kind of metrics can be used to calculate the difference between output images. In this case, the difference is caused only by limited precision, so a simple metric such as the Euclidean distance can be used. Let $\mathbf{x}$ and $\mathbf{y}$ be two w by h images, $\mathbf{x} = (x_1, x_2, x_3, ..., x_{wh})$, $\mathbf{y} = (y_1, y_2, y_3, ..., y_{wh})$, where x_{lw+k} and y_{lw+k} are the pixel values at location (k, l). The normalized Euclidean distance between $\mathbf{x}$ and $\mathbf{y}$ can be calculated from [33]

$$d^2_{E,norm}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{k=1}^{wh}(x_k - y_k)^2}{wh}. \tag{5.1}$$

The calculated Euclidean distance describes the square of an average difference between pixels at the same position, bigger value denoting bigger difference.

The test pipeline was run using 32-bit and 16-bit floating point textures and 16-bit and 8-bit fixed point textures. The input for the pipeline was provided as 16-bit fixed point values, but after the first algorithm, the processing was done using the specified format. Finally, the data was converted to 8 bits before reading back from the GPU. The comparison was done based on these converted 8-bit values and 32-bit floating point texture format was used as a reference because it provides the best precision. Table 5.2 summarizes measured processing times and normalized Euclidean distances for the test image.

Table 5.2: Comparison of different texture formats. 32-bit floating-point is used as a reference.

Texture format	**Bit depth**	**Processing time**	**Euclidean distance**
Floating point	32	100.0%	0.0000
Floating point	16	59.1%	0.1602
Fixed point	16	145.9%	0.0066
Fixed point	8	47.5%	2.2415

As expected, the processing time almost halves when the bit depth is decreased from 32 to 16 in floating point format. With fixed point, however, the processing performance gains are even higher when moving from 16 bits to 8 bits. This could be caused by optimizations that are done on hardware level for 8-bit data. The normalized Euclidean Distances indicate that the difference between 32-bit floating point and 16-bit fixed point is insignificant. A larger difference in the image can be seen when moving to 8 bits. Storing the intermediate buffers in only 8 bits is clearly not enough to provide an adequate image quality, although it has the fastest processing time.

Based on the measurements, the best option for a texture format appears to be 16-bit floating point. It provides better processing performance than 32-bit floating point, yet giving an adequate image quality. Another good aspect of using floating point textures is that the best precision is available in low values where the human eye is most sensitive to differences. The difference in the measured Euclidean distance is most likely caused by rounding errors that happen on the upper, more inaccurate values where differences are not that significant for the human eye.

5.3 Texture Size

GPU manufactures typically advice to use as small textures as possible to gain the best performance. The smaller the used textures are, the bigger the probability that the texture is already in internal caches when needed is. In image processing, smaller

textures can be used by dividing the image into smaller tiles. However, since there must be a certain amount of overlapping pixels between in each file, the decrease in tile size increases the total amount of processed pixels and thus adds some overhead. In addition, smaller tiles lead into more runs for the graphics pipeline and more state changes in the GPU which can have a significant impact on processing times.

The number of pixels to process in the whole image including overlapping pixels is given by

$$N = ho_x n_x + wo_y n_y + o_x o_y n_x n_y + wh, \tag{5.2}$$

where h is the height of the image, o_x is the number overlapping pixels in horizontal direction, $n_x > 1$ is the number of tiles in horizontal direction, w is the width of the image, o_y is the number of overlapping pixels in vertical direction and $n_y > 1$ is the number of tiles in vertical direction. To find the optimal texture size, the pipeline was run by dividing the input image into different amount of tiles in horizontal and vertical direction. In this case $w = 5376$, $h = 3024$ and $o_x = o_y = 6$, so the total number of pixels is

$$N = 18144n_x + 32256n_y + 36n_x n_y + 16257024. \tag{5.3}$$

Measured processing times and calculated values for N are illustrated in figure 5.2. The total number of pixels is scaled so that the 100% line matches the minimum value of N. The graph clearly indicates that the increase in the total number of pixels is not significant compared to the total pixel count. To simplify the graph, the texture dimensions are described by a single value d_t which is given by

$$d_t = \sqrt{\frac{w}{n_x}\frac{h}{n_y}}. \tag{5.4}$$

This represents the width and height of a square texture containing the same amount of pixels as the texture in question.

