TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

**JONI-MATTI MÄÄTTÄ**

# DESIGN AND IMPLEMENTATION OF A GENERIC IMAGE RECONSTRUCTION PIPELINE FOR CAMERA PHONES

Master of Science Thesis

Examiners:
Prof. Timo D. Hämäläinen
Dr. Tech. Jarno Vanne
Examiners and topic approved in
the Information Technology
Department Council meeting on
8th May 2013

# TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
**MÄÄTTÄ, JONI-MATTI**: Yleiskäyttöisen kuvankäsittelyliukuhihnan suunnittelu ja toteutus kamerapuhelimiin
Diplomityö, 43 sivua, 0 liitesivua
Toukokuu 2013
Pääaine: Tietokone- ja prosessoritekniikka
Tarkastajat: Prof. Timo D. Hämäläinen, TkT Jarno Vanne
Avainsanat: Ohjelmistokehys, kuvankäsittelyliukuhihna, juovapuskuri, muistinhallinta

Valtaosa nykyisistä edullisen hintaluokan matkapuhelimista sisältää kameran, joka käyttää yksinkertaista optiikkaa ja halpaa kamerasensoria. Riittävän kuvanlaadun takaamiseksi kuvaa käsitellään kuvankäsittelyalgoritmeilla, jotka yhdessä muodostavat kuvankäsittelyliukuhihnan. Paras suorituskyky saavutetaan yleensä rautapohjaisella liukuhihnalla. Ohjelmistopohjaisia ratkaisuja voidaan kuitenkin suosia tuotantokustannusten minimoimiseksi ja liukuhihnan joustavuuden sekä laajennettavuuden parantamiseksi. Jotta liukuhihnan muistinkulutus voidaan minimoida vähämuistisessa ympäristössä, on järkevää toteuttaa ohjelmistopohjainen kuvankäsittelyliukuhihna käyttäen juovapuskureita. Juovapuskurit monimutkaistavat liukuhihnan hallintaa, mikä kuitenkin on ratkaistavissa automaation avulla.

Tämä diplomityö esittelee yleiskäyttöisen ohjelmistokehyksen juovapuskuripohjaiselle kuvankäsittelyliukuhihnalle. Ohjelmistokehys soveltuu vähämuistisiin ympäristöihin ja helpottaa kuvankäsittelyalgoritmien lisäämistä, poistamista ja muuttamista sekä muita liukuhihnan hallintatehtäviä merkittävästi. Säästöt kehitysajassa voivat olla jopa kuukausia. Ohjelmistokehyksen suorituskykyä ja muistinkäyttöä verrataan nykyisiin toteutuksiin käyttäen todellista kuvankäsittelyliukuhihnaa testitapauksena. Työssä pohditaan myös, kuinka juovapuskuripohjainen liukuhihna voitaisiin rinnakkaistaa entistä paremman suorituskyvyn saavuttamiseksi moniydinpuhelimissa. Rinnakkaistetun liukuhihan toteuttamiseksi esitellään kaksi lähestymistapaa: dataviipaloitu liukuhihna ja työjonopohjainen liukuhihna.

Tutkimus osoittaa, että ohjelmistokehys säästää muistia yli 99% verrattuna perinteisiin toteutuksiin, jotka käyttävät ping-pong puskurointia täyden koon kuvapuskureilla. Toteutettu ohjelmistokehys parantaa myös suorituskykyä paremman välimuistin käytön ansiosta ja lisää kuvankäsittelyliukuhihnan joustavuutta useilla erilaisilla konfiguraatioilla.

# ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
**MÄÄTTÄ, JONI-MATTI**: Design and Implementation of a Generic Image Reconstruction Pipeline for Camera Phones
Master of Science Thesis, 43 pages, 0 Appendix pages
May 2013
Major: Computer and processor engineering
Examiners: Prof. Timo D. Hämäläinen, Dr. Tech. Jarno Vanne
Keywords: Software framework, image reconstruction pipeline, line buffer, memory management

The majority of the current affordable mobile devices contain a camera with simple optics and a low-cost camera sensor. In these devices, the quality of the captured images is made acceptable with various image processing algorithms that together form an image reconstruction pipeline. The best performance is often achieved with a hardware pipeline, but software implementations can be preferred to minimize production costs and to maximize flexibility. To minimize memory consumption in such a limited-resource environment, it is reasonable to implement a software-based image reconstruction pipeline using line buffers. The line buffers complicate the management of the pipeline, which, however, can be solved by increasing development tool automatization.

This Thesis presents a generic software framework for a line-buffer-based image reconstruction pipeline. The presented framework is capable of operating in low-memory environments and significantly eases algorithm insertions, changes of processing order, and other pipeline management tasks. The savings in development time can be even months. The performance and memory usage of the software framework is compared to contemporary implementations by using a real image reconstruction pipeline as the test case. The Thesis also discusses how the line-buffer-based pipeline could be parallelized to achieve improved performance on multi-core devices. Two promising approaches are considered: slice-based parallelization and work-queue-based parallelization.

The experiments show that the software framework offers over 99% memory savings compared with traditional implementations using a ping-pong buffer scheme with full-sized image buffers. The implemented framework also enhances processing performance due to better cache usage and increases flexibility with various pipeline configurations.

# PREFACE

This Master of Science Thesis, **Design and Implementation of a Generic Image Reconstruction Pipeline for Camera Phones**, has been done in the Department of Pervasive Computing, at Tampere University of Technology (TUT) during the years 2010–2013. The ground work for the Thesis was made in a joint project between TUT and Nokia Corporation during 2010–2011. The rest of Thesis was finished during Spring 2013. A significant part of the Thesis is based on the following journal article:

*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.*

I want to thank my colleagues at TUT, staff from Nokia, and my family and friends for all guidance and support I got while making this Thesis. I am especially grateful to my instructor Jarno Vanne from TUT for guiding me throughout the whole process, including the collaboration project, the paper and this Thesis, and giving invaluable ideas for all aspects of the Thesis.

# CONTENTS

1 Introduction....1
2 Image Reconstruction Pipeline in Digital Cameras....3
2.1 Special Considerations for Camera Phones....6
2.2 Image Data Formats....7
2.2.1 Raw Bayer....7
2.2.2 RGB888....8
2.2.3 CIELAB....8
2.2.4 YCbCr....9
2.3 Image Processing Algorithms....10
2.4 Methods for Pipeline Memory Management....12
2.4.1 Ping-Pong Buffering....12
2.4.2 Line-Buffer-Based Approach....13
3 Generic Software Framework for a Line-Buffer-Based Image Reconstruction Pipeline....15
3.1 Algorithm Interface....17
3.2 Framework Architecture....18
3.3 Processing Image Borders....20
3.4 Pipeline Configurations....22
4 Line Buffer Memory Management....26
4.1 Reducing Memory Footprint....28
4.2 Cache Associativity Optimization....30
5 Performance Analysis....31
5.1 Basic Experiments....31
5.2 Evaluating the Effects of Cache Associativity Optimization....35
6 Parallelizing the Pipeline....37
6.1 Slice-Based Parallelization....37
6.2 Work-Queue-Based Parallelization....39
7 Conclusion....43
References....44

# TERMS AND DEFINITIONS

| | |
|---|---|
| **AWB** | Auto white balance. A technique to automatically balance the image so that white looks natural white regardless of the lighting conditions. |
| **CCD sensor** | Charge-coupled-device sensor. An image sensor where pixels are represented by p-doped metal-oxide-semiconductor (MOS) capacitors. |
| **CIELAB** | A color space with dimension *L* for lightness, and *a** and *b** for the color-opponent dimensions. |
| **CMOS sensor** | An image sensor that uses complementary metal-oxide-semiconductor (CMOS) technology for its integrated circuit. |
| **Demosaicing** | A procedure where missing color information of the Bayer pattern is produced by interpolating the neighboring pixels, so that each pixel will have three intensity values, one for each primary color. |
| **Line buffer** | A contiguous memory block which is capable of storing the pixels of one horizontal line of an image. |
| **Pixel** | A basic unit of programmable color on a computer display or in a computer image. |
| **RAW Bayer** | An image format that utilizes a specific color filter array to yield, e.g., an RGGB pattern where every odd line of the image is an RG line and every even line is a GB line. |
| **sRGB** | Standard RGB color space where each color is composed of three primary colors: red, green and blue. |
| **Thread pool** | A collection of generic worker threads that are available for processing tasks. |
| **YCbCr** | A widely used color space for representing digital color images, comprising of luminance component *Y* and two chrominance components *Cb* and *Cr*. |

# 1 INTRODUCTION

Image reconstruction is relevant to any device containing a digital camera. In order to have acceptable image quality, raw image data are processed with various image processing algorithms. The widely used image processing algorithms include color filter array (CFA) interpolation, noise filtering, and several color correction operations. The combination of algorithms is organized into an image reconstruction pipeline.

In current mobile devices, the camera modules have to be small and low-cost in order to meet consumer price points. Unfortunately, these aggressively priced modules tend to suffer from non-ideal optics, variance in manufacturing, and incomplete manufacturing testing that may introduce several artifacts, for example, chromatic aberration and dead pixels, into the raw image data acquired by the camera sensor [1]. Thus, long image reconstruction pipelines are usually needed in order to preprocess the raw image data and compensate for existing artifacts. In addition, the image processing algorithms are usually computationally intensive. Due to these reasons, efficient implementation is needed to reach adequate performance. Hardware implementations usually offer the best performance and the lowest energy consumption, but they would cost too much if deployed in low or mid-priced mobile devices. Software implementations, on the other hand, are typically more cost-effective and offer much more flexibility for configuring the pipeline. They allow image processing algorithms to be executed in any desired order and changed, for example, in firmware upgrades. It is also possible to use the same software implementation with different camera modules without any modifications [2]. Hence, it is reasonable to implement the image reconstruction pipeline in software.

In mobile devices, the limited amount of physical memory places particular restrictions on an image reconstruction pipeline. Therefore, full-size image buffers cannot be used to store intermediate results, but they can be replaced by line buffers that minimize the memory usage. In this context, a line buffer is a contiguous memory block that is capable of storing the pixels of one horizontal line of an image. The line buffers are circulated while the image data flow through the pipeline. The line buffers complicate the management of the pipeline, but this overhead can be removed by increasing the automatization level of development tools.

Existing image reconstruction pipelines have been briefly discussed before in [3]-[5] in the context of the used algorithms and their implementations. However, to the best of our knowledge, none of the references consider a line-buffer-based image reconstruction

pipeline that would allow easy reconfiguration of the image processing chain regardless of the used algorithms.

The purpose of this Thesis is to study how a generic line-buffer-based software framework can be designed and implemented so that image processing algorithms can be added and removed from the pipeline and their order of execution can be changed without effort. The performance and memory consumption of the software framework is evaluated by comparing it with contemporary implementations. Also, solutions for pipeline parallelization are discussed.

The rest of the Thesis is organized as follows. The image reconstruction pipeline used in digital cameras and the relevant image formats are introduced in Chapter 2. Chapter 2 also discusses the basics of image processing and describes different memory management schemes for an image reconstruction pipeline. The implementation of the designed line-buffer-based software framework is then described in detail in Chapter 3. Chapter 4 discusses the memory management optimizations applied in the implemented software framework. In Chapter 5, the performance characteristics of the implemented software framework are analyzed and the framework is compared with contemporary implementations. Parallelized implementations for the framework are discussed in Chapter 6. Chapter 7 concludes the Thesis.

# 2 IMAGE RECONSTRUCTION PIPELINE IN DIGITAL CAMERAS

Producing high-quality digital images requires a large amount of processing between the raw camera sensor data and the final output image. Various image processing algorithms are applied to the image in order to enhance the image color and edges, and to remove artifacts, such as noise and geometric distortions from the image. The image processing algorithms are executed in a sequence that maximizes the output quality of each image processing algorithm. The sequential chain of processing stages, where each processing stage executes one specific image processing algorithm, is called an *image reconstruction pipeline* [3].

A typical example of a color image reconstruction pipeline used in digital cameras is illustrated in Figure 2.1. In order to produce the final output image, the raw image data captured from the camera sensor goes through several processing stages. Also, a number of color space conversions are made in the pipeline to be able to execute each image processing algorithm in a color space which is the most suitable for the algorithm in question. The image reconstruction pipelines are usually kept confidential by the digital camera companies [3], so there can be differences between vendor-specific pipelines regarding the number of image processing algorithms, their order of execution and the used color spaces. However, usually they all contain more or less the same pipeline stages than the ones illustrated in Figure 2.1.

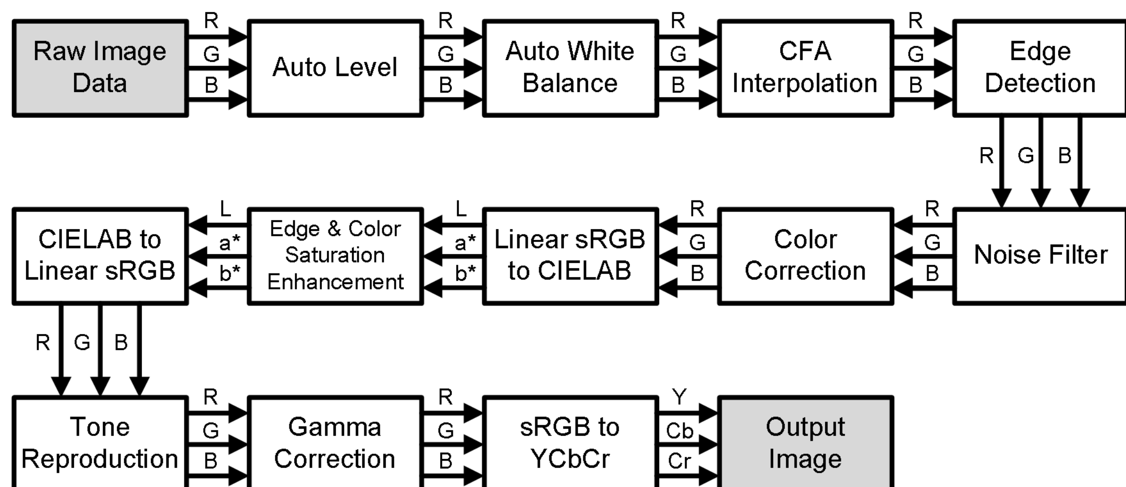



***Figure 2.1:*** *An example of a color image reconstruction pipeline.*

The input image data for the image reconstruction pipeline usually comes from the CCD/CMOS camera sensor [3] in a raw format where each pixel location contains the intensity of only one of the three primary RGB colors, that is, red, green or blue. In practice, this is made by placing a color filter array (CFA) pattern on top of the camera sensor element. Figure 2.2 depicts the Bayer color filter array which is the most commonly used CFA. [4] It can use, for example, RGGB pattern in which every odd line is an RG line and every even line is a GB line. Thus, there is 50% green, 25% red and 25% blue pixels in the resulted raw image data. The amount of green pixels is dominant because the human eye is most sensitive to green light. [6, pp. 489–490]

| R | G | R | G | R | G |
|---|---|---|---|---|---|
| G | B | G | B | G | B |
| R | G | R | G | R | G |
| G | B | G | B | G | B |
| R | G | R | G | R | G |
| G | B | G | B | G | B |

***Figure 2.2:*** *The Bayer color filter array.*

In the example image reconstruction pipeline, the RAW Bayer image data is processed through the following stages (see Figure 2.1):

***Auto level***. The auto level operation clamps the black level and corrects the exposure value [3]. The black level requires clamping because even black color induces a dark current to the camera sensor due to thermally generated electrons in the sensor substrate [4]. The auto level operation first calculates the histogram of the raw image data and determines two control points $\alpha$ and $\beta$ corresponding to 0.1% and 99.9% of the accumulated histogram, respectively. Then the dynamic range between the control points is stretched out to fit the full dynamic range of the output. The control point $\alpha$ defines which intensity value of the original image is mapped to zero, whereas the control point $\beta$ defines the intensity value to be mapped to $2^b-1$ which is the maximum intensity value available for a bit depth *b*. All intensity values outside of the range $[\alpha, \beta]$ are discarded. The pixels of the Bayer pattern are processed without taking the color channel of the pixel into account. Hence, the image is treated as a monochrome image at this stage. [3]

***Auto white balance (AWB)***. The human eye is able to adjust to different lighting conditions based on its memory of white, so that white is perceived as white even in blue or pink light. However, contrary to the human eye, the camera sensor captures the

true illumination of the scene. The purpose of AWB is to determine the lighting conditions in the scene and adjust for them, so that white looks white regardless of the illumination. [6]

***CFA interpolation***. In this stage, the missing color information of the Bayer pattern is produced by interpolating the neighboring pixels, so that each pixel will have three intensity values, one for each primary color. This procedure is also called demosaicing. [4] There are several techniques available for demosaicing, such as bilinear interpolation, constant hue-based interpolation and gradient based interpolation. Of these demosaicing methods, bilinear interpolation is the simplest one. However, due to its band-limiting nature, it smooths the image causing color fringes. [7]

***Edge detection***. The interpolated RGB image is then passed to an edge detection algorithm which extracts the edges from the image [3]. A well-known algorithm for extracting the edge information is the Sobel filter [6].

