Darwin College Research Report

DCRR-011

GPU-Based Raw Digital Photo Manipulation

Sarah J. Fortune

June 2010
(electronic edition 8 November 2013)

Darwin College
Cambridge University
United Kingdom CB3 9EU
www.dar.cam.ac.uk/dcrr

ISSN 1749-9194

Abstract

Digital camera use an array of single colour sensors arranged in Bayer pattern. One colour component is captured at each pixel in the array. The missing colours in the raw sensor data are interpolated by a process called demosaicing. Demosaicing is one of the first steps in the image processing pipeline. This makes a high quality demosaicing algorithm especially important, as interpolation failures can cause highly visible artifacts which are worsened by further processing steps.

Current demosaicing methods are computationally expensive and often too slow for real-time applications. I propose two methods to improve the performance of Adaptive Homogeneity-Directed demosaicing, the most widely used demosaicing algorithm and one of the most effective.

The first method is to take advantage of parallelisation provided by the GPU, using the NVIDIA CUDA platform. I describe how the algorithm was adapted for best performance on the GPU architecture and evaluate the performance gains.

I also present a new demosaicing technique which improves the performance of current methods while maintaining image quality. It takes advantage of the fact that demosaicing artifacts occur along horizontal and vertical lines, but that these features occupy only a small area of natural images. A quick first pass interpolation is applied to the sensor data. The result is used to identify regions in the image where artifacts are likely to occur. A high quality but computationally expensive demosaicing method is applied to these areas. The performance and effectiveness of this method is evaluated and compared to existing techniques.

Acknowledgements

I wish to express my gratitude to Dr Harle for supervising this project and for his support and encouragement over the year. I also wish thank Michael Gallagher for his care and attention.

"To photograph truthfully and effectively is to see beneath the surfaces and record the qualities of nature and humanity which live or are latent in all things."

Ansel Adams

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital cameras capture images in 'raw' format which is the equivalent of a digital negative. Raw images contain the original data captured by the sensors before any further processing or compression has been applied to the image. They may be automatically converted into JPEG images by the camera. However, this causes an information loss that is unacceptable for professional photographers. The colour depth of raw images is 12 or 14 bits, while JPEG limits the colour depth to 8 bits. Additionally, JPEG is an inherently lossy format and introduces compression artifacts.

Using raw format also gives photographers control over the white balance or colour temperature. White balancing is a process of colour correction used to compensate for different lighting sources. For example, photographs taken in overcast daylight and those taken under artificial lighting will have different temperatures. Automatic white balancing can be performed by the camera, but the method is dependent on the camera model and may not be effective in all situations. However, once white balancing is performed the original colour data cannot be recovered. For this reason, photographers will use raw format when they need greater control over the process or where the method employed by the camera is inadequate.

The raw image does not contain full RGB colour data. Single chip cameras,

which make up the majority of digital cameras, use a single sensor overlaid with a colour filter array, CFA. This effectively produces a grid of single colour sensors. As a result each pixel in the image data contains only one colour value. The CFA typically uses a checker-board arrangement known as the Bayer pattern[Bay76], 2.1. The Bayer pattern contains twice as many green sensors as red or blue sensors.

The raw image is 'developed' into an RGB image by a process called demosaicing. This involves estimating the missing values using the sensor data available. The most effective demosaicing methods also use assumptions about the properties of natural images. There is a huge selection of demosaicing algorithms, using all kinds of techniques from linear interpolation to pattern matching[Cok94, CCP99] to artificial neural networks[KHO00].

It turns out that demosaicing is quite a difficult problem and basic approaches produce unacceptable artifacts. A common artifact is 'zippering', a checker-board pattern which appears along horizontal and vertical edges. This effect can be seen in Figure 2.2 on page 7. It caused by the diagonal alignment of colours in the Bayer pattern.

The most effective demosaicing method in widespread use today is Adaptive Homogeneity-directed Demosaicing[HP05], AHD. It is a sophisticated method which attempts to minimise artifacts while maintaining sharp edges in the result. The standard implementation of AHD is the open source tool 'dcraw'[1]. Dcraw decodes proprietary raw image formats and supports several demosaicing methods. It is used internally by many other image processing tools — the author's homepage lists 55 projects currently using dcraw.