As expected, the processing time decreases when the texture size increases. The fastest processing time was achieved with the biggest texture size, i.e when $n_x = 2$ and $n_y = 1$. However, there is no significant decrease in processing times when the texture dimension is decreased. This is most likely caused by improved cache hit rate inside the GPU. A more significant increase in the processing time can be seen when the texture dimension is decreased below 256. The most optimal texture size according to these measurements in terms of memory usage and processing time appears to be 390x384. The corresponding values for n_x and n_y are 14 and 8, respectively. The measured processing time with this configuration was 113% of the best achievable processing time which is still tolerable. The memory consumption,

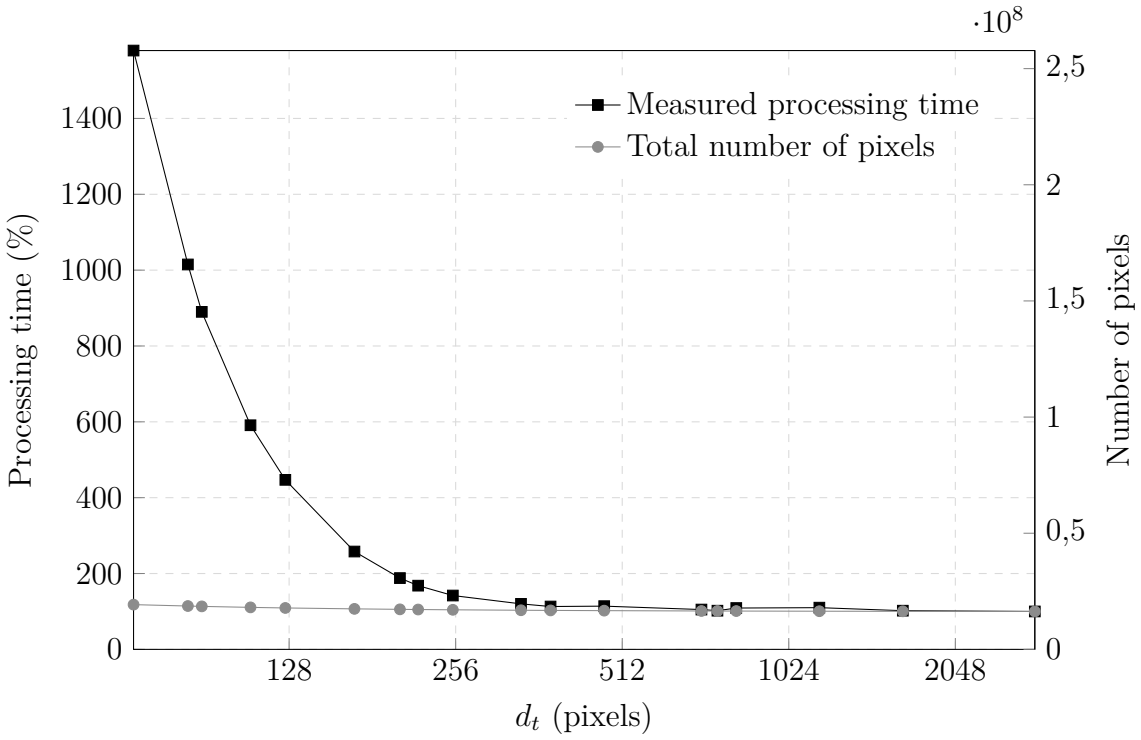


Figure 5.2: The processing time of the pipeline with different texture sizes.

however, is only 1.84% of the memory required by the biggest possible texture which in this case is 2694x3024.

5.4 Concurrent Processing and Texture Transfers

So far, processing times have been measured so that the processing time of each tile has been calculated separately and then summed together. However, processing tiles individually can have a severe impact on performance, because the GPU must synchronize with the CPU each time the data is read back. It takes a while to transfer the data back from the GPU which denotes that the GPU is idle for a relatively long period of time. One way to solve the issue is to provide as much commands and data to the GPU at once as possible.

In our framework, providing more input to the GPU at once is fairly simple. The input algorithm can be queried for multiple input tiles at once before they are passed to the pipeline. Similarly, the output textures can be buffered into an array before they are read back. The execution timelines of concurrent and non-concurrent cases are illustrated in figure 5.3. The achieved performance gain is related to the amount of input textures provided to the GPU at once, but the downside of doing multiple texture uploads and downloads at once is the fact that more input and output textures are needed, resulting into a higher memory usage. Since the pipeline is