***Noise filter***. This stage reduces noise in the image. The previously extracted edge information is used in this stage to prevent edges from being smoothed out while the noise is reduced from the other parts of the image. [3]

***Color correction***. In the color correction phase, the image is transformed from sensor RGB color space to linear sRGB space. This color calibration step, when performed well, makes it possible to achieve better color reproduction in the image. [3]

***Linear sRGB to CIELAB***. The image is transformed from linear sRGB color space to CIELAB color space which is the preferred color space for edge and color saturation enhancement [3].

***Edge & color saturation enhancement***. The reason for using CIELAB color space in this stage is that it can express color differences well. Edge enhancement is applied only to the $L$ component using the same edge information that was extracted in the edge detection phase. Color saturation enhancement, on the other hand, is made by scaling the $a*$ and $b*$ components in a way that the hue and brightness of the pixels is preserved while the color saturation is increased. [3]

***CIELAB to linear sRGB***. In this stage, the image is transformed back to linear sRGB color space for tone reproduction.

***Tone reproduction***. In tone reproduction, the high dynamic range of the original image is mapped to a low dynamic range of standard image file formats in a way that preserves the visual brightness and contrast of the original scene. Usually, the compression is based on the minimum and maximum luminance or average luminance of the original image. [8]

***Gamma correction***. The purpose of gamma correction is to account for the non-linearity of the monitors.

***sRGB to YCbCr***. At the end of the pipeline, the image is transformed from the linear sRGB color space to YCbCr color space ready for JPEG compression [3].

## 2.1 Special Considerations for Camera Phones

Mobile phones as a digital imaging platform poses many challenges for the image reconstruction pipeline. The camera sensor module in mobile phones must be small and low-cost, which impacts the image quality. The use of non-ideal camera optics causes artifacts to the image acquired from the camera sensor. The possibility for dead pixels is also much greater than in normal digital cameras due to incomplete manufacturing testing. The image reconstruction pipeline must be designed to compensate for such variations in the sensor data. [1] Usually this means creating a longer image reconstruction pipeline with additional processing stages. Some of artifacts are relevant also in normal digital cameras, but their influence is greatly increased in mobile imaging platforms in which they should be taken into account.

The small size of the camera optics shortens the distance between the main lens and the pixel array. This results in chromatic aberrations and blurriness near the edges of the sensor. [1] Chromatic aberration is a type of distortion in which color fringes are produced along the edges of the image. Image processing algorithms for detecting and eliminating chromatic aberration have been successfully developed. [9] Blurriness, on the other hand, can be reduced by using a smart sharpening filter which sharpens only the border regions of the image.

Another common artifact in camera phone images is vignetting, due to which the brightness of the image attenuates towards the edges of the sensor, as seen in Figure 2.3a. The reason for this is that a portion of the incoming light is obstructed by the field stop or lens rim near the edges of the sensor. Vignetting correction is rather challenging because the vignetting pattern is dependent on the camera settings. However, adaptive

(a) (b)

***Figure 2.3:*** *Vignetting artifact. (a) An image with strong vignetting. (b) Same image after vignetting correction.*

techniques for eliminating vignetting have been proposed. [10] Figure 2.3b visualizes the image after vignetting correction.

The image acquired from the camera sensor may also suffer from geometrical distortions caused by the optical lens. A typical form of geometrical distortion in digital cameras is the barrel distortion where the outer regions of the image appear smaller than the center of the image. This causes the image to look like it has been wrapped around a barrel, as illustrated in Figure 2.4a. Another possible form of geometrical distortion is the pincushion distortion which is the opposite of barrel distortion. Pincushion distortion makes the edges of the image to expand causing straight lines to bend outward from their ends. [11] This is illustrated in Figure 2.4b.

## 2.2 Image Data Formats

Since the image reconstruction pipeline utilizes many color spaces, different image data formats are needed when storing the image data between the processing stages. One color space may even have several possible data representations in terms of data ordering and bit depth. Data ordering determines how the pixel data is organized in the memory. The choice of bit depth, on the other hand, is a compromise between accuracy and memory usage. What follows next is an explanation of different image data formats that are used in a typical image reconstruction pipeline.

### 2.2.1 Raw Bayer

According to CCP2 specification [12], RAW Bayer data coming from the camera sensor of the mobile phone can have a bit depth of 6, 7, 8, 10 or 12 bits. The bit depth specifies the number of bits used to store each pixel intensity value regardless of the pixel color. The 8-bit format is the only one that directly follows the byte boundary. A naive way to

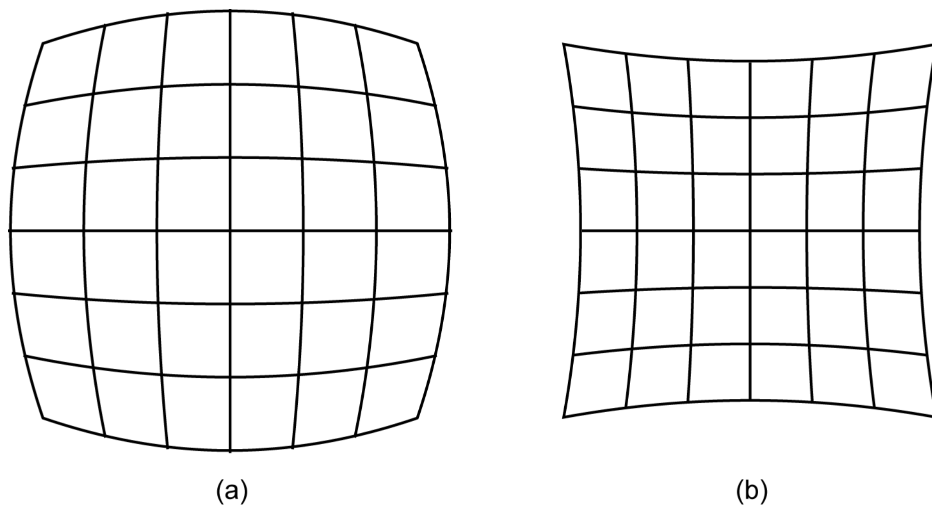
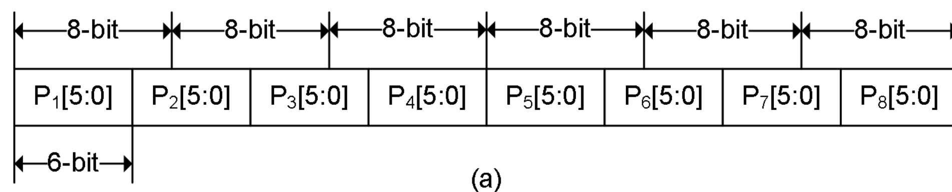



***Figure 2.4:*** *(a) Barrel distortion. (b) Pincushion distortion.*

store the other formats is to introduce padding. For example, each 6-bit intensity value could take one byte of storage space. However, this would mean that two bits of memory would be wasted for each intensity value. According to CCP2, packing is used instead to make better use of the memory. Through packing, four 6-bit intensity values can be stored into three bytes, as is illustrated in Figure 2.5a. The 7-bit RAW Bayer format follows the same packing scheme than the 6-bit format, but the 10-bit and 12-bit formats are packed in a slightly different way. The 10-bit pixels, for example, are packed into bytes by storing the eight most significant bits of four pixels into the first four bytes, and the remaining two least significant bits into the fifth byte. This is illustrated in Figure 2.5b.



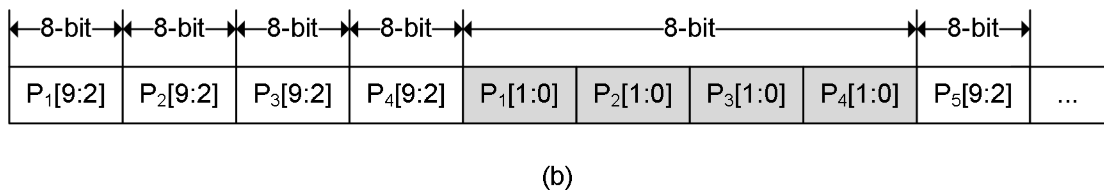




***Figure 2.5:*** *Packing RAW Bayer pixels into bytes. (a) four 6-bit pixels fit directly into three bytes (b) four 10-bit pixels are packed into five bytes, MSBs first. [12]*

RAW Bayer data can be stored into a one-dimensional contiguous buffer in a row-wise manner. The bit depth determines the requirements for the line length. Usually it is convenient if the byte and pixel boundaries are aligned at the end of each line.

### 2.2.2 RGB888

Full-color RGB images are stored in a 24-bit format, where the intensity of each of the three primary colors, red, green and blue, is represented in eight bits. This image data format is called RGB888. [13, pp. 412–413] A typical way to store RGB888 data is to interleave the color components in a raster scan order, so that all three color components of each pixel are stored in adjacent memory locations. The other way to store RGB888 data is to have three separate buffers, one for each color channel.

### 2.2.3 CIELAB

CIELAB is a perceptually uniform color space in which color differences are expressed similarly as in the human eye. CIELAB color space consists of three color components: $L$, $a*$ and $b*$. The achromatic $L$ channel indicates the lightness of the color. The

chromatic $a^*$ channel is a red-green opponent channel where the positive values indicate redder colors and the negative values greener colors. The chromatic $b^*$, on the other hand, is a yellow-blue opponent channel similarly to $a^*$. [3]

### 2.2.4 YCbCr

The YCbCr color space is widely used in digital color images. It seizes the fact that the human visual system is more sensitive to luminance than color. Thus, the luminance component *Y* is usually represented with a higher resolution than the chrominance components *Cb* and *Cr*. There are three sampling formats available for storing the YCbCr data: 4:4:4, 4:2:2 and 4:2:0. The sampling patterns are illustrated in Figure 2.6. In 4:4:4 sampling, the chrominance components are sampled at the same resolution as the luminance component. In 4:2:2 sampling, the chrominance components have the same vertical resolution as the luminance but only half the horizontal resolution. Finally, in 4:2:0 sampling, the chrominance channels have only half the resolution of the luminance in both horizontal and vertical direction. [14, pp. 17–19]

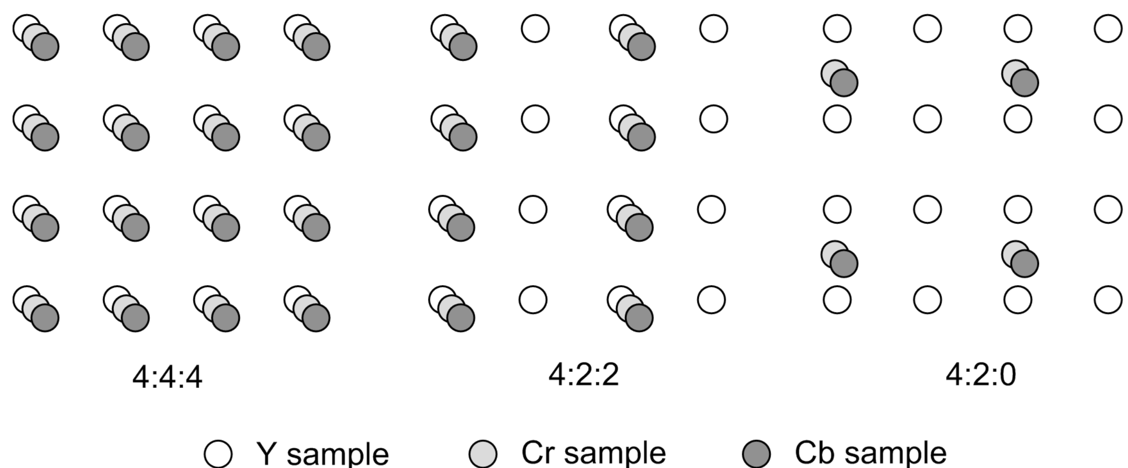



***Figure 2.6:*** *4:4:4, 4:2:2 and 4:2:0 sampling patterns for YCbCr image format. [14]*

The components of YCbCr are each stored in eight bits. The data is ordered either in interleaved or planar format, or in separate buffers. In interleaved format, the samples of *Y*, *Cb* and *Cr* are interleaved together in a raster scan order according to Figure 2.7a. There are two versions of the planar format: line planar and frame planar. In the line planar format, the image is stored line by line, storing the complete *Y* component of a line followed by the line's *Cb* and *Cr* components (illustrated in Figure 2.7b). The frame planar format, on the other hand, first stores the full *Y* component of the whole image followed by the full *Cb* and *Cr* components (illustrated in Figure 2.7c).

| Y0 | Cb0 | Y1 | Cr0 | Y2 | Cb1 | Y3 | Cr1 |
|---|---|---|---|---|---|---|---|
| Y4 | Cb2 | Y5 | Cr2 | Y6 | Cb3 | Y7 | Cr3 |
| Y8 | Cb4 | Y9 | Cr4 | Y10 | Cb5 | Y11 | Cr5 |
| Y12 | Cb6 | Y13 | Cr6 | Y14 | Cb7 | Y15 | Cr7 |

(a) 4:2:2 Interleaved

| Y0 | Y1 | Y2 | Y3 | Cb0 | Cb1 | Cr0 | Cr1 |
|---|---|---|---|---|---|---|---|
| Y4 | Y5 | Y6 | Y7 | Cb2 | Cb3 | Cr2 | Cr3 |
| Y8 | Y9 | Y10 | Y11 | Cb4 | Cb5 | Cr4 | Cr5 |
| Y12 | Y13 | Y14 | Y15 | Cb6 | Cb7 | Cr6 | Cr7 |

(b) 4:2:2 Line Planar

| Y0 | Y1 | Y2 | Y3 |
|---|---|---|---|
| Y4 | Y5 | Y6 | Y7 |
| Y8 | Y9 | Y10 | Y11 |
| Y12 | Y13 | Y14 | Y15 |
| Cb0 | Cb1 | Cb2 | Cb3 |
| Cb4 | Cb5 | Cb6 | Cb7 |
| Cr0 | Cr1 | Cr2 | Cr3 |
| Cr4 | Cr5 | Cr6 | Cr7 |

(c) 4:2:2 Frame Planar

***Figure 2.7:*** *Data ordering formats for YCbCr 4:2:2.*

## 2.3 Image Processing Algorithms

Image processing algorithms can be divided into two categories: algorithms that are executed in the spatial domain and algorithms that are executed in the frequency domain. Spatial domain refers to the original image space, and thus, algorithms executed in the spatial domain directly manipulate the pixels of the image. Algorithms in the frequency domain, on the other hand, use Fourier transform to convert the image into the frequency domain where the processing takes place. [15] Only spatial domain algorithms are practical for digital cameras, since Fourier transform is too complex operation for digital camera image reconstruction pipelines.