However, the despite the advantage of raw images, there are some drawbacks. Processing raw images can be very computationally expensive. A raw image is typically an order of magnitude larger than the corresponding JPEG file. The best demosaicing algorithms require extensive computation time. Demosaicing an average sized raw image using AHD takes around ten seconds. This makes it unusable for real-time applications such as editing raw images

[1] Available from http://www.cybercom.net/~dcoffin/dcraw/

in an interactive application. Also, when demosaicing is performed on the camera, long computation times can restrict the capture rate.

The aim of this project was to improve the performance of demosaicing. Currently, applicantions are limited by the performance of the demosaicing methods. Applications which update raw images in real-time are forced to use less effective demosaicing routines, reducing the user experience. High quality processing of raw image sets is usually performed as a batch process due to the long execution times.

The goal is to improve the performance of AHD demosaicing to the point where it can be used in real-time applications. This would enable photographers editing raw images to view their changes immediately. It would allow for greater experimentation as more versions of the image could be produced in a shorter amount of time.

I propose two methods for improving the performance of AHD demosaicing. The first method is to take advantage of the GPU hardware found in most consumer PCs. GPUs provide a platform for low cost highly-parallel computation. They are designed for high performance streaming computation and are optimised for floating point operations and two dimensional memory access, which is ideal for this application. However, until recently they were not widely used for general purpose computing. I plan to take advantage of the NVIDIA CUDA platform which allows the GPU to be programmed in a C-like language.

I have implemented a parallel version of AHD demosaicing for NVIDIA GPUs and a sequential version in C for comparison. I have found the GPU version achieves a 30x speed-up compared to the C version. The processing time for an average sized image is under one second, making it much more practical for real-time applications.

The second method I propose for improving performance is a new demosaicing algorithm called Mask Demosaicing. It is intended to provide high quality results, comparable to the results of AHD, but with a shorter execution time. It achieves the performance improvement by identifying areas in the image

that are prone to artifacts and applying a high quality demosaicing algorithm, in this case AHD, to these areas. A fast demosaicing method is applied to the remaining areas. Two different methods of identifying the problematic areas are proposed and evaluated, the first is based on edge detection, the second on artifact detection.

Chapter 2

Background and Related Work

Camera Design

Digital cameras contain a CCD sensor which detects light intensity. However, it cannot directly detect colour or the wavelength of light. Colour information can be captured by placing a colour filter array, CFA, over the sensor. This gives an image with one colour value per pixel, the 'raw' image. The most common CFA pattern is is the Bayer Matrix, invented by Bruce Bayer of Eastman Kodak[Bay76]. This is an alternating pattern of red and green rows, and blue and green rows. There are twice as many green filters as red or blue because the human eye is more sensitive to green. The most popular Bayer pattern is GRBG, Figure 2.1 on page 6. It has been proved that this is the most optimum arrangement for a three colour filter in order to reduce aliasing[ASH05]. Another CFA pattern is the CYGM filter which uses cyan, yellow, green and magenta filters. The CYGM pattern is more sensitive than an RGB pattern, so it gives more accurate luminance information but at the expense of reduced colour accuracy.

It possible to build three colour sensors which can detect red, green and blue values at each pixel. The 3CCD sensor[Woo05] uses a prism to split the light into separate colour components. These are captured using three different

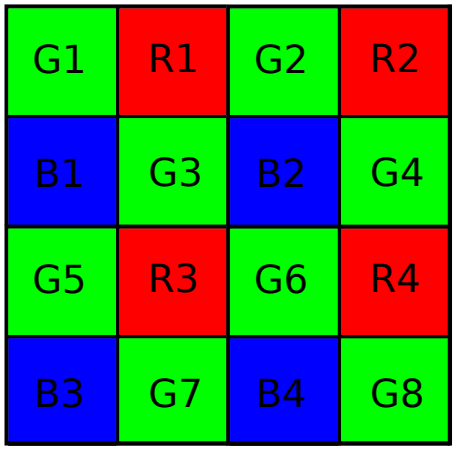


Figure 2.1: The most widely used Bayer pattern[Bay76]. It is a repeating pattern of the 2x2 GRBG tile. Variations of the Bayer pattern based on rotations and translations of the basic tile also exist.