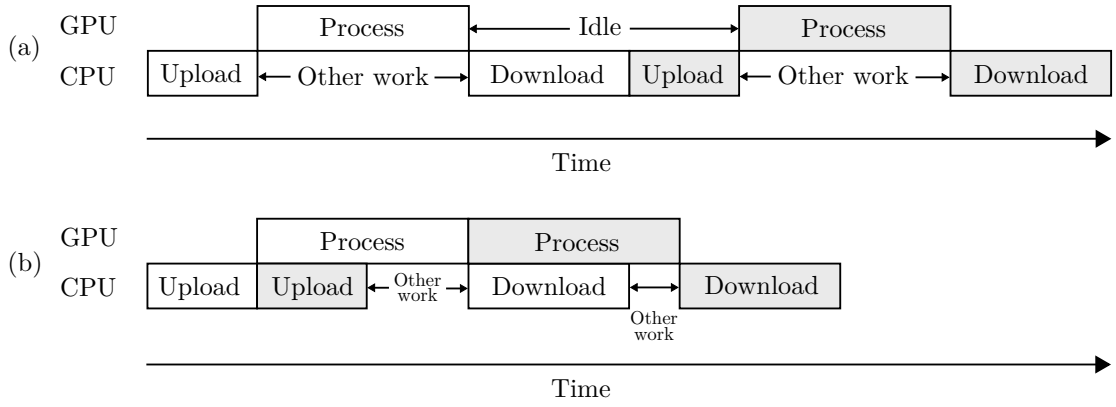


Figure 5.3: Execution timelines in cases (a) when there is no concurrency between texture transfers and processing and (b) when processing is done in parallel with texture transfers.

constructed from multiple stages, the input texture is needed only on the first stage. Therefore, once the first algorithm has run, the input texture is available for the next tile. Let the amount of tiles passed simultaneously to the pipeline be q. Instead of allocating q input buffers and q output buffers, the framework can allocate only one input texture and reuse it for all tiles. However, to store all output buffers, q textures are required.

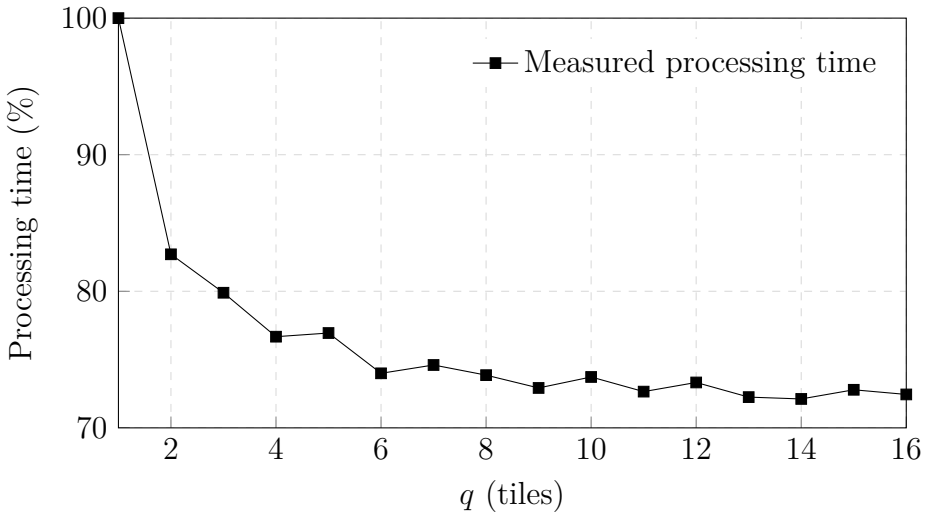


Figure 5.4: The processing time with different q values.

Since the image is split into n_x tiles in horizontal dimension and n_y tiles in vertical dimension, the biggest possible value for q is $n_x n_y$. To find the most optimal value, the total processing time was measured for q values between 0 and 16. The measurements were taken with the optimal values for n_x and n_y which, based on section 5.3, are 14 and 8. Measured processing times are illustrated as a function of q in figure 5.4. The graph shows that the processing time decreases until a certain

optimal number of input textures in reached. The optimal value for q appears to be between 6 and 8. Increasing the value beyond that only increases the memory consumption without giving a significant improvement in performance, since the GPU has enough data to process without going into the idle state.

6. EVALUATION

The framework was evaluated by comparing it to a traditional CPU implementation. The test pipeline used floating point textures and same algorithms as the one used in chapter 5. All measurements were taken using optimized parameters from the same chapter, as summarized in table 6.1.

Table 6.1: The framework processing parameters used in measurements.

Description	**Symbol**	**Value**
Bit depth	b	16
Number of tiles horizontally	n_x	14
Number of tiles vertically	n_y	8
Number of output textures	q	8

6.1 Processing Performance

The processing performance of the framework was measured against single-threaded and multi-threaded CPU implementations. Since the test device had four CPU cores, the multi-threaded version is approximately four times faster than the single-threaded version, hence representing the best available performance on the CPU side. Measured processing times for each algorithm and for the whole pipeline are listed in table 6.2. With the tested pipeline, GPU processing took 78.9% less time than the single-threaded CPU implementation and 40.6% less time than the multi-threaded implementation. The time required for buffer transfers was 41% of the total processing time which clearly highlights the bandwidth bottleneck problem. However, since processing is done in parallel with buffer transfers, the GPU is able to work almost continuously without synchronizing with the CPU.

The measurements also indicate that the GPU is the most efficient in algorithms that are computationally heavy, such as demosaicing. Matrix multiplications in color correction and RGB to YCbCr conversion were also more efficient on the GPU. On the other hand, simple single-pixel algorithms such as linearization and white balance were proven to be faster on the CPU. Also, there is no significant performance improvement in algorithms that were implemented using a lookup table, such as gamma correction. These kind of algorithms can be implemented efficiently

Table 6.2: Measured GPU processing times compared to a CPU implementation.