Spatial domain algorithms can be further classified into single-pixel operations and filtering operations. A single-pixel operation uses only the original pixel ( $in_{xy}$ ) as input and transforms it to the output pixel ( $out_{xy}$ ) by some function $out_{xy} = f(in_{xy})$ , where *x* and *y* denote the coordinates of the pixel. Gamma correction $out_{xy} = in_{xy}^{\gamma}$ is an example of a single-pixel operation where $\gamma$ denotes the encoding (< 1) or decoding (> 1) gamma value. [15]

In contrast to the single-pixel operations, filtering operations require additional information about the neighboring pixels, whose selection is specified by a *filter kernel*. Usually, rectangular filter kernels are used. [15] Figure 2.8 illustrates the operation of a

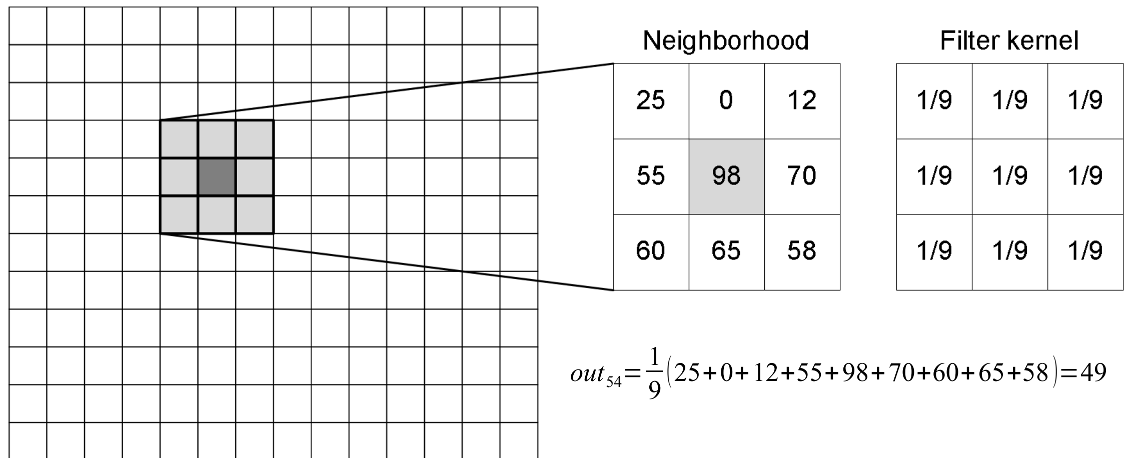



***Figure 2.8:*** *Operation of a mean filter with a* $3\times3$ *kernel size.*

mean filter on a grayscale image using a filter kernel of size $3\times3$. The filter kernel is centered around the original pixel. Odd-sized kernels are more widely used, since even-sized kernels do not have an unambiguous center pixel and, thus, result in a non-symmetrical neighborhood.

When the filter kernel is placed on the image borders, it goes partially outside of the original image. A scenario of such placement is illustrated in Figure 2.9a in which a $5\times5$ kernel is placed on the top border of the image. Gonzales [15] describes several ways to overcome this issue. One solution is to restrict processing so that the filter kernel is always placed fully inside of the original image. Processing an image this way produces a cropped output image. The decrease in size depends on the size of the filter kernel is. However, usually the original image size needs to be preserved and cropping is not desirable. Fortunately, there are several ways to fill in the missing pixels outside of the image. The first one is to pad the image with zeros which corresponds to treating missing pixels as black pixels in RGB space. This is illustrated in Figure 2.9b. A better approach is to mirror the image on the borders so that the existing image data are used in place of the missing data. Mirroring can be performed in two ways: *Direct mirroring* (see Figure 2.9c) mirrors the image data along the border of the image, whereas

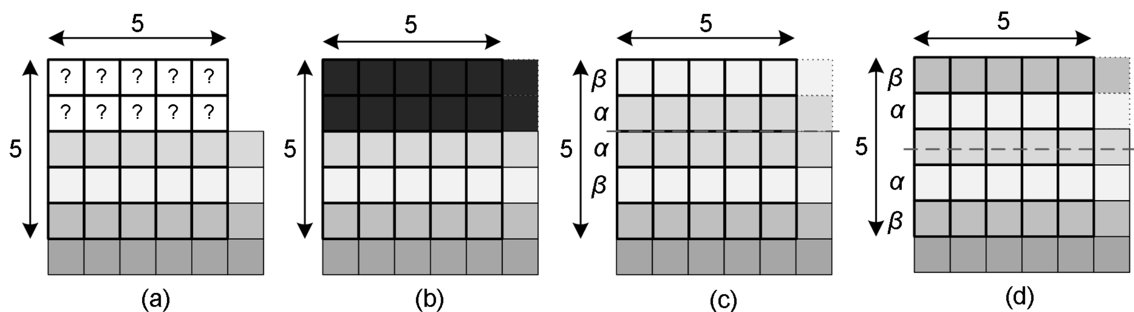



***Figure 2.9:*** *(a) On image borders, the filter kernel goes partially outside of the image. (b) Padding with zeros. (c) Direct mirroring. (d) Alternating mirroring.*

*alternating mirroring* (see Figure 2.9d) uses the border pixel center as the mirroring axis. The applied mirroring technique depends on the image format. For example, RGB888 allows both alternatives, but only alternating mirroring can be applied to RAW Bayer images to preserve correct data order of alternating RG and GB lines.

## 2.4 Methods for Pipeline Memory Management

In the image reconstruction pipeline, each processing stage uses the output of the previous stage as its input. In order to be able to execute the entire pipeline, temporary processing results must be somehow stored between the processing stages. With a hardware image reconstruction pipeline this is straightforward, since the involved memory modules are built right into the circuit design. However, if the image reconstruction pipeline is implemented in software, the temporary processing results must be stored into the internal memory of the device. Usually the amount of internal memory is quite low in affordable mobile phones so only a limited amount of memory can be used by the image reconstruction pipeline. Since many of today's mid-range camera phones have at least a 5-megapixel camera, meaning that one image takes roughly 14 megabytes of memory in RGB888 format, the memory restriction is quite relevant. Thus, special attention must be paid on pipeline memory management to minimize memory usage within the execution of the image reconstruction pipeline.

An unoptimized implementation of the image reconstruction pipeline would use full image buffers to store the outputs of all processing stages. In this case, the reference image reconstruction pipeline with thirteen processing stages would need thirteen image buffers, each utilizing an appropriate image data format for storing the processing results. This includes the final output image buffer which is required no matter how the memory is managed inside the pipeline. Assuming that the 5-megapixel camera sensor produces 8-bit RAW Bayer data and the pipeline uses YCbCr 4:4:4, 24-bit CIELAB and RGB888 formats, the total amount of memory needed for the image buffers would be 164 megabytes. This kind of memory usage would be infeasible.

### 2.4.1 Ping-Pong Buffering

Memory usage of the unoptimized implementation can be decreased by using a ping-pong buffer scheme. In the ping-pong buffering, only two image buffers are used to hold image data. One of the image buffers is used as the input buffer while the other one acts as the output buffer. After completing the execution of a processing stage, the buffers are swapped so that the output buffer of the completed stage is used as the input buffer in the next stage and vice versa. The output buffer of the last processing stage contains the final output image, so there is no need for a separate output image buffer. This design pattern is adopted from games [16]. Figure 2.10 illustrates the operation of the image reconstruction pipeline using the ping-pong buffering.

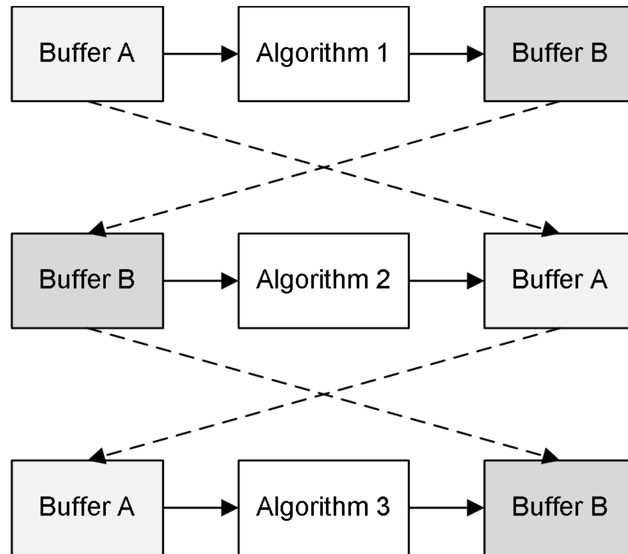



***Figure 2.10:*** *Ping-pong buffering uses two buffers which are swapped after each image processing algorithm.*

When using the ping-pong buffering, the size of the image buffers is specified by the largest image data format used in the pipeline. This way the buffers are capable of holding the processing results regardless of which processing stage is in question. In the previous example, the memory usage would be roughly 28 megabytes, since the largest image data format in the example uses 24 bits per pixel. The drop from 164 megabytes to 28 megabytes is a substantial improvement, but it is still not enough for mobile phones.

### 2.4.2 Line-Buffer-Based Approach

A more advanced approach of reducing memory usage is to use line buffers instead of image buffers. A line buffer is a contiguous memory block which is capable of storing the pixels of one horizontal line of an image. An example of a line buffer is illustrated in Figure 2.11. To take advantage of using line buffers instead of image buffers, the pipeline logic must be line buffer based. This means that each processing stage must process data as soon as it has enough input line buffers to be able to produce at least one new output line. This is in contrast to the unoptimized implementation and the ping-pong buffering, in which each processing stage must be fully completed before the execution can move on to the next stage.

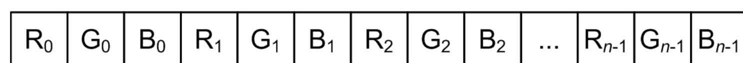



***Figure 2.11:*** *A line buffer holding interleaved RGB data of one horizontal line of an image.*

The number of line buffers required within the execution of the pipeline depends on the kernel sizes of the used image processing algorithms. For example, an image processing algorithm that uses a $5 \times 5$ filter kernel requires five input line buffers to be available in order to produce at least one output line. Assuming that the image processing algorithm in question produces always exactly one output line, the processing stage executing the algorithm requires six line buffers to manage input and output data. The total memory usage of the image reconstruction pipeline can be calculated by summing up the requirements of each of its processing stages. Using line buffers, memory savings can be remarkable. Therefore, it is reasonable to implement a software-based image reconstruction pipeline using line buffers.

# 3 GENERIC SOFTWARE FRAMEWORK FOR A LINE-BUFFER-BASED IMAGE RECONSTRUCTION PIPELINE

A hard-coded line-buffer-based image reconstruction pipeline is almost impossible to maintain since every small modification may break it apart, whether it is simply changing the order of the image processing algorithms or a modification to a certain algorithm. A generic software framework was designed to create a highly reconfigurable image reconstruction pipeline that allows image processing algorithms to be added and removed from the pipeline and their order of execution to be changed without effort. In addition, one essential design aspect was to abstract the management of line buffers so that the individual image processing algorithms do not need to manage circulation and mirroring of lines by themselves. The software framework handles every detail of line buffer management and takes care of giving the correct lines to each algorithm.

The basic structure of the software framework is illustrated in Figure 3.1. The image reconstruction pipeline consists of an ordered set of $K$ processing stages through which the image is processed. Each processing stage is capable of executing one image processing algorithm. The software framework is made generic by abstracting image processing algorithms in a way that allows them to be used via a common algorithm interface. Through interface abstraction, all the processing stages can execute the algorithms in a unified way. To support as many image data formats as possible, the software framework does not make any assumptions about the contents of the line buffers—any image data format that can be represented in line buffers is supported.

Full-size image buffers would allow each image processing algorithm to be executed for the entire image before continuing to the next algorithm. However, in the line-buffer-based approach, the amount of line buffers is minimized by processing the

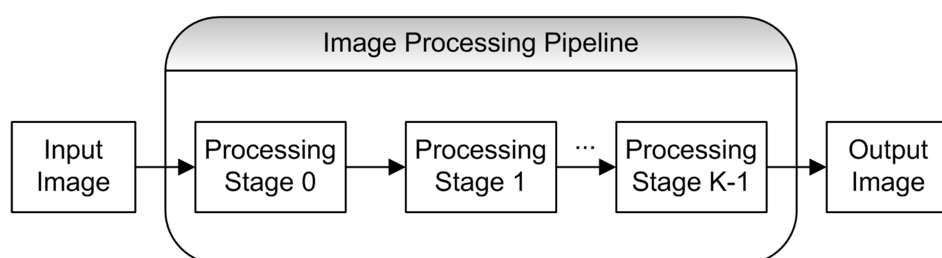



***Figure 3.1:*** *The basic structure of the software framework.*

available data up to the latest possible stage. That is, whenever the image processing algorithm of the processing stage $k$ produces enough output lines for the stage $k + 1$, the image processing algorithm of the stage $k + 1$ is executed. The execution continues as deep in the pipeline as possible before returning back to the beginning of the pipeline. Figure 3.2 illustrates the operation of the pipeline.

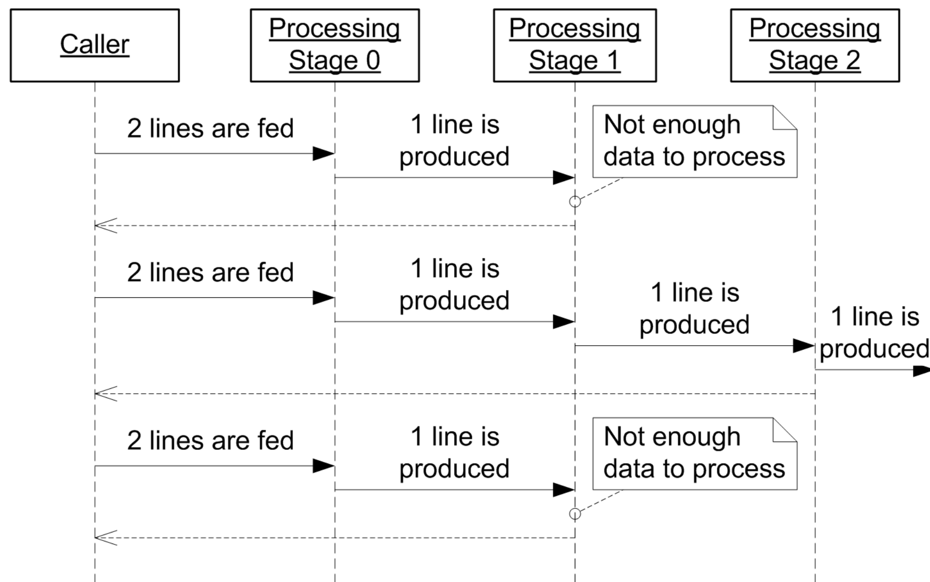



***Figure 3.2:*** *Processing image data in the line-buffer-based image reconstruction pipeline.*

Some camera systems handle image processing by first saving the image as an intermediate file to the file system and then loading and processing the image in a background process [17]. Streaming can be used to read the contents of the intermediate file without having to store the whole input image in the physical memory. In order to support streaming, the software framework allows the input image to be sent to the image reconstruction pipeline in groups of a few lines. In the extreme case, the lines of the input image can be sent to the pipeline one by one.