CCDs. Alternatively, the Foveon X3 sensor uses three vertically stacked photo-diodes[LH02]. Each photo-diode captures a different wavelength of light. However, both of these solutions are more expensive than a simple single chip camera using one CCD and a colour filter array. The sensor is one of the most expensive camera components, accounting for 10% to 25% of the total cost[APS98].

Demosaicing Methods

An image captured using a CFA is missing two colour values at each pixel. The missing values are estimated by a process called demosaicing. This section will discuss various approaches to demosaicing. The biggest challenge is avoiding 'zippering' artifacts, which are checker-board artifacts that appears along horizontal and vertical edges, 2.2.

Figure 2.2: Image demosaiced with bilinear interpolation. It shows zippering artifacts along horizontal and vertical lines, highlighted in the insets.

Bilinear Interpolation

The most basic demosaicing algorithm is bilinear interpolation. The missing values are interpolated by averaging colour values available at the eight neighbouring pixels. Missing green values at position i, j are interpolated using the formula:

$$G'_{i,j} = \frac{G_{i-1,j} + G_{i+1,j} + G_{i,j-1} + G_{i,j+1}}{4} \tag{2.1}$$

The interpolation of missing red values depends on their position with respect to the Bayer array.

Where j is a red and green row, the missing red pixels are interpolated by:

$$R'_{i,j} = \frac{R_{i-1,j} + R_{i+1,j}}{2} \tag{2.2}$$

Where j is a blue and green row, and i is a green pixel, the red value is:

$$R'_{i,j} = \frac{R_{i,j-1} + R_{i,j+1}}{2} \tag{2.3}$$

Where j is a blue and green row, and i is a blue pixel, the red value is:

$$R'_{i,j} = \frac{R_{i-i,j-1} + R_{i+1,j-1}R_{i-i,j-1} + R_{i+1,j+1}}{4} \tag{2.4}$$

Blues values are interpolated using the same method as red values.

Bilinear is a simple and fast method of interpolation. It can be implemented easily and economically in hardware[FZY09, GLAAWV08]. It is a good solution where speed is more important than quality, for example generating previews in interactive programs. However, it produces noticeable zippering, blurring and moiré artifacts, as seen in Figure 2.2 on page 7.

Constant Hue Based Interpolation

Bilinear interpolation treats each colour channel separately. However, in natural images the colour channels are highly correlated. In RGB images the cross-correlation has been found to be 0.86 for red/green, 0.79 for red/blue, and 0.92 for green/blue[KTK98]. Cok[Cok87] proposed a method which takes advantage of this cross-correlation. It has since formed the basis of several other demosaicing techniques[jr., Kim99, Wel89, Fre87, LP94, Hib95]. It is based on the assumption that the colour ratio is constant within an object and changes smoothly along edges. The aim is to try to avoid artifacts, which are characterised by abrupt changes in colour. This approach is based on a technique from computer vision, where a scene is assumed to represent a "Mondrian world", a single planar surface of Lambertian or matte surfaces[AGLM93]. Light reflected from these planar objects is of uniform intensity.

Cok defines hue as the ratio of chrominance and luminance. The human eye is more sensitive to luminance than chrominance, so the higher sampled green channel is taken to represent luminance. The red and blue channels represent chrominance components. Therefore, hue is defined as the vector $(R/G, B/G)$.

If X and Y are neighbouring coordinates in the image, assuming a constant hue gives the following equality:

$$\frac{R_Y}{G_Y} = \frac{R_X}{G_X} \tag{2.5}$$

This can be rearranged to give an equation to interpolate R_Y using the interpolated green value at Y and the interpolated hue of the neighbouring pixels.

$$R'_Y = G'_Y(\frac{R_X}{G_X})' \tag{2.6}$$

The blue channel can be interpolated in a similar way. The green values can be interpolated using either bilinear interpolation or a more complicated method such as pattern matching. The pattern matching[Cok94] method uses a template to recognise geometric features like edges, stripes and flat areas, and then applies an appropriate interpolation method for that feature.