Algorithm	**Processing time (+/-%)**	**Processing time (multi-threaded) (+/-%)**
Linearization	-28.8%	+98.6%
White Balance	-36.8%	+76.3%
Vignetting Correction	-94.8%	-85.5%
Demosaicing	-83.5%	-54.1%
Color Correction	-92.1%	-78.0%
Gamma Correction	-67.8%	-10.2%
RGB to YCbCr Conversion	-74.5%	-28.8%
Whole pipeline	-78.9%	-40.6%

on the CPU because data can be processed in-place without making a buffer copy, which is inevitable in the GPU implementation.

6.2 Memory Consumption

The memory consumption of the framework was compared to a CPU version which implements a simple ping-pong buffering scheme. On the CPU side, ping-pong buffering can be implemented with two memory buffers that have the same size. The size is determined by the biggest output buffer in the pipeline. To achieve the same image quality as the GPU implementation, 16-bit fixed point values (48 bits per pixel) were used. Therefore, the memory requirement for the CPU implementation is 2 * 6 * 5376 * 3024 B $\approx$ 186.0 MB. The memory consumption of the GPU implementation is highly dependent on parameters n_x, n_y and q. Increasing n_x and n_y decreases the memory consumption while increasing q has an opposite effect. In real use-cases, the framework can be tuned according to requirements and hardware limitations. For example, for low-end devices the memory consumption could be decreased at the cost of increased processing times, while high-end devices with high resolution cameras might have the opposite approach.

The total memory consumption of the framework using optimized parameters is calculated in table 6.3. The listed memory usage denotes the additional memory that is required for the specific buffer. In case the listed value is zero, the algorithm is able to re-use one of the previously allocated textures. Texture formats are listed using a naming scheme defined by Direct3D [32]. The calculations do not include memory required for Direct3D initialization since that is highly dependent on platform and therefore difficult to calculate precisely. Also, small buffers such as vertex and index buffers are not included in calculations since they are insignificant compared to

the total memory consumption. According to the calculations, the total memory consumption including buffers on both CPU and GPU side is 74.2 MB which is 60.1% lower than with a traditional two-buffer CPU implementation.

Table 6.3: The memory consumption of the framework and buffers.

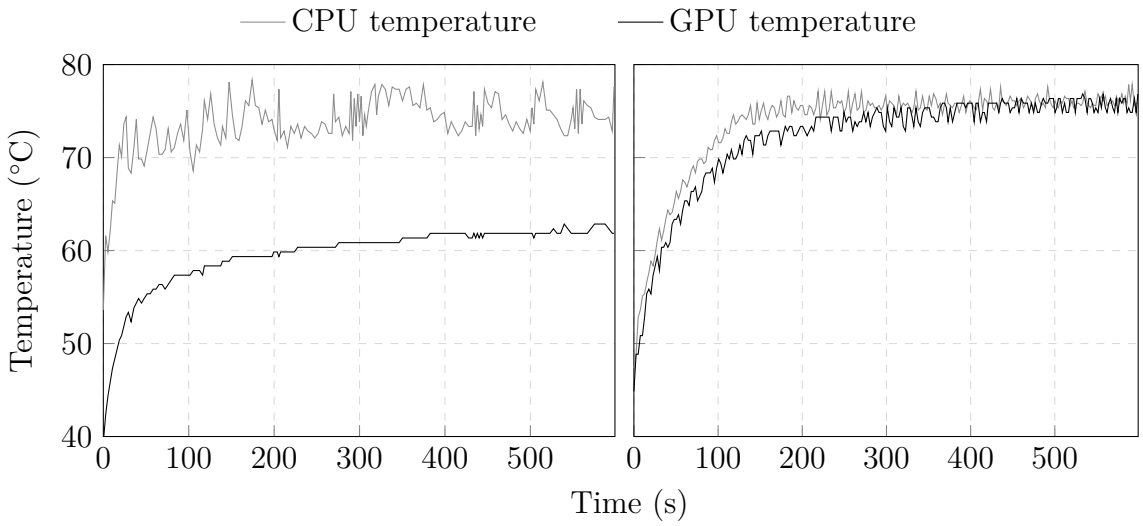
Algorithm	**Texture**	**Format**	**Size**	**Memory usage**
Input Algorithm	Output	R16G16B16A16_FLOAT	195x192	+292.5 kB
Linearization	Input	R16G16B16A16_FLOAT	195x192	+0 kB
	Output	R16G16B16A16_FLOAT	195x192	+292.5 kB
White Balance	Input	R16G16B16A16_FLOAT	195x192	+292.5 kB
	Output	R16G16B16A16_FLOAT	195x192	+0 kB
Vignetting Correction	Input	R16G16B16A16_FLOAT	195x192	+0 kB
	Output	R16G16B16A16_FLOAT	195x192	+0 kB
Demosaicing	Input	R16G16B16A16_FLOAT	195x192	+0 kB
	Output	R16G16B16A16_FLOAT	390x384	+1170.0 kB
Color Correction	Input	R16G16B16A16_FLOAT	390x384	+0 kB
	Output	R16G16B16A16_FLOAT	390x384	+1170.0 kB
Gamma Correction	Input	R16G16B16A16_FLOAT	390x384	+0 kB
	Output	R16G16B16A16_FLOAT	390x384	+0 kB
RGB to YCbCr Conversion	Input	R16G16B16A16_FLOAT	390x384	+0 kB
	Output	R8G8B8A8_UNORM	390x384	+585.0 kB
Output Algorithm	Input	R8G8B8A8_UNORM	390x384x8	+4680.0 kB
Total (textures)				**8482.5 kB**
Input Buffer	-	Raw Bayer (10 bpp)	5376x3024	+19.4 MB
Output Buffer	-	YCbCr 4:4:4 (24 bpp)	5376x3024	+46.5 MB
Total (input and output buffer)				**65.9 MB**
Total				**74.2 MB**