Memory savings through line buffers are not so remarkable, if the output image needs a full-sized image buffer. Therefore, the software framework allows a partial-sized system output buffer to be used to hold the output image data. The pipeline is stalled each time the system output buffer gets full, so that the contents of the buffer can be read before continuing the execution. The contents of the system output buffer can be, for example, sent to a JPEG compressor which directly streams the compressed image to a file, thus, having no need to store the full output image in the physical memory.

## 3.1 Algorithm Interface

The algorithm interface of the software framework must be able to support both single-pixel and filtering operations efficiently. The key to a common algorithm interface between single-pixel and filtering operations is that single-pixel operations are treated as filtering operations with a $1 \times 1$ filter kernel. Thus, all supported operations can be classified as filtering operations having a fixed rectangular filter kernel. At processing stage $k$, the following input parameters are specified in the algorithm interface: *the number of input lines* ($h_k^I$), *the maximum number of output lines* ($h_k^O$), *filter kernel placement on the top* ($t_k$) *and bottom* ($b_k$) *borders*, *and the algorithm function pointer* ($p_k$).

The parameter $h_k^I$ equals the vertical size of the filter kernel. The image processing algorithm can be run as soon as there are enough input line buffers to cover the whole filter kernel. The horizontal size of the filter kernel is not specified, because the image processing algorithm manages the actual processing in the horizontal direction.

The parameter $h_k^O$ determines the upper bound for the output lines per a single execution. Hence, it also defines the amount of line buffers reserved for the pipeline output. In general, image processing algorithms can be divided into three categories: fixed number of output lines ($N$), variable number of output lines ($0 \ldots N$), and no output lines. Usually filtering operations take a fixed number of input lines and produce exactly one output line per execution. That is, they belong to the first category. In the second category, the number of output lines may vary. For example, a $2\times$ downscaling algorithm produces one output line on only every second time of execution, whereas a $2\times$ upscaling algorithm produces two output lines on each execution. An algorithm that collects statistics may not produce output lines at all, so it belongs to the third category.

The parameter $t_k$ equals the number of lines the filter kernel should be placed outside the top border of the input image at the beginning. This ensures that the first output line corresponds to the first line of the input image. The parameter $b_k$ determines how many lines the filter kernel should go outside the bottom border of the image at the end. This indicates when to stop processing. By having these parameters, the placement of non-symmetrical and even-sized filter kernels is supported unambiguously.

The pointer $p_k$ specifies the actual implementation of the image processing algorithm. The input and output line buffers are passed to the algorithm function in array parameters. Algorithm-specific processing parameters, on the other hand, are passed to the algorithm function as a void pointer. After execution, the algorithm function returns the number of written output lines ($o_k$).

## 3.2 Framework Architecture

The most essential part of the framework architecture is the operation of the processing stages. In the software framework, each processing stage is designed as a modular component which is capable of executing any image processing algorithm that complies with the algorithm interface. Thus, the image reconstruction pipeline can be simply set up by forming a chain of similar processing stages and specifying the algorithm and its related information for each processing stage. Processing stages use a shared memory manager to allocate and free line buffers.

The operation of the processing stage $k$ ( $k \in [0, K-1]$ ) is illustrated in Figure 3.3. The processing stage manages input and output line buffer (LB) arrays. The sizes of the arrays are determined based on the algorithm interface. The line buffers are always copied by reference, since making full copies of their contents would significantly degrade performance.

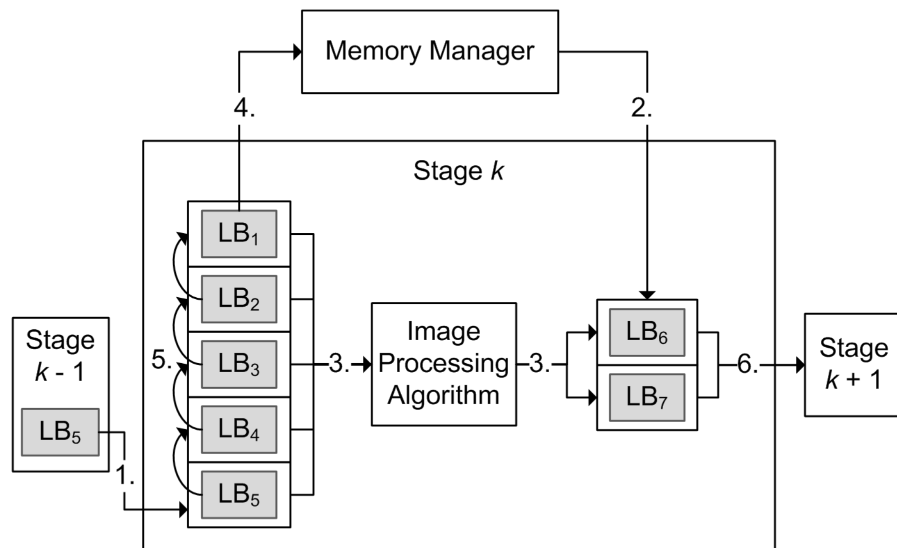



***Figure 3.3:*** *The operation of the processing stage.*

The operation can be divided into six phases:

***Phase 1.*** The stage $k$–1 feeds one line buffer to the stage $k$. The line buffer is inserted into the input line buffer array at the last index.

***Phase 2.*** The output line buffer array is filled with free line buffers by acquiring them from the memory manager. The maximum number of output line buffers is always reserved, since the software framework is unaware of the amount of line buffers the image processing algorithm needs each time.

***Phase 3.*** The image processing algorithm is run. The algorithm fills some of the output line buffers with valid data and indicates how many output line buffers have been written.

***Phase 4.*** The oldest input line buffer at index 0 is released back to the memory manager because it is not needed anymore.

***Phase 5.*** The remaining input line buffers are moved one index up in the input line buffer array so that it is ready to receive the next input line buffer.

***Phase 6.*** The output line buffers having valid data are fed to the next stage one by one. The unused output line buffers are returned to the memory manager.

The processing stage is also parameterized with the line buffer sizes of the input and output images, and the corresponding image heights. The line buffer sizes are required to acquire correct line buffers from the memory manager. The height of the input image is used to handle the mirroring of image data at the bottom border of the image. The height of the output image can be used to make an algorithm to automatically flush its remaining output lines after all input lines have been given to it. The software framework keeps calling the algorithm function until the total number of output lines reaches the height of the output image.

The line buffer sizes and image heights change as a function of the original input image size. The new parameters need to be updated to all processing stages individually, since the framework cannot automatically manage line buffer sizes and image heights with algorithms that apply scaling. The framework can be configured with new parameters at run-time. A run-time configuration phase is executed each time the pipeline needs to process input images of different size.

In the case of a partial-sized system output buffer, the software framework stalls the image reconstruction pipeline immediately when the system output buffer gets full. Before stalling, the current state of the pipeline is stored to allow the execution to be continued from the point where it was interrupted. Since the stalled pipeline may contain data that can be processed before feeding new input lines, the pipeline is drained until all data have been processed.

Feeding and draining are illustrated in Figure 3.4. Feeding is done from left to right. However, draining is executed in the reverse order, because the output line buffer arrays may contain output lines that have not been fed to the next processing stage. These remaining output lines have to be fed to the next stage before the previous stage can continue its execution. Otherwise, the remaining output lines would be overwritten and lost.

Draining begins from the last processing stage and moves towards the beginning of the image reconstruction pipeline. At each processing stage, the remaining output lines are fed to the next processing stage. Feeding is continued towards the end of the

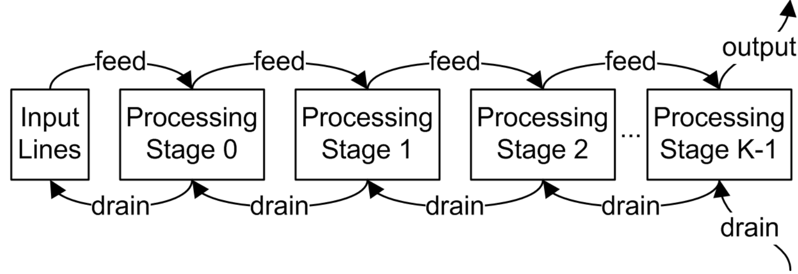



***Figure 3.4:*** *Feeding and draining in the image processing pipeline.*

pipeline until there is nothing more to process. After that, draining continues to the previous processing stage. When the first processing stage has fed the remaining output lines to the next stage, the software framework feeds the remaining system input lines to the pipeline. This ensures that all original input lines are always fed to the pipeline. At any time of draining, the output buffer can get full and the pipeline needs to be stalled. Draining must be repeated until the image reconstruction pipeline ends up in a non-stalled state.

## 3.3 Processing Image Borders

The software framework is unaware of the used image data formats, so the framework is unable to determine the correct mirroring on the left and right borders of the image. Therefore, horizontal mirroring is left as the responsibility of the image processing algorithms. Mirroring on the top and bottom borders, however, can be handled by the software framework without effort. The first image processing algorithms in the pipeline usually deal with Bayer image format which has different odd and even lines. To ensure correct Bayer data order, mirroring has to be done using the vertical center of the topmost or bottommost line as the mirroring axis. Thus, alternating mirroring is used.

In order to support mirroring, the size of the input line buffer array needs to be larger than the input line count $h_k^I$. By that way, all the needed input lines can be stored at any time of execution. At stage $k$, the size of the input line buffer array ( $c_k^I$ ) is determined by

$$c_k^I = max(2t_k + 1, h_k^I + b_k) \ . \tag{1}$$

The term $2t_k+1$ is the number of array elements required on the top border and $h_k^I + b_k$ on the bottom border.

Filling the input line buffer array is started from the element at index $0 + t_k = t_k$, so that after mirroring the top border, the input line at index 0 corresponds to the first line of the filter kernel at the very first time of execution. Mirroring of the top border is performed when the processing stage has received

$$s_k = max(h_k^I - t_k, t_k + 1) \tag{2}$$

lines. Mirror copies of the line buffers are done by reference to minimize performance overhead. The *max* operation is required because $s_k$ depends largely on the configuration of the image processing algorithm ( $h_k^I, t_k, b_k$ ). In some cases, mirroring could be done even earlier, but $s_k$ ensures that the execution of the image processing algorithm can be started right after mirroring. Depending on the image processing algorithm, it might be possible to execute the algorithm multiple times after mirroring has been performed. The number of executions on the top border $m_k^{top}$ can be calculated as

$$m_k^{top} = max(1, 2t_k + 1 - h_k^I + 1) = max(1, 2t_k - h_k^I + 2) \,. \tag{3}$$

Figure 3.5a visualizes the mirroring on the top border, when $h_k^I = 5$ , $t_k$ = 1 and $b_k$ = 3. According to (1), $c_k^I$ = 8. In this case, the image processing algorithm can be run only once.

Mirroring on the bottom border, on the other hand, can be performed immediately when the last line of the input image has been received and placed to the input line buffer array at index $h_k^I - 1$ . This is illustrated in Figure 3.5b with the same configuration as in Figure 3.5a. The height of the input image is denoted by *h*. The image processing algorithm can be executed

$$m_k^{bottom} = b_k + 1 \tag{4}$$

times.

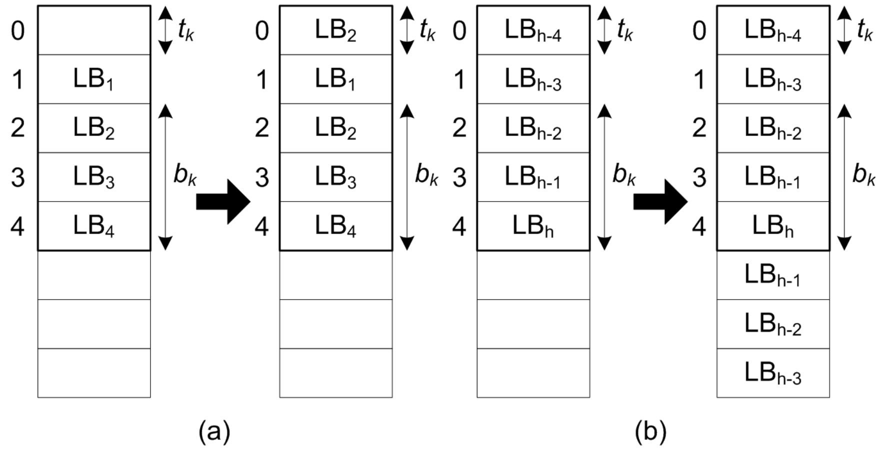



***Figure 3.5:*** *Mirroring with a configuration* $h_k^I$ *= 5,* $t_k$ *= 1 and* $b_k$ *= 3. (a) On the top border. (b) On the bottom border.*

The mirrored lines have two reference copies in the input line buffer array. Hence, a mirrored line cannot be released back to the memory manager until both references of it

are useless. In the software framework, release of the oldest input line buffer at index 0 can be started when the processing stage has received $f_k$ input lines which is defined by

$$f_k = s_k + t_k - (m_k^{top} - 1) . \quad (5)$$

At this point, there are no mirror copies left in the input line buffer array. Releasing the oldest input line can be continued normally until the mirroring of the bottom border is performed. After mirroring, some lines can still be released when the image processing algorithm is executed $m_k^{bottom}$ times. The execution counter $p_k$ is first set to $p_k = m_k^{bottom}$ and is decremented by one after each execution. The oldest input line can be released as long as

$$p_k > 2 b_k - h_k^I + 2 . \quad (6)$$

When (6) changes to false, the oldest input line is left unreleased. Processing is continued until $p_k = 0$ , after which all remaining input lines are released at once.

## 3.4 Pipeline Configurations

The algorithms used by a real image reconstruction pipeline place certain restrictions on the pipeline configuration. Statistics-dependent algorithms such as tone reproduction [8] assume by default that the image statistics is calculated in between the pipeline. However, in the designed pipeline, image data would enter the next processing stage before the statistics would have been collected for the whole image. Therefore, statistics are not available at that point of time and cannot be used to control the processing. Even simple histogram operations are not possible in the middle of the pipeline since histogram calculation requires that the whole image is analyzed before the analysis results can be used.

There are at least three alternative solutions to enable the use of statistics in the pipeline. The first solution is to split the pipeline before each statistical algorithm and place a full intermediate image buffer between the pipelines to store the results from the previous pipeline. An example of such a split pipeline is illustrated in Figure 3.6. Statistics are collected at the end of the previous pipeline and are then used at the beginning of the next pipeline. Since the next pipeline will not be started before the previous pipeline has been fully executed, the statistics are accurate and, therefore, usable immediately in the next pipeline.