Constant hue based interpolation improved on the linear interpolation in use at the time[DLK78] and it was widely used commercially[RSBS02]. However, the constant hue assumption does not always hold, and the method can break down on textured areas and sharp edges, see Figure 2.3 on page 11. It has since been superseded by more effective modern methods, but its techniques still form the basis of many demosaicing algorithms.

Edge Directed Interpolation

Zippering artifacts occur on horizontal and vertical edges. A straightforward means of avoiding this issue is to detect edges in the Bayer image, and interpolate along direction of the edge which avoids zippering. There have been various approaches to edge directed interpolation[KB02, Hib95, PTAB10, LP94]. The method proposed by Hibbard[Hib95] calculates the horizontal and vertical gradients in the green channel, which contains twice as much information as the red or blue channels. If the difference between the horizontal and vertical gradients is above a certain threshold, the larger gradient determines the direction of interpolation. Otherwise, it is assumed that the gradients refer to either a flat area or a highly textured noisy area, in which case the algorithm falls back to bilinear interpolation.

Edge directed interpolation has been extended to exploit the correlation between colour channels by Laroche and Prescott[LP94]. Their method detects gradients in the red and blue channels. The green channel is interpolated according to the direction of the gradient. Red and blue channels are interpolated using bilinear interpolation.

Hamilton and Adams[Ada97] used second order gradients to detect edges.

Figure 2.3: Demosaicing using Cok's constant hue assumption[Cok87]. This method shows a slight improvement over bilinear interpolation. Zippering is still visible, but the magnitude of error in colour is reduced.

Horizontal and vertical edges in the green channel are detected by the presence of high frequencies in the green and chroma (red or blue) channel. Edges are detected using the laplacian of the green channel and the gradient of the chroma channel:

$$H_{i,j} = |G_{i-1,j} - G_{i+1,j}| + |2R_{i,j} - R_{i-2,j} - R_{i+2,j}| \tag{2.7}$$

$$V_{i,j} = |G_{i,j-1} - G_{i,j+1}| + |2R_{i,j} - R_{i,j-2} - R_{i,j+2}| \tag{2.8}$$

The larger gradient value determines the direction of interpolation. The green channel is interpolated by averaging the neighbouring green pixels and using a 'correction term' from the chroma channel. The correction term acts as an averaging filter which reduces aliasing. If the horizontal gradient, $H_{i,j}$, is larger the vertical gradient, $V_{i,j}$, then the green channel is interpolated using the horizontal method:

$$G'_{i,j} = \frac{G_{i,j-1} + G_{i,j+1}}{2} + \frac{2R_{i,j} - R_{i,j-2} - R_{i,j+2}}{4} \tag{2.9}$$

Otherwise, the following vertical method is used:

$$G'_{i,j} = \frac{G_{i-1,j} + G_{i+1,j}}{2} + \frac{2R_{i,j} - R_{i-2,j} - R_{i+2,j}}{4} \tag{2.10}$$

A similar method is used to interpolate the red and blue channels, except diagonal edges are detected instead of horizontal and vertical.

Edge directed interpolation avoids the blurring caused by bilinear interpolation and gives sharper results along horizontal and vertical edges. However it has the potential to cause misguidance artifacts. These artifacts occur when the algorithm estimates the edge in the wrong direction. This can cause breaks to appear in straight lines, as well maze-like artifacts, as seen in Figure 2.4 on page 13. This kind of artifact is particularly noticeable as the human eye is sensitive to straight lines.

Figure 2.4: Image demosaiced using Hamilton and Adams edge directed interpolation[JJ97]. Their method produces fewer zippering artifacts than constant hue based interpolation. However, mazing artifacts are still visible.

Adaptive Homogeneity Directed Demosaicing

This section describes the Adaptive Homogeneity-Directed demosaicing method, AHD, proposed by Hirakawa and Parks[HP05]. AHD is currently one of the most effective demosaicing algorithms[GGAS05] and is widely used in open source demosaicing software. It improves on the method proposed by Hamilton and Adams[JJ97] by determining the interpolation direction using the 'homogeneity' of the neighbourhood instead of by explicitly detecting edges. Hirakawa also adds an extra artifact removal stage by applying a median filter to the interpolated image.