It is possible to split the image into smaller parts to save memory on the CPU side too. Määttä et al. have presented a line-buffer-based image processing framework which managed to achieve a total memory consumption of 636 kB in the pipeline [34]. Although their pipeline contained few algorithms more than the one presented in this thesis, the results clearly indicate that a proper CPU-based pipeline can be implemented with much less memory usage than 8482.5 kB achieved by the GPU-based implementation.

6.3 Thermal Measurements

The compare the heat generation of GPU and CPU, the test pipeline was run in a loop for 10 minutes with both implementations. The measured CPU and GPU

temperatures are plotted in figure 6.1. The measurements indicate that the average temperature increases faster at the beginning in the CPU implementation, but after around 100 seconds, the GPU implementation catches up. One unexpected perception is that utilizing the GPU also increases CPU temperatures considerably. At the end of the measurements, the average temperature of all sensors was 70 °C for the CPU implementation and 76 °C for the GPU implementation. This indicates that the GPU is more likely to cause overheating than the CPU, especially when images are processed excessively as in this test.



(a) CPU pipeline (b) GPU pipeline

Figure 6.1: Measured GPU and CPU temperatures for (a) CPU implementation and (b) GPU implementation during processing.

Another interesting thermal-related property that can be measured is how constant the processing time stays when multiple images are processed in a row. The same pipelines were run with the same input image for 10 minutes and the processing time of each run was measured. The measured processing times are illustrated in figure 6.2. The initial CPU throughput is used as a reference, hence representing the 100% processing time. The graph shows that there is very little variance in the GPU processing time throughout the measurement. In the CPU implementation, however, the measured processing time varies considerably. In addition, the CPU processing time increases significantly after few images have been processed. Throughout the whole measurement, the average GPU processing time was 45% of the initial CPU time while the average CPU processing time was 130% of its initial value.

One reason for the increase in processing times on the CPU side could be that the system is trying to prevent overheating by decreasing the CPU frequency. To examine this issue further, the same pipeline was run again for 10 minutes for

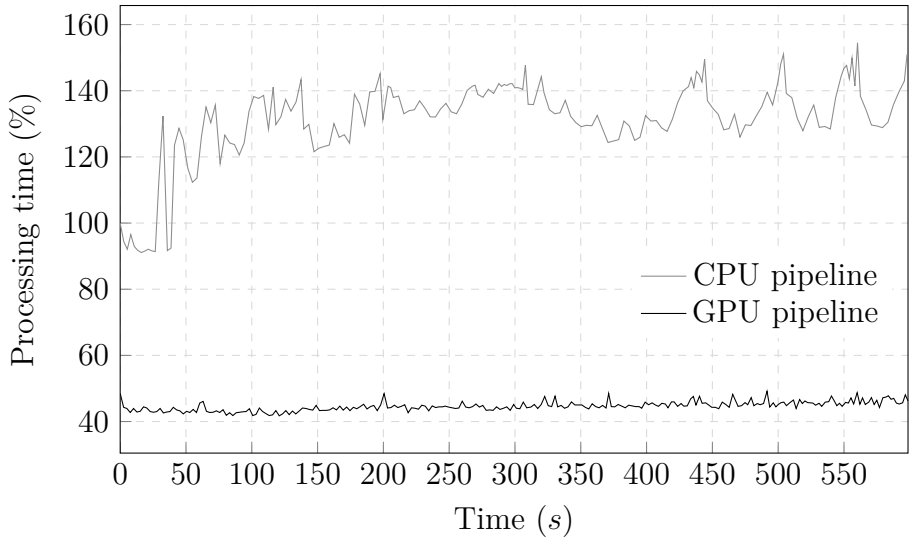


Figure 6.2: The processing time of a single image during measurements.

both implementations, but during processing, the CPU frequency of each core was measured. The average frequency of all four cores is illustrated in figure 6.3. The measurements show that there is plenty of variance in the CPU frequency at the beginning, but after few minutes the values get more stable. The measured CPU frequency at the end of the CPU measurement was 26% less than the initial frequency which explains the drop in throughput.