Even if multiple splits are needed, it may be that only one intermediate image buffer is required since the software framework allows the same image buffer to be used as both input and output. This is applicable only if the pipeline does not introduce any scaling to the image, which is usually the case. When scaling is not used, the number of produced output lines never exceeds the number of already processed input lines, which makes it possible to use the same image buffer for both input and output. However, if

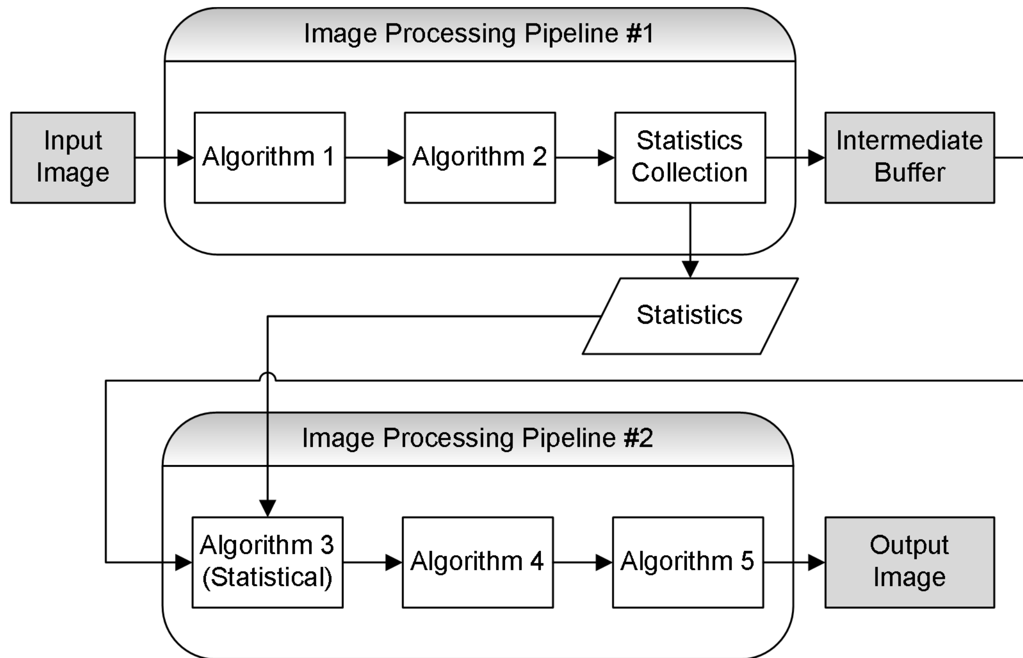



***Figure 3.6:*** *Enabling the use of statistical algorithms through a split pipeline and intermediate buffers.*

the input image format uses a smaller line size than the output image format, input image data must be padded to match the size of the output image lines to prevent unprocessed input lines from being overwritten by the new output lines written by the pipeline. Padding of input data is illustrated in Figure 3.7.

The major drawback in splitting the pipeline is the increased memory usage due to the use of full intermediate image buffers. Even in the best case, one full image buffer is needed which makes the memory savings of the software framework not so remarkable. Thus, in memory-bound environments, the use of a split pipeline with intermediate buffers is not a viable option.

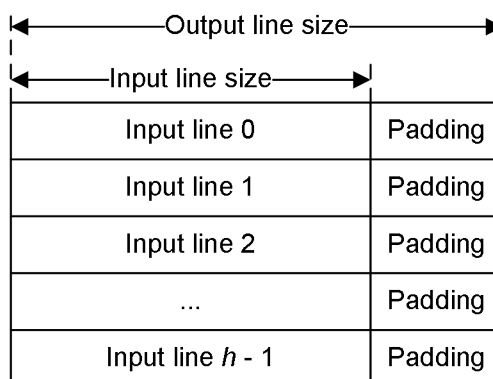



***Figure 3.7:*** *Padding input data to allow the use of a single image buffer for both input and output.*

The second solution to enable statistics in the pipeline is a variation of the split pipeline where the use of intermediate buffers is avoided. The principle of this approach is to first run a preprocessing pipeline which only collects statistics and does not produce any output image. The actual image reconstruction pipeline then runs the full pipeline with all algorithms and uses the statistics collected from the preprocessing pipeline. Figure 3.8 depicts this configuration.

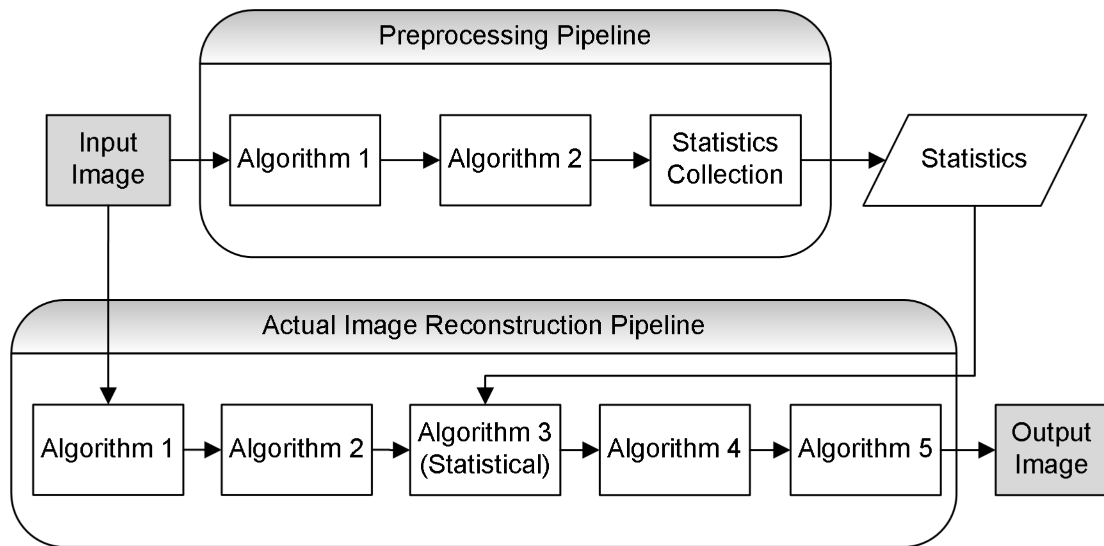



***Figure 3.8:*** *Eliminating intermediate buffers by duplicating a part of the image reconstruction pipeline.*

The drawback of the bufferless split pipeline is that the preprocessing pipeline must execute all algorithms of the full pipeline up to the point where the statistics would be collected. Since the intermediate results are not stored anywhere, these algorithms must be re-run in the actual image reconstruction pipeline, which means longer execution time of the total pipeline. The performance overhead is not too large if the statistics collection is in the early stages of the pipeline. However, if the statistics collection would be in the later stages of the pipeline, the overhead of a separate preprocessing pipeline would become too high and would not be usable, since mobile devices are not only memory-bound but also limited by their processing power, so there is not much room for extra processing.

Split pipeline configurations produce 100% accurate results with a sacrifice of either memory usage or performance. However, a better compromise for mobile devices may be achieved by approximating statistics. Consequently, the third solution relies on pre-computed statistics calculated from the latest viewfinder image. Since the viewfinder of the mobile phone is updated in real-time when the user is in camera mode, the last viewfinder image before taking a photo is approximately equal to the actual image that the user takes. Although the viewfinder image typically has a lower resolution, the

requested statistics can be calculated from it accurately enough. This way, the image reconstruction pipeline need not to be split since the statistics are already available for use in the image processing algorithms. The viewfinder-based pipeline is illustrated in Figure 3.9.

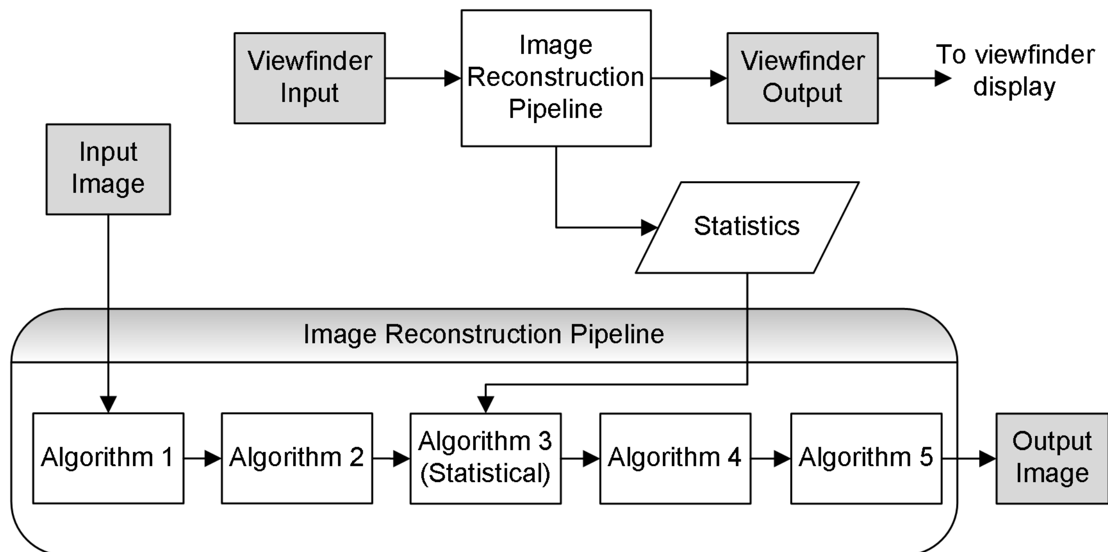



***Figure 3.9:*** *An image reconstruction pipeline using statistics calculated from the latest viewfinder image.*

The viewfinder image acquired from the camera sensor must be always run through the image reconstruction pipeline in order to be able to show it on the viewfinder display. Therefore, there is no extra performance overhead of this step in the pipeline. The statistics calculation can be performed while processing the viewfinder input image since the results are not used in the same pipeline. The memory overhead is also avoided due to the absence of intermediate buffers. Thus, the viewfinder-based solution is recommended for the mobile devices. According to subjective comparison, it does not result in any observed loss in image quality despite approximate statistics. This is the preferred way of using the software framework with statistical image processing.

# 4 LINE BUFFER MEMORY MANAGEMENT

A simple way to manage line buffer memory is to dynamically allocate them from the heap when needed and deallocate them after use. However, dynamic memory allocation is slow and it fragments the memory [18]. Memory fragmentation introduces even more overhead and may finally result in memory allocation fails due to absence of continuous memory area that is large enough to hold the requested block. Memory fragmentation is a serious issue in mobile devices, in which a small physical memory is particularly prone to fragmentation.

The software framework uses memory pools to avoid performance penalties and memory fragmentation. A memory pool is a large pre-allocated chunk of memory that is split into smaller constant-sized blocks to allow fast allocations at runtime [18]. When an application requests a memory block, it is retrieved from the pool without any new allocations from the heap. When the memory block is no longer needed, it is released back into the pool instead of being deallocated and freed to the heap. Thus, there is no performance overhead of acquiring and releasing memory blocks. Since the large chunks of memory are allocated only once, memory fragmentation is avoided. Memory pools also improve the spatial coherence of the data, which reduces the amount of data cache misses, thus, enhancing performance further.

An exemplar structure of the implemented memory manager is illustrated in Figure 4.1. Since an image reconstruction pipeline may need line buffers of multiple sizes, a unique memory pool is reserved per applied line buffer size. The memory pools are stored in a singly-linked list to allow any number of pools to be easily managed by the memory manager with a minimal memory overhead.

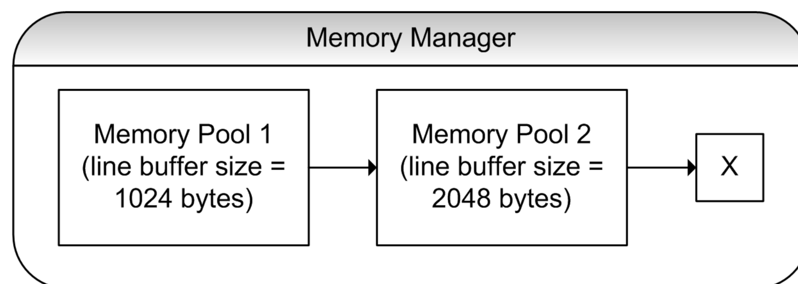



***Figure 4.1:*** *Memory manager and the memory pools.*

The memory pools are created in the run-time configuration phase after the line buffer sizes of processing stages have been defined. The pipeline configuration determines the total size of each memory pool. An upper boundary for the total size of the memory pool x can be determined by

$$mem_x = W_x \sum_{k=0}^{K-1} \left( h_k^I(w_k^I) + h_k^O(w_k^O) \right) \text{, where}$$
$$h_k^I(w_k^I) = \begin{cases} h_k^I, \text{if } w_k^I = W_x \\ 0, \text{otherwise} \end{cases} \text{ and } h_k^O(w_k^O) = \begin{cases} h_k^O, \text{if } w_k^O = W_x \\ 0, \text{otherwise} \end{cases} . \quad (7)$$

In (7), $W_x$ is the size of the line buffers in the memory pool *x*, $w_k^I$ is the size of the input line buffers for the *k*:th algorithm and $w_k^O$ is the size of the output line buffers for the *k*:th algorithm.

Memory pools are allocated independently as large memory chunks which are then split into line buffers of the correct size. At the beginning, all line buffers are available, so they are marked as free. Each memory pool manages its free line buffers in a singly-linked list. Usually, each linked list node is dynamically allocated from the heap, but this would practically cancel out the advantages of the memory pools. Instead, as in [18], the linked list is constructed by reusing the memory of the free memory blocks—in our case, the memory of the free line buffers. When a line buffer is unused, the first four bytes of it are applied to store the pointer to the next line buffer in the linked list. This way, the linked list can be constructed from the free line buffers themselves without allocating any explicit linked list nodes. Figure 4.2 depicts the utilized linked list structure at two occasions.

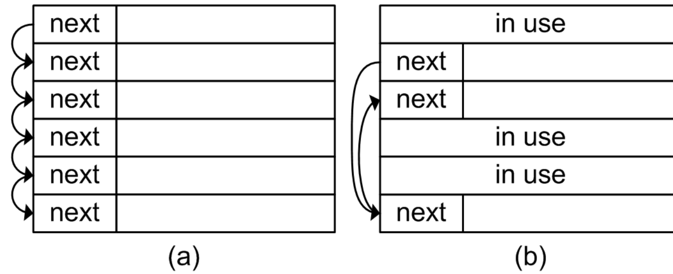



***Figure 4.2:*** *Linked list inside a memory pool. (a) At the beginning. (b) After being used for a while.*

The implemented memory management makes the use of line buffers simple and efficient. When acquiring a line buffer, the memory pool holding the line buffers of the requested size is searched from the linked list of memory pools. This is an O(*n*) operation, where *n* denotes the number of memory pools. Since *n* is usually small, the cost of a linear search through the linked list is negligible. After the right memory pool has been found, the first free line buffer is taken from the head of the linked list of free

line buffers. This is an O(1) operation. When a line buffer is released, the right memory pool is again searched from the linked list of memory pools. In the chosen memory pool, the line buffer is inserted into the head of the linked list of free line buffers in O(1) time.

## 4.1 Reducing Memory Footprint

The maximum line buffer usage inside each processing stage is reached only when that particular stage is processing data. At that point, the other processing stages may not require the full number of input and output lines, and thus, some of the line buffers can be shared between processing stages that use the line buffers of the same size. This way, the memory consumption of the software framework can be further optimized.

Figure 4.3 visualizes the state of a three-stage pipeline at three consecutive occasions when each image processing algorithm (IPA) uses a $3\times3$ filter kernel. To illustrate the

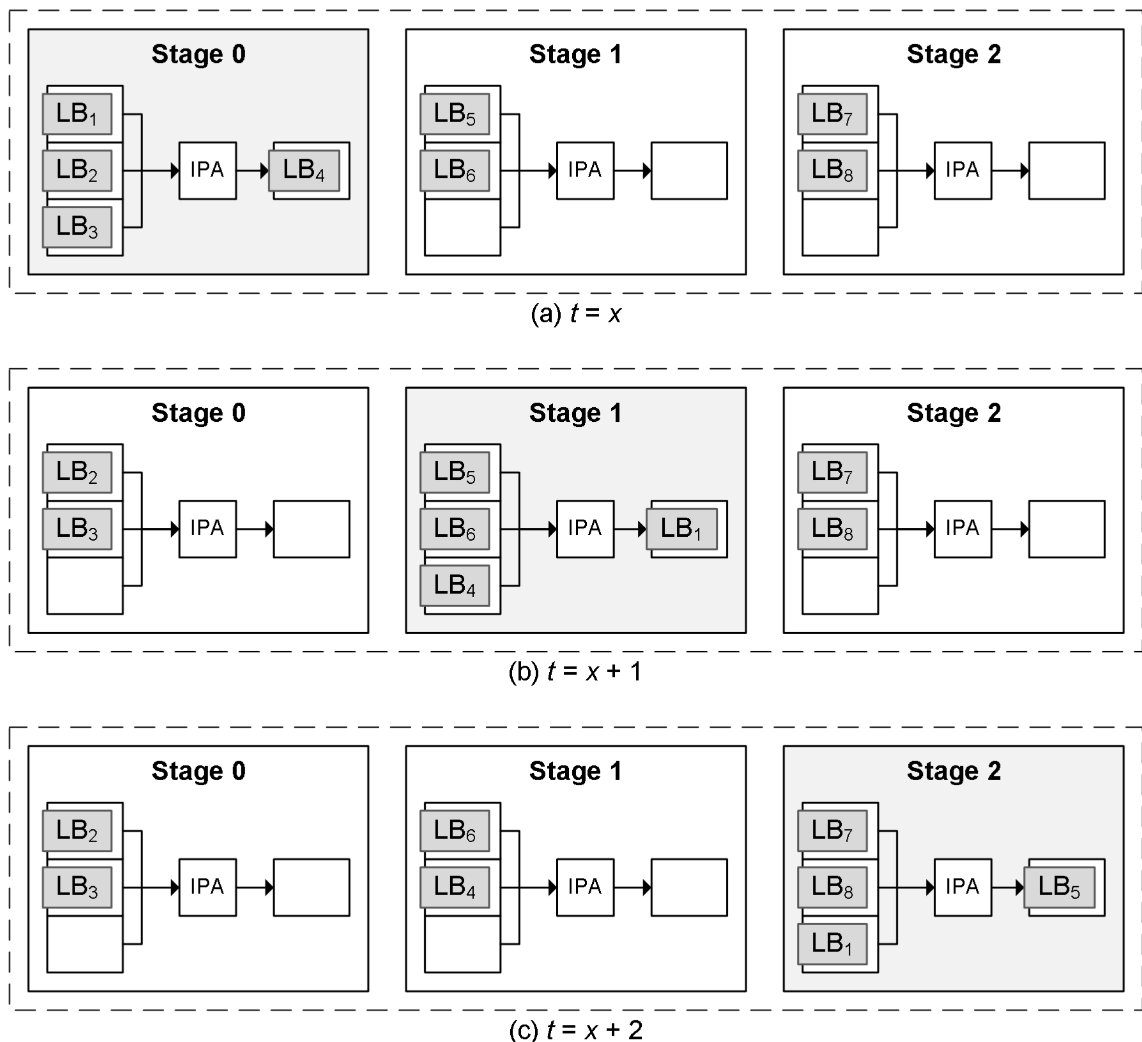



***Figure 4.3:*** *(a) Stage 0 is processing. (b) The output from stage 0 is fed to stage 1 and processing begins. (c) The output from stage 1 is fed to stage 2 and processing begins.*

line buffer consumption, the pipeline uses the same image format in every stage so that the same line buffers can be circulated among all three stages.

When the processing stage *k* runs its image processing algorithm, it has a full set of input lines $h_k^I$ and it allocates a full number of output line buffers $h_k^O$. Thus, the memory usage peaks at that point. After the output line buffers have been filled by the algorithm, the oldest input line buffer is immediately released. This drops the input line consumption to $h_k^I - 1$. The first output line buffer is then fed to the next processing stage *k* + 1. The ownership of the line buffer is transferred and the output line consumption of the processing stage *k* drops to $h_k^O - 1$. The same happens in every consecutive processing stage. Thus, the image reconstruction pipeline is always in a state where the currently active stage *i* requires $h_i^I$ input line buffers and $h_i^O$ output line buffers, and the rest of the processing stages $j \neq i$ require only $h_j^I - 1$ input line buffers and $h_j^O - 1$ output line buffers. This opens up the possibility for allocating even less line buffers during run-time configuration.

Mirroring causes some special cases where the pattern is violated and memory optimization is not possible. When the processing stage *k* is set to execute a filtering operation ( $i_k > 1$ ) and either $t_k = 0$ or $b_k = 0$, the oldest input line buffer cannot be immediately freed during mirroring. In that case, the input line consumption remains in $h_k^I$ even though the stage becomes inactive, preventing any memory optimizations. The output line consumption, on the other hand, can always be optimized regardless of the image processing algorithm and its parameters.

The memory optimization is possible within each memory pool *x*. The tight upper boundary for the total size of the memory pool *x* can be formalized as

$$mem_x = W_x \left( \sum_{k=0}^{K-1} \left( m_k^I + m_k^O \right) + min \left( 2, \sum_{k=0}^{K-1} \left( e_k^I + e_k^O \right) \right) \right), \text{ where}$$

$$m_k^I = \begin{cases} h_k^I, \text{if } w_k^I = W_x \wedge i_k > 1 \wedge (t_k = 0 \vee b_k = 0) \\ h_k^I - 1, \text{if } w_k^I = W_x \\ 0, \text{otherwise} \end{cases}, \quad m_k^O = \begin{cases} h_k^O - 1, \text{if } h_k^O > 0 \wedge w_k^O = W_x \\ 0, \text{otherwise} \end{cases}, \tag{8}$$

$$e_k^I = \begin{cases} 0, \text{if } w_k^I = W_x \wedge i_k > 1 \wedge (t_k = 0 \vee b_k = 0) \\ 1, \text{if } w_k^I = W_x \\ 0, \text{otherwise} \end{cases}, \text{ and } \quad e_k^O = \begin{cases} 1, \text{if } h_k^O > 0 \wedge w_k^O = W_x \\ 0, \text{otherwise} \end{cases}.$$

In (8), $m_k^I$ represents the required number of input line buffers for the *k*:th algorithm at any instant and $e_k^I$ represents the number of extra input line buffers needed when the *k*:th algorithm is processing data. Respectively, $m_k^O$ and $e_k^O$ represent similar requirements for the output line buffers. At maximum, two extra line buffers per each size $W_x$ are needed regardless of the number of stages. The extra line buffers are shared by all stages using the same line buffer size.

The memory savings are highest when all algorithms in the pipeline use the same line buffer size and lowest when they all need different buffer sizes. In the worst case, this optimization provides no additional gain. However, at least a part of the algorithms use the same image formats in a typical image reconstruction pipeline [3], so the introduced memory optimization is useful and well justified.

## 4.2 Cache Associativity Optimization

Memory access pattern affects cache performance significantly when dealing with data-intensive programs. Due to cache associativity, power-of-two image widths tend to conflict with the cache line size of the processor [19] and can, thus, decrease the performance of the software framework.

As an example, a $9 \times 9$ averaging blur for an 8-bit grayscale image of size $2048 \times 1536$ is considered. The input and output line buffers in this case are 2048 bytes wide. For each output pixel, the averaging blur operation needs to access all nine line buffers to read the $9 \times 9$ neighborhood of that particular pixel. When, for example, the first byte is read from all nine line buffers, the same cache line set is being hit. If the processor is assumed to have an eight-way set-associative cache with a cache line size of 64 bytes, a cache miss occurs every time the ninth line buffer is being read since the set only has eight cache line slots. This makes the cache act against itself and is often referred as cache trashing.

The best solution to fix the critical stride issue is to use padding [19]. The software framework optimizes the line buffer sizes so that when a power-of-two line buffer size is requested, it pads it with four extra bytes. For example, in case of a 2048 byte line buffer, the software framework would use a 2052 byte line buffer instead. This prevents the cache from trashing itself and reduces the number of cache misses significantly with power-of-two images.

The cache associativity optimization concerns only the special case of power-of-two images and, thus, does not enhance the performance of the pipeline with non-power-of-two images. If the image width is not a power of two, there is no performance benefit of padding the line buffers with extra bytes. The image width in digital photography usually is not a power of two. However, some embedded systems may rely on power-of-two images and the optimization is useful for those cases.

# 5 PERFORMANCE ANALYSIS

The software framework is compared with a traditional image reconstruction pipeline which uses the ping-pong buffer scheme. The memory consumption of the software framework can be divided into three parts: the system input buffer (lines streamed from the camera sensor), the line buffers within the pipeline, and the system output buffer. Hence, the total amount of line buffer memory ($mem_{total}$) is

$$mem_{total} = h_{-1}^{I} w_{-1}^{I} + \sum_{k=0}^{K-1} \left( h_k^I w_k^I + h_k^O w_k^O \right) + h_K^O w_K^O \,, \tag{9}$$

where $h_{-1}^{I}$ is the number of lines in the system input buffer, $w_{-1}^{I} = w_0^I$ is the size of the lines in the system input buffer, $h_K^O$ is the number of lines in the system output buffer, and $w_K^O = w_{K-1}^O$ is the size of the lines in the system output buffer.

## 5.1 Basic Experiments

All the basic experiments were accomplished with an image reconstruction pipeline [3] adopted from digital cameras. Table 5.1 tabulates the selected algorithm configurations and their reference implementations in the literature.

***Table 5.1:*** *Algorithm configurations in the evaluated image reconstruction pipeline.*

| Algorithm | $h_k^I$ | $B_k^I$ | $format_k^I$ | $h_k^O$ | $B_k^O$ | $format_k^O$ |
|---|---|---|---|---|---|---|
| Auto Level [3] | 1 | 2 | 16-bit RAW Bayer | 1 | 2 | 16-bit RAW Bayer |
| Auto White Balance [3] | 1 | 2 | 16-bit RAW Bayer | 1 | 2 | 16-bit RAW Bayer |
| CFA Interpolation (hue-based, 1st pass) [7] | 3 | 2 | 16-bit RAW Bayer | 1 | 3 | 24-bit RGB |
| CFA Interpolation (hue-based, 2nd pass) [7] | 3 | 3 | 24-bit RGB | 1 | 3 | 24-bit RGB |
| Edge Detection (Sobel) [6] | 3 | 3 | 24-bit RGB | 1 | 4 | 24-bit RGB + 8-bit edge |
| Noise Filter [20] | 9 | 4 | 24-bit RGB + 8-bit edge | 1 | 4 | 24-bit RGB + 8-bit edge |
| Color Correction [3] | 1 | 4 | 24-bit RGB + 8-bit edge | 1 | 4 | 24-bit RGB + 8-bit edge |
| Linear sRGB to CIELAB | 1 | 4 | 24-bit RGB + 8-bit edge | 1 | 7 | 48-bit CIELAB + 8-bit edge |
| Edge & Color Saturation Enhancement [20] | 3 | 7 | 48-bit CIELAB + 8-bit edge | 1 | 6 | 48-bit CIELAB |
| CIELAB to Linear sRGB | 1 | 6 | 48-bit CIELAB | 1 | 3 | 24-bit RGB |
| Tone Reproduction [21] | 1 | 3 | 24-bit RGB | 1 | 3 | 24-bit RGB |
| Gamma Correction | 1 | 3 | 24-bit RGB | 1 | 3 | 24-bit RGB |
| sRGB to YcbCr | 1 | 3 | 24-bit RGB | 1 | 3 | YCbCr 4:4:4 |

The used pipeline configuration is illustrated in Figure 5.1. A hybrid approach was used for statistics-dependent algorithms. First a preprocessing pipeline was run to calculate the histogram for auto-level and to extract macro blocks for auto white balance. Then the actual image reconstruction pipeline was run with the rest of the algorithms using statistics collected from the latest viewfinder image. The use of a preprocessing pipeline is justified by the fact that auto-level and AWB are the first two algorithms in the pipeline. To have a fair comparison between the software framework and the ping-pong buffer scheme, both approaches were set to use calculated statistics for auto-level and AWB and statistics collected from the viewfinder image for the rest of the algorithms.

In Table 5.1, $format_k^I$ and $format_k^O$ specify the pixel formats for the input and output images, respectively. It was assumed that the camera sensor produces 16-bit RAW Bayer data and the pipeline uses interleaved YCbCr 4:4:4, 48-bit CIELAB and

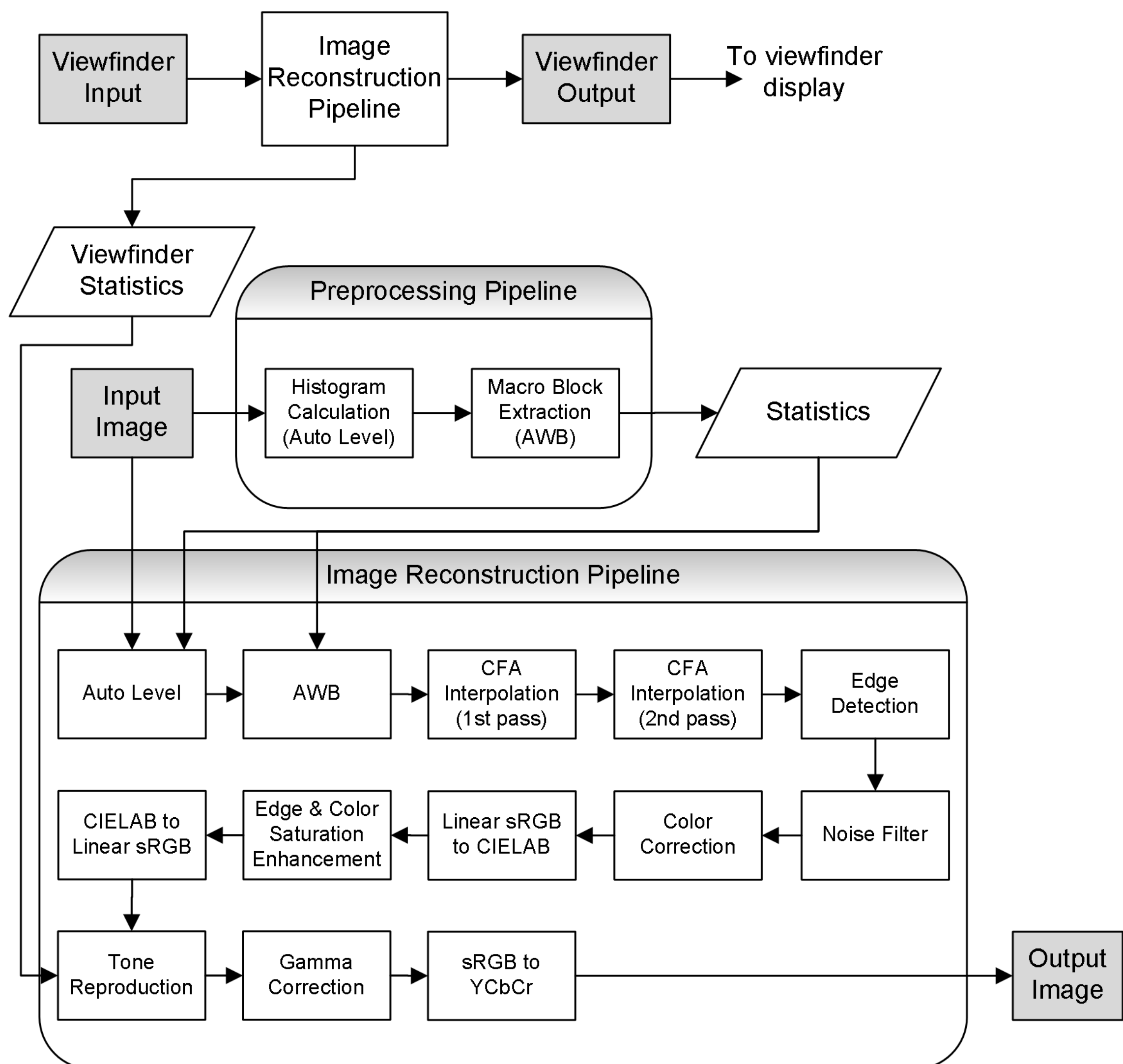



***Figure 5.1:*** *The pipeline configuration used for the basic experiments.*

24-bit RGB formats. 48-bit CIELAB was used instead of the 24-bit counterpart to retain better image quality between color conversions. Some of the formats were also appended with 8-bit edge information so that the edge map from the edge detection phase could be passed along the pipeline to all algorithms that require it. The number of bytes required per pixel for input and output on each stage, $B_k^I$ and $B_k^O$, are determined by the pixel formats. With this information, the line buffer sizes for an input image having a width $w$ can be calculated as $w_k^I = w B_k^I$ and $w_k^O = w B_k^O$.