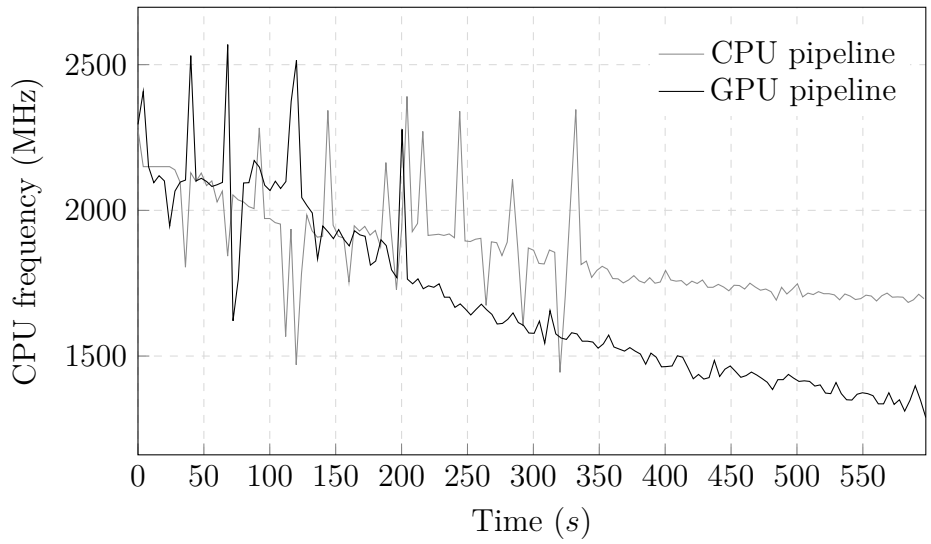


Figure 6.3: The average frequency or all CPU cores during processing.

Another unexpected perception is that the CPU frequency drops even more significantly during GPU processing. Although the CPU frequency at the end of the GPU measurement was 44% less than maximum, there does not seem to be a significant

decrease in GPU processing times. This perception could be explained by the measured temperatures in figure 6.1. Since the total temperature increases constantly during GPU processing, the only option for the system to prevent overheating might be reducing the CPU frequency. Clearly this approach is not enough the stop overheating since temperatures seemed to be increasing even after 10 minutes. On the CPU side, however, the decreased CPU frequency seems to stop temperatures from rising at around 350 seconds. The inability to reduce the GPU frequency to prevent overheating is probably dependent on the platform, but in this case it can cause severe overheating if both CPU and GPU are strained excessively for a long time.

6.4 Power Consumption

The power consumption of the framework was evaluated by processing 1000 images through the pipeline. For every hundred frames, the battery charge of the device was measured using an API provided by the Windows Phone platform. The test device had a 3400 mAh battery which was fully charged before taking the measurements. The measured battery charge levels are plotted in figure 6.4. As expected, the power consumption was linear, so a fitting line was calculated using linear regression. Because the GPU processes images much faster than the CPU, the battery charge is illustrated as a function of number of processed images instead of time.

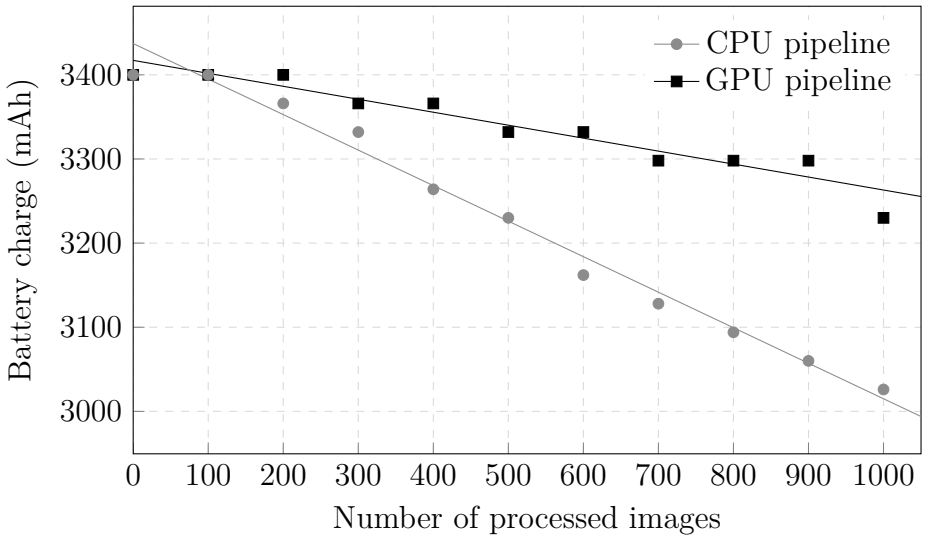


Figure 6.4: The battery charge of the test device when 1000 images were processed consequently.