The software framework and the test program were implemented in C++ and compiled with GCC 4.4.3 using –O2 optimizations on a 32-bit Windows environment. The details of the test environment are listed in Table 5.2. The algorithms were implemented identically for both the software framework and the ping-pong buffer scheme, so that the influence of the algorithms is completely excluded from the results. The software framework was configured to use a system output buffer which could hold eight lines of the output image at a time. Feeding the input lines, on the other hand, was done in groups of 16 lines, so that the possible performance impact of the draining feature would be visible in the test results.

***Table 5.2:*** *The test environment.*

| | |
|---|---|
| **Processor** | Intel Core 2 Duo E8400 (2 x 3.0 GHz) |
| **L1 Caches** | 2x32 kB instruction caches; 64-byte cache lines<br>2x32 kB data caches; 64-byte cache lines |
| **L2 Cache** | 6 MB shared 24-way set-associative; 64-byte cache lines |
| **Main Memory (RAM)** | 4096 MB |
| **Operating System** | Windows XP Professional 32-bit |

All tests were run using a 6-megapixel camera resolution ( $3088 \times 2048$ ). Since the ping-pong buffer scheme uses two full-sized image buffers, its memory usage is constant regardless of the image processing algorithms. The size of the required image buffers in ping-pong depends on the largest pixel format, which in this case is the 48-bit CIELAB with edge information (7 bytes per pixel). Thus, with the applied $3088 \times 2048$ image resolution, the ping-pong buffer scheme consumes 84.44 MB of memory. The memory usage of the implemented framework, on the other hand, is dependent on the image processing algorithms. Table 5.3 tabulates the absolute memory usage of the implemented framework and the respective percentage compared with the ping-pong scheme. Furthermore, the memory usage of the whole pipeline is analyzed on the algorithm-level to show how the memory distributes within the pipeline.

According to the tests, the software framework saves 99.3% of memory compared with the ping-pong pipeline when the whole pipeline is concerned without advanced memory optimizations. The memory consumptions of the individual algorithms are all less than 0.15%. The absolute memory consumption of the whole pipeline, 636 kB, fits well for memory-constrained mobile devices.

***Table 5.3:*** *Memory usage and performance results.*

| Algorithm | Memory Usage (kB) | Memory Usage (%) | Execution Time (+/-%) |
|---|---|---|---|
| Auto Level | 12 | 0.01% | -4.4% |
| Auto White Balance | 12 | 0.01% | -2.6% |
| CFA Interpolation | 63 | 0.07% | -0.1% |
| Edge Detection | 39 | 0.05% | -6.3% |
| Noise Filter | 121 | 0.14% | -3.7% |
| Color Correction | 24 | 0.03% | -11.6% |
| Linear sRGB to CIELAB | 33 | 0.04% | +1.4% |
| Edge & Color Saturation Enhancement | 81 | 0.09% | -0.8% |
| CIELAB to Linear sRGB | 27 | 0.03% | -1.8% |
| Tone Reproduction | 18 | 0.02% | -0.6% |
| Gamma Correction | 18 | 0.02% | -3.5% |
| sRGB to YCbCr | 18 | 0.02% | -0.2% |
| System input buffer | 97 | 0.11% | - |
| System output buffer | 72 | 0.08% | - |
| **Whole pipeline** | **636** | **0.74%** | **-4.2%** |
| **Whole pipeline (memory-optimized)** | **327** | **0.38%** | **-4.2%** |

If the advanced memory optimizations are enabled, the memory consumption drops further down to 327 kB, offering 48.6% savings compared to the unoptimized version. In overall, the memory-optimized software framework saves 99.6% of memory compared with the ping-pong pipeline.

If the designed pipeline would be split before tone reproduction to enable statistical calculations, a full-sized intermediate buffer having 24-bit RGB data would be needed. This would add 18.09 MB to the total memory consumption for our approach. However, the memory consumption would still be only 22.2% compared to that of the ping-pong pipeline. Thus, the framework saves 77.8% of memory even with a split pipeline using intermediate buffers. A bufferless split pipeline would not be feasible because tone reproduction is the third last algorithm in the pipeline resulting in a too large performance loss of running the pipeline twice up to that point.

Table 5.3 also reports the relative execution time of the implemented framework over the ping-pong scheme both in the case of individual algorithms and the whole pipeline. The execution times for the individual algorithms were measured by running the algorithms separately, that is, without any other stages in the pipeline. The execution times for both approaches were calculated as an average of processing the same image ten times.

The software framework was better than the ping-pong pipeline with 11 out of 12 individual algorithms. The difference in execution time varied from -11.6% to +1.4%.

When the whole pipeline was executed, the software framework consumed 4.2% less time compared with the ping-pong pipeline. The performance improvement of the framework is due to the fact that the line buffer memory, being significantly smaller than the full-sized image buffers in the ping-pong pipeline, has better cache locality. During execution, the line buffers stay better in the cache, which decreases the amount of data cache misses. This was verified by analyzing both pipelines with Intel VTune Amplifier XE 2011. According to VTune, the software framework caused 16.1% less data cache misses than the ping-pong pipeline when the whole pipeline was executed. Especially the L2 cache misses were reduced by 89.8%.

## 5.2 Evaluating the Effects of Cache Associativity Optimization

Cache associativity optimization was made to improve the running time performance of the software framework in cases where the image width is a power of two. To evaluate the effects of the cache associativity optimization, another test program was implemented to benchmark the pipeline with and without the optimization. The test program used a $9 \times 9$ averaging blur as a test algorithm which was run repeatedly by varying the number of processing stages. An image with a $2048 \times 1536$ resolution was used as the input for the test pipeline. The cache associativity optimization benchmarks were run on the same environment as the basic experiments. The test results are illustrated in Figure 5.2.

The results show that the cache associativity optimization improves the running time of the software framework significantly. The total running time of the test pipeline

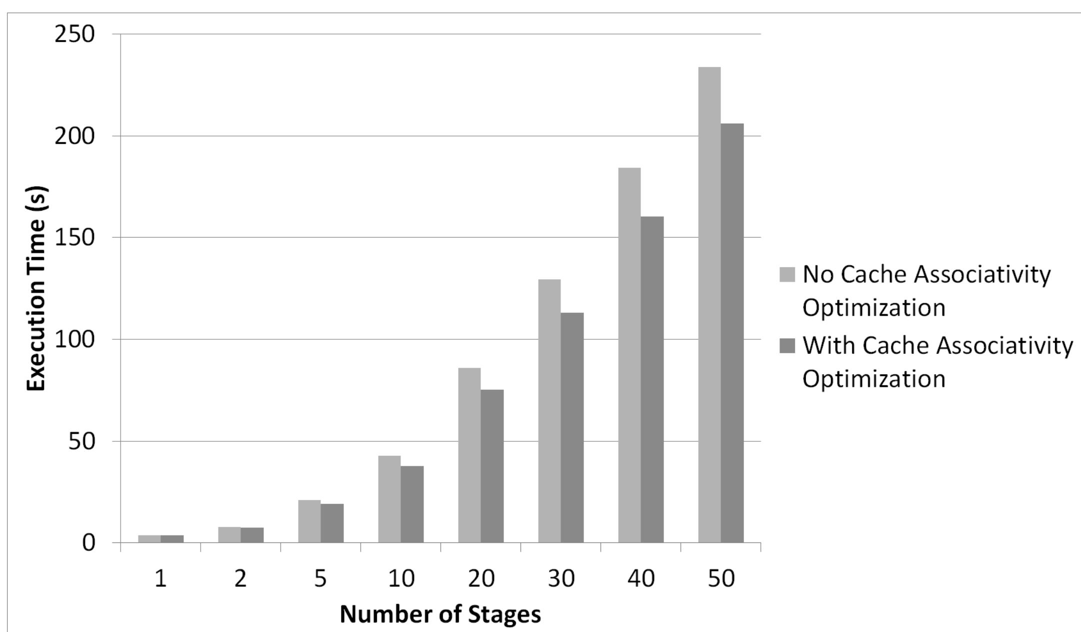



***Figure 5.2:*** *Results from the cache associativity benchmarks.*

decreased by 9.17% on average when taking all test cases into account. With only one stage, the unoptimized pipeline was slightly faster than the cache-optimized pipeline. With two or more stages, the cache-optimized pipeline performed always better than the unoptimized one. The improvement grew as the number of stages was increased until a peak improvement of 12.95% was reached with 40 stages. Since the optimization increases the memory usage only by 0.2%, the performance improvement really outweighs the memory overhead.

# 6 PARALLELIZING THE PIPELINE

A software-based image reconstruction pipeline has many advantages over a hardware-based one, but it may not attain adequate performance when high-resolution images need to be processed within strict time requirements. As the processing power of the mobile phones continues to grow, it is more and more cost-effective to build the image reconstruction pipeline in software. Recently, many smart phones have been equipped with a multi-core processor for enhanced performance. Therefore, it is beneficial to study whether the line-buffer-based image reconstruction pipeline could be parallelized to take advantage of the new multiprocessing opportunities to improve performance.

In order to take advantage of the multi-core processors in the newest smart phones, the image reconstruction pipeline should utilize parallelism on the program level. Two prospective alternatives for implementing program-level parallelism in an image reconstruction pipeline are *data-level parallelism* [22] and *task-level parallelism* [23]. Data-level parallelism focuses on distributing the data to multiple processing nodes which all execute the same task. In task-level parallelism, the computation problem is divided into multiple distinct tasks that each are executed on separate processing nodes. The processing nodes communicate with each other while they execute.

Task-level parallelism could be implemented by running each image processing algorithm of the pipeline on a different processing node. However, since images contain a very large amount of pixel data that goes through the same pipeline, data-level parallelism is a better mechanism because it is easier to implement and offers better scalability than task-level parallelism. Data-level parallelism can be achieved in the image reconstruction pipeline at least by two ways: slice-based parallelization and work-queue-based parallelization.

## 6.1 Slice-Based Parallelization

Slice-based parallelization is considered the simplest way to distribute pipeline execution to multiple processing nodes. The input image is divided horizontally into smaller input slices that are fetched to individual pipelines that each perform all image processing algorithms for that slice. After processing, the output slices are merged back together to form the final image. Each individual slice is processed on a separate thread by a separate pipeline to achieve data parallelism. The image slices should be of roughly equal size so that the workload is split evenly for all threads. Otherwise, the thread with the largest slice would limit the performance of the whole pipeline. Figure 6.1 illustrates

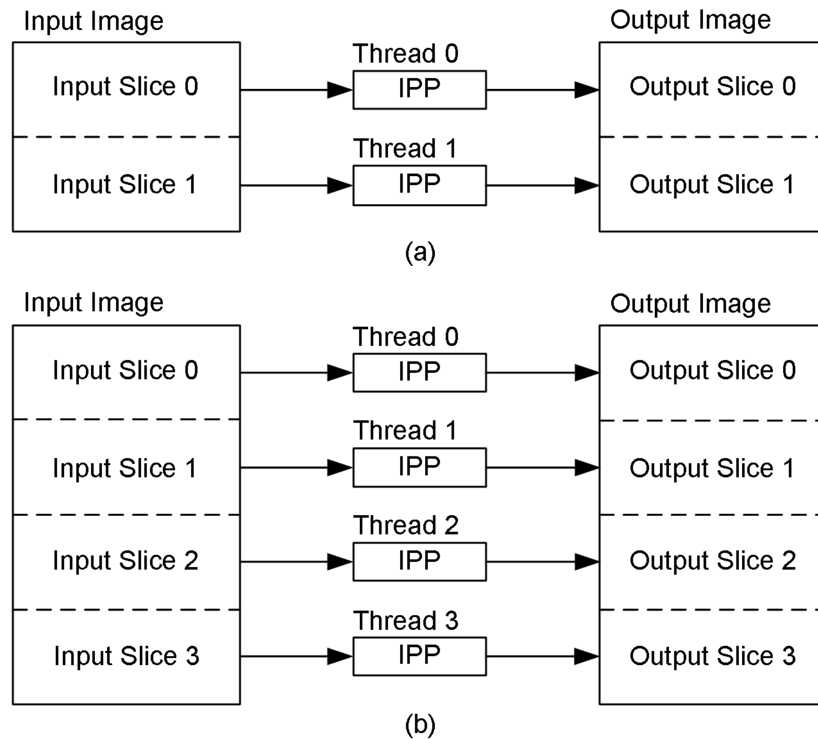



***Figure 6.1:*** *Slice-based parallelization using (a) two threads, (b) four threads.*

the slice-based pipeline with a dual-core and a quad-core processor. Scalability is achieved with the fact that a large image can be easily split into multiple slices, whereas the number of image processing algorithms would limit the division to tasks if task-level parallelism was used.

The reason for using only horizontal stripes is that vertical stripes would complicate line-buffer-based processing and merging significantly. Retaining the original line length keeps the parallelized pipeline easier to manage since there is no need to split line buffers into smaller ones, which actually would require knowing the exact byte representation of the input image lines. Merging is also kept simple because the produced output slices can be appended with each other to form the final image without having to deal with non-contiguous memory areas.

Slice-based parallelization is easy to implement but it has several disadvantages. Slicing the input image must be done in a way that ensures that each image processing algorithm receives enough input data so that mirroring will not be used around the stripes, since it would result in incorrect processing causing visible artifacts in the output image. Thus, the slices overlap slightly and the lines around the stripes are fed to both pipelines that share that particular slice edge. The amount of overlap depends on how large filter kernels are used in the image processing algorithms. Overlapping causes the same edge lines to be processed by two different threads, so slice-based parallelization processes data unnecessarily multiple times. With a small number of slices, the overhead is marginal, but it increases as the number of slices grows. For

example, doubling the number of slices from four to eight increases the overhead by 133%.

Another disadvantage is that the memory consumption of the slice-based pipeline is significantly larger than that of the original single-threaded pipeline because of two reasons. Firstly, the memory consumption is proportional to the number of threads used, since each thread uses its own pipeline instance. For example, with four threads, the memory consumption is quadrupled compared to the original single-threaded pipeline. Secondly, a full-sized output buffer is required so that each threaded pipeline can write to it without any synchronization issues. Due to these requirements, the memory consumption of the pipeline will be over 50% compared to the ping-pong pipeline. The improvement over the ping-pong pipeline is still significant, but not compared to the original single-threaded pipeline.

Since the processing overhead and memory consumption increase as a function of processing nodes, slice-based parallelization is practical for a small number of cores only. Thus, it is not suitable for scenarios where the available number of processor cores grows significantly.

## 6.2 Work-Queue-Based Parallelization

Work-queue-based scheme is a more complex parallelization approach than the slice-based parallelization. The principle in the work-queue-based parallelization is to divide processing into small tasks so that each task addresses one input line of the original input image. The tasks must be executed in order, but their execution can overlap with certain restrictions.

Each task is responsible of executing the whole image reconstruction pipeline until the next input line is required. In a normal scenario, this means that the execution of the pipeline goes from the first processing stage up to the last processing stage before the task completes. Parallelism is achieved with a pattern which resembles pipelining used in processors, as illustrated in Figure 6.2. When the first task is executing the fourth processing stage, the second task can be executing the third processing stage, the third task executing the second processing stage and so on.

The tasks must be synchronized so that the execution of a newer task cannot pass the execution of an older one in the pipeline. Especially, the same processing stage must not be executed simultaneously by two different threads. This requirement is caused by the fact that the image processing algorithms may in many cases require that they are run in a data-successive order. An example of such an algorithm is geometrical distortion correction. The ability to have multiple concurrent executions of the same image processing algorithm would also require a thread-safe implementation for the actual algorithm, if it relied on collecting any information to shared variables. In order to not

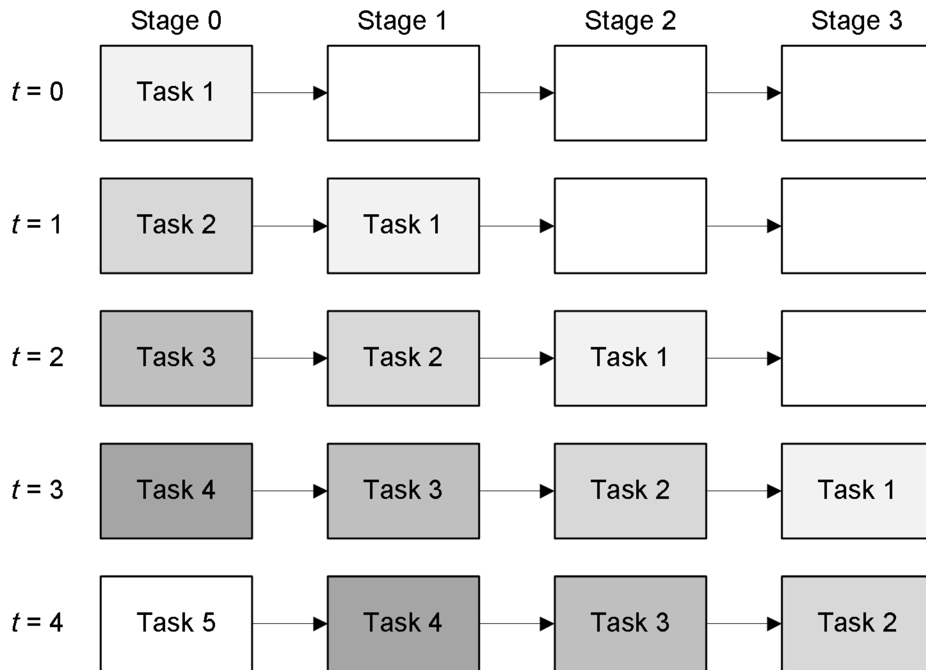



***Figure 6.2:*** *Pipelining pattern in the work-queue-based parallel pipeline.*

complicate the algorithm implementations, the order of execution for each algorithm must remain the same as in the single-threaded pipeline.

The structure of the work-queue-based pipeline is illustrated in Figure 6.3. The system uses a thread pool design pattern [24] where a number of worker threads is created to execute the tasks that are available in a shared work queue. The work queue stores all waiting tasks created by the image reconstruction pipeline. Each thread in the thread pool requests a task from the work queue with a first-in-first-out (FIFO) principle and executes that task inside the pipeline. During the execution of the task, the image reconstruction pipeline may spawn new tasks that are added to the work queue. When the thread completes the task, it requests the next task from the work queue and continues so on until the work queue is empty.

The biggest advantage of the work-queue-based approach is that it is truly scalable. The number of threads in the thread pool can be chosen freely since there are much more tasks than there can be threads in any reasonable case. It is usually set to match the number of processing nodes to take advantage of all of the available parallel processing power. The best performance is achieved this way because less threads would not utilize all the processing power whereas more threads would create overhead from thread scheduling. Also, increasing the number of threads up to the number of processing nodes does not create any duplicate processing or memory overhead that was inevitable with slice-based parallelization.

However, the work-queue-based approach has also its downsides. Managing the execution of the image reconstruction pipeline without access conflicts requires a lot of

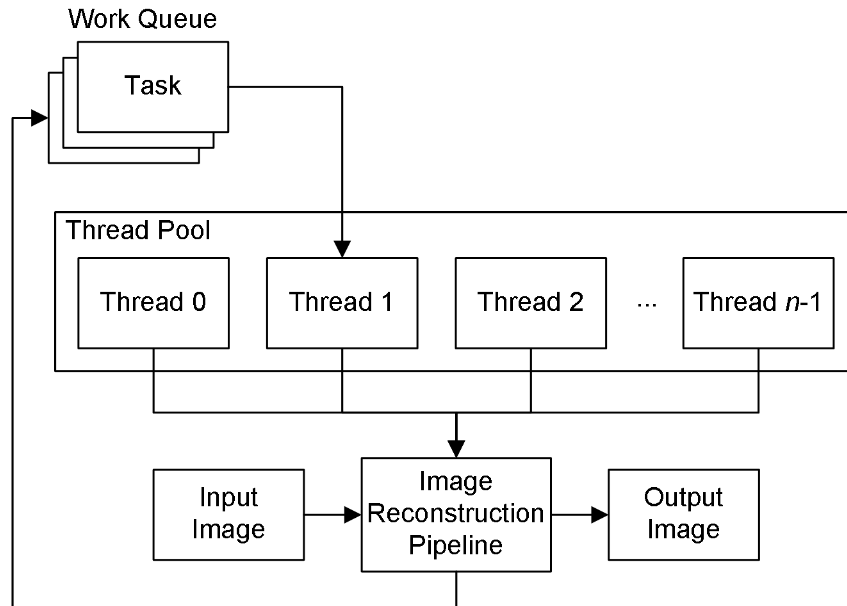



***Figure 6.3:*** *Work-queue-based pipeline uses a thread pool to drive the execution of the image reconstruction pipeline.*

synchronization which creates overhead. Each processing stage must synchronize access to its state variables so that the operation is thread-safe. Also, threads may have to block the execution of their current tasks in certain cases if an older task is not completed early enough. Since the tasks are always requested for execution in the correct order, it is usually the case that the older task is ready earlier than the next one, but operating system scheduling may introduce variations in the running time so that blocking is indeed needed. The work queue requires also a thread-safe implementation so that all worker threads can access the work queue in a safely manner. The overhead from the work-queue synchronization is marginal. However, it is hard to predict without an implementation whether the processing stage synchronization overhead will become too dominant since it depends on the operating system and the realized execution pattern.

The used task division also causes other synchronization issues. The execution of a task must be halted if an older task needs to go through the pipeline multiple times. This scenario is possible when an image processing algorithm in the middle of the pipeline can be run multiple times, for example, due to the top or bottom border being processed. The same applies to algorithms that produce more than one output line on one execution of the image processing algorithm. The execution of the halted task can be continued as soon as the correct order of executing the processing stage can be ensured. This means that several threads may have to block their execution due to one task that requires larger amount of processing.

One possible way of overcoming the halting issue is to allow the large task to be broken down into multiple sub-tasks as soon as the need for multiple execution is detected [25]. The thread executing the main task would continue executing the first

sub-task whereas the other sub-tasks would be added to the work queue. The work queue implementation would also have to be changed from a FIFO to a priority queue to maintain the correct order of both tasks and the new sub-tasks. In addition, this approach would require that the execution of tasks can be suspended so that if a newer task has already been taken into execution, it is interrupted and the thread moves on to execute the more important sub-task instead. This kind of suspension scheme requires the ability to store the state of each task, which in turn makes the pipeline much more complex.

Another disadvantage of the work-queue-based pipeline is that the advanced memory optimizations can no longer be used. Since multiple processing stages are being executed simultaneously, the pipeline must prepare for the worst case where all processing stages require the full number of input and output line buffers at the same time. Therefore, the memory consumption of the work-queue-based approach follows the equation (7). Nevertheless, the memory consumption does not grow as the number of threads is increased, so it is much better than in the case of slice-based parallelization.

# 7 CONCLUSION

This Thesis presented a novel software framework for the management of a line-buffer-based image reconstruction pipeline. The software framework reduces memory consumption drastically without any performance compromises. According to our experiments, the software framework requires 99.6% less memory than the traditional ping-pong pipeline in a realistic image reconstruction pipeline. The framework also offers better performance than the ping-pong pipeline, but is still generic enough to support a wide variety of image processing algorithms. Therefore, the framework is well suited for a wide range of resource-constrained devices that need image processing. Easy configurability also guarantees that the software framework decreases the time spent in the management of the image reconstruction pipeline. It is estimated that the development time of image reconstruction pipeline, excluding algorithm development, can be reduced by 90%.

This Thesis also considered two potential parallelization approaches that could speed up the performance of the implemented image reconstruction pipeline in mobile multi-core processors. Slice-based parallelization would be easy to implement but it suffers from the limited scalability and increased memory consumption. Better scalability makes the work-queue-based scheme more promising parallelization approach but its synchronization overhead and implementation complexity may restrict its usage in practice.

In the future, the software framework will be redesigned to conform to these parallelization approaches in order to see their actual performance gain and memory usage over the original single-threaded version. If the work-queue-based pipeline overcomes its synchronization and complexity concerns, it would be a prominent competitor to the hardware-based image reconstruction pipelines in the forthcoming mobile phones with numerous processor cores.

# REFERENCES

[1] Bellas, N., Yanof, A. An Image Processing Pipeline with Digital Compensation of Low Cost Optics for Mobile Telephony. Proceedings of the 2006 IEEE International Conference on Multimedia and Expo, Toronto, ON, Canada, July 9–12, 2006. USA 2006, IEEE. pp. 1249–1252.

[2] SMIA 1.0: Introduction and Overview [WWW]. Nokia Corporation, ST Microelectronics NV. 2004. [Cited 6/5/2013]. Available at: http://read.pudn.com/downloads95/doc/project/382834/SMIA/SMIA_Introduction_and_overview_1.0.pdf

[3] 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.

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

[5] Nikkanen, J., Gerasimow, T. & Kong, L. Subjective effects of white-balancing errors in digital photography. Optical Engineering 47(2008)11, pp. 1133201-1–113201-15.

[6] Hornberg, A. Handbook of Machine Vision. Germany 2006, Wiley-VCH. 821 p.

[7] Ramanath, R., Snyder, W. E. & Bilbro, G. L. Demosaicing methods for Bayer color arrays. Journal of Electronic Imaging 11(2002)3, pp. 306–315.

[8] Kao, W.-C., Chen, L.-Y. & Wang, S.-H. Tone Reproduction in Color Imaging Systems by Histogram Equalization of Macro Edges. IEEE Transactions on Consumer Electronics 52(2006)2, pp. 682–688.

[9] Chung, S.-W., Kim, B.-K., Song, W.-J. Detecting and Eliminating Chromatic Aberration in Digital Images. Proceedings of the IEEE International Conference on Image Processing (ICIP 2009), Cairo, Egypt, November 7–12, 2009. USA 2009, IEEE. pp. 3905–3908.

[10] Zheng, Y., Lin, S., Kambhamettu, C., Yu, J., Kang, S. B. Single-Image Vignetting Correction. IEEE Transactions on Pattern Analysis and Machine Intelligence 31(2009)12, pp. 2243–2256.

[11] Montabone, S. Beginning Digital Image Processing: Using Free Tools for Photographers. USA 2010, Apress. 312 p.

[12] SMIA 1.0 Part 2: CCP2 Specification [WWW]. Nokia Corporation, ST Microelectronics NV. 2004. [Cited 6/5/2013]. Available at: http://www.sunex.com/SIMA/SMIA_CCP2_specification_1.0.pdf

[13] Gregory, J. Game Engine Architecture. USA 2009, A K Peters.

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

[15] Gonzalez, R. C. & Woods, R. E. Digital Image Processing. USA 2008, Prentice Hall. 976 p.

[16] Gruen, H. & Story, J. Resolve Your Resolves [WWW]. USA 2008, UBM Tech. [Cited 6/5/2013]. Available at: http://www.gamasutra.com/view/feature/131995/resolve_your_resolves.php

[17] Pat. US20090273686. Methods, computer program products and apparatus providing improved image capturing. Nokia Corporation. (Kaikumaa, T., Kalervo, O., Ilmoniemi, M., Bodex, R., Yong, S.-H. & Baxter, A.). Application number 12/150,966, 2.5.2008. (5.11.2009).

[18] Llopis, N. C++ For Game Programmers. USA 2003, Charles River Media. 412 p.

[19] Preisz, E. & Garney, B. Video Game Optimization. USA 2010, Course Technology PTR. 368 p.

[20] Kao, W.-C., Chen, Y.-J. Multistage Bilateral Noise Filtering and Edge Detection for Color Image Enhancement. IEEE Transactions on Consumer Electronics 51(2005)4, pp. 1346–1351.

[21] Zhang, H. & Lucchese, L. A fast tone reproduction algorithm for high dynamic range image display. 2004 IEEE 6th Workshop on Multimedia Signal Processing, Siena, Italy, September 29–October 1, 2004. USA 2004, IEEE. pp. 275–278.

[22] Baumstark, L. Jr. & Wills, L. Exposing Data-Level Parallelism in Sequential Image Processing Algorithms. Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE 2002), Richmond, VA, USA, October 29–November 1, 2002. USA 2002, IEEE. pp. 245–254.

[23] Hung, L. D. & Sakai, S. Dynamic Estimation of Task Level Parallelism with Operating System Support. Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN 2005), Las Vegas, NV, USA, December 7–9, 2005. USA 2005, IEEE.

[24] Syer, M. D., Adams, B. & Hassan, A. E. Identifying Performance Deviations in Thread Pools. Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011), Williamsburg, VA, USA, September 25–30, 2011. USA 2011, IEEE. pp. 83–92.

[25] Gajski, D. D., Abdi, S., Gerstlauer, A. & Schirner, G. Embedded System Design: Modeling, Synthesis and Verification. USA 2009, Springer. 384 p.