The measurements show that the GPU requires much less power to process the same image with same algorithms. Based on the linear regression line, the average power consumption and the power consumption per image have been calculated to table 6.4. Since these calculations were made based on the total power consumption

of the test device, also idle power consumption is included in the results.

Table 6.4: Calculated power consumptions.

Implementation	**Power consumption (average)**	**Power consumption (per image)**
CPU	3690 mW	1.65 mWh
GPU	3510 mW	0.603 mWh

The average power consumption was only 5% lower with the GPU implementation, but the power consumption for processing a single image was 63% lower than with the CPU implementation. The difference in the average power consumption is insignificant, but the improvement in per image power consumption is a clear advantage of the GPU implementation. The ability to process images faster on the GPU leads into longer battery duration as long as the number of processed images stays the same.

7. CONCLUSION

This thesis studied the use of GPU as an image processor in mobile devices. The most notable limitation in implementing algorithms was the inability select the output pixel position based on its value. This makes statistics gathering algorithms, for example, really difficult to implement on the GPU. Basic single-pixel and filtering algorithms, however, seemed to suit well for the GPU. The biggest performance gains were achieved with algorithms that were computationally heavy.

A generic framework for GPU-based image processing was introduced to ease the use of GPU for image processing tasks. The framework also made it possible to implement algorithms without having to take care of hardware limitations such as the maximum texture size. According to measurements, the GPU-based implementation was over 41% faster than an equivalent CPU implementation in initial runs. When the pipeline was run for multiple times in a row, the performance difference became even higher, since the GPU was able to keep the processing time constant while the CPU implementation suffered from a serious decrease in performance after the test device warmed up. There was no significant difference in average power consumption, but because of shorter processing times, the GPU was able to process one image with 63% lower power consumption than the CPU.

The biggest drawbacks in using the GPU as an image processor were increased heat and memory consumption. Overheating was visible after the GPU was strained for couple of minutes continuously. In normal use where images are captured randomly, the problem is probably not that critical. Although the memory consumption can be tuned according to requirements, the GPU implementation requires more memory than a line-buffer-based CPU implementation, because there are limitations on how small the size of a single tile can be made before it has a significant impact on processing times.

Despite the observed drawbacks, the GPU provides a really interesting platform for image processing especially on high-end devices where there are plenty of memory and other hardware resources available. It can be assumed that the performance of mobile GPUs continues to increase faster than CPUs which emphasizes the use of GPU in forthcoming products. In the future, the framework will be improved by adding support to DirectCompute which will enable the use of GPU for even more general-purpose computing.

REFERENCES

[1] Nakamura J. Image Sensors and Signal Processing for Digital Still Cameras. Boca Raton 2006, CRC Press. 321 p.

[2] Nummela V. Camera Lenses. 2008. Nokia internal training material.

[3] Lukac, R. Single-Sensor Imaging: Methods and Applications for Digital Cameras. Boca Raton 2009, CRC Press. 600 p.

[4] Litwiller D. CMOS vs. CCD: Maturing Technologies, Maturing Markets. Photonics Spectra 39(2005)8, pp. 54-61.

[5] SMIA 1.0 Part 2: CCP2 Specification [WWW]. Nokia Corporation, ST Microelectronics NV. 2004. [Accessed on 4.3.2014]. Available at: http://www.sunex.com/SIMA/SMIA_CCP2_specification_1.0.pdf

[6] Richardson, I. E. H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia. UK 2003, Wiley. 320 p.

[7] Hamilton, E. JPEG File Interchange Format [WWW]. C-Cube Microsystems. 1992. [Accessed on 10.3.2014]. Available at: http://www.w3.org/Graphics/JPEG/jfif3.pdf

[8] Sullivan, G. & Estrop, S. Recommended 8-Bit YUV Formats for Video Rendering [WWW]. Microsoft Corporation. April 2002, updated on November 2008. [Accessed on 12.3.2014]. Available at: http://msdn.microsoft.com/en-us/library/windows/hardware/dd206750

[9] Russ, J. C. The Image Processing Handbook. 6th Edition. Boca Raton 2011, CRC Press. 838 p.

[10] Gonzalez, R. C. & Woods, R. E. Digital Image Processing. 2nd edition. New Jersey 2002, Prentice Hall. 793 p.

[11] Kao, W.-C., Wang, S.-H., Chen, L.-Y. & Lin, S.-Y. Design Considerations of Color Image Processing Pipeline for Digital Cameras. IEEE Transactions on Consumer Electronics 52(2006)4, pp. 1144-1152.

[12] Bordallo López, M., Nykänen, H., Hannuksela, J., Silvén, O. & Vehviläinen, M. Accelerating image recognition on mobile devices using GPGPU. IS&T/SPIE Electronic Imaging 2011, San Francisco, California, USA, January 23-27, 2011. Bellingham, WA, 2011, SPIE.

[13] Ramanath, R., Snyder, W. E., Yoo, Y. & Drew, M. S. Color Image Processing Pipeline. IEEE Signal Processing Magazine 22(2005)1, pp. 34-43.

[14] Battiato, S., Bruna, A. R., Messina, G. & Puglisi, G. Image Processing for Embedded Devices. Italy 2010, Bentham Science Publishers. 376 p.

[15] Zheng, Y., Lin, S. & Kang, S. B. Single-Image Vignetting Correction. IEEE Transactions on Pattern Analysis and Machine Intelligence 31(2009)12, pp. 2243-2256.

[16] Jia, J. & Tang, C-K. Tensor Voting for Image Correction by Global and Local Intensity Alignment. IEEE Transactions on Pattern Analysis and Machine Intelligence 27(2005)1, pp. 36-50.

[17] Trussell, H.J. & Hartwig, R. E. Mathematics for Demosaicking. IEEE Transactions on Image Processing 11(2002)4, pp. 485-492.

[18] Malvar, H. S., He, L-W & Cutler, R. High-quality linear interpolation for demosaicing of Bayer-patterned color images. IEEE International Conference on Acoustics, Speech, and Signal Processing, Montreal, Quebec, Canada May 17-21, 2004. USA 2004, IEEE.

[19] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. & Purcell, T. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum 27(2007)1, pp. 80-113.

[20] Hughes, J. F., Van Dam, A., McGuire, M., Sklar, D. F., Foley, J. D., Feiner, S. K. & Akeley, K. Computer Graphics: Principles and Practice. 3rd Edition. USA 2013, Addison-Wesley Professional. 1264 p.

[21] Bailey, M. & Cunningham, S. Graphics Shaders: Theory and Practice. 2nd Edition. Boca Raton 2012, CRC Press. 518 p.

[22] Sherrod, A. & Jones, W. Beginning DirectX 11 Game Programming. Boston 2012, Cengage Learning. 384 p.

[23] Pharr, M. & Fernando, R. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. USA 2005, Addison-Wesley Professional. 880 p.

[24] NVIDIA. CUDA Parallel Computing Platform [WWW]. [Accessed on 28.4.2014]. Available at: http://www.nvidia.com/object/cuda_home_new.html

[25] Munshi, A. The OpenCL Specification, version 1.2. [WWW]. Khronos OpenCL Working Group. November 2011, updated on November 2012. [Accessed on 28.4.2014]. Available at: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[26] Microsoft. Compute Shader Overview [WWW]. [Accessed on 28.4.2014]. Available at: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx

[27] Garrard, A. Moving to Mobile Graphics and GPGPU - Forget Everything You Know. SIGGRAPH 2013, Anaheim, USA, July 21-25, 2013. New York, NY 2013, ACM Press.

[28] Cheng, K-T. & Wang, Y-C. Using Mobile GPU for General-Purpose Computing - A Case Study of Face Recognition on Smartphones. 2011 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), Hsinchu, Taiwan, April 25-28, 2011. 2011, IEEE.

[29] Singhal, N., Yoo, J. W., Choi, H. Y. & Park, I. K. Design and Optimization of Image Processing Algorithms on Mobile GPU. In proceeding of: International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2011, Vancouver, BC, Canada, August 7-11, 2011. New York, NY 2011, ACM Press.

[30] Zink, J., Pettineo, M. & Hoxley, J. Practical Rendering & Computation with Direct3D 11. Boca Raton 2011, CRC Press. 648 p.

[31] Engel, W. GPU Pro 2: Advanced Rendering Techniques. Natick, MA 2011, A K Peters. 470 p.

[32] Direct3D 11 Reference [WWW]. Microsoft Corporation. [Accessed on 21.3.2014]. Available at: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476147%28v=vs.85%29.aspx

[33] Wang, L., Zhang, Y. & Feng, J. On the Euclidean Distance of Images. IEEE Transactions on Pattern Analysis and Machine Intelligence 27(2005)8, pp. 1334-1339.

[34] Määttä, J.-M., Vanne, J., Hämäläinen, T. D. & Nikkanen, J. Generic Software Framework for a Line-Buffer-Based Image Reconstruction Pipeline. IEEE Transactions on Consumer Electronics 57(2011)3, pp. 1442-1449.