AHD Algorithm steps

Interpolation Horizontal and vertical interpolation is performed over the entire image, giving two interpolated images H' and V'. The green channel is interpolated using the same method as Hamilton and Adams, formulae (2.9) and (2.10). However, Hirakawa and Parks arrived at this method independently. Starting from the constant-hue assumption, they assume that the difference between the green and chroma (red and blue) channels varies slowly. They designed a low-pass filter that would minimise the difference between the green and chroma channels, $G-R$ and $G-B$. The discrete form of the filter gives the same interpolation method as Hamilton and Adams.

The red and blue channels are interpolated without regard to the direction, the same method is used for both the horizontal and vertical images. They make the assumption that $R-G$ is slowly varying, at a rate lower the sampling frequency. Then $R-G$ can be reconstructed from the surround pixels by applying a low pass filter, L, to the sampled R interpolated G values:

$$(R-G)' = L * (R-G') \tag{2.11}$$

Rearranging this gives the method for interpolating the red channel:

$$R'_{i,j} = G'_{i,j} + \frac{(R_{i\pm1,j\pm1} - G'_{i\pm1,j\pm1})}{4} \tag{2.12}$$

Figure 2.5: Image produced using Adaptive Homogeneity-Directed demosaicing[HP05], AHD. There is no zippering and the misguidance and mazing artifacts are almost entirely absent.

Figure 2.6: The horizontal and vertically interpolated images, H' and V'. Each version gives the best results where the edges lie in the same direction as the interpolation. The homogeneity metric will be used to determine the most appropriate direction for the different parts of the image.

This formula applies to the case where i, j is at a blue pixel in the Bayer pattern and sampled red values are available at the diagonals. A similar method is applied to the other positions in the Bayer pattern.

Homogeneity Map The next step is to calculate the homogeneity metric for each pixel in the horizontal and vertical images. The metric will be used to decide which interpolation direction should be used for the final result. It measures how similar the pixel is to the region surrounding it. Artifacts are caused by high variations in colour and contrast, so a region containing artifacts will have a lower homogeneity metric.

Hirakawa defines the homogeneity of a coordinate as the number of points

within a neighbourhood that have similar colour. The homogeneity is calculated in the CIELAB colour space. CIELAB is designed to reflect the sensitivity of the human eye. The distance between colours in CIELAB colour space corresponds more accurately to perceived difference than in the RGB colour space[WS00]. The conversion method is described in Appendix A.

The homogeneity metric is based on the luminance distance d_L, and the chrominance distance d_{ab}, which are defined as the euclidean distances:

$$d_L(x, y) = |x_L - y_L| \tag{2.13}$$

$$d_{ab}(x, y) = \sqrt{(x_a - y_a)^2 + (x_b - y_b)^2} \tag{2.14}$$

where x and y are CIELAB colours.

The homogeneity metric of a coordinate is the number of pixels within a 5x5 neighbourhood where the luminance and chrominance distances are below the thresholds ϵ_L and ϵ_{ab} respectively. The thresholds are determined dynamically according to the content of the region. In practise, a 3x3 neighbourhood can be used instead, in order to reduce the amount of computation needed, in exchange for a slight sacrifice in image quality.

Separate horizontal and vertical thresholds ϵ_{LH} and ϵ_{LV} are determined from the horizontal and vertically interpolated images. The threshold ϵ_L is the minimum of these two values as it corresponds to the more uniform and less artifacted direction. For each interpolation direction, the threshold is defined to be the larger distance between the pixel in question and its immediate neighbours in the interpolation direction. So, for the horizontal image it is the maximum of the distances between the left and right neighbours.

$$\epsilon_{LH}(i, j) = \max(d_L(H'_{i,j}, H'_{i,j-1}), d_L(H'_{i,j}, H'_{i,j+1}) \tag{2.15}$$

$$\epsilon_{LV}(i, j) = \max(d_L(V'_{i,j}, V'_{i-1,j}), d_L(V'_{i,j}, V'_{i+1,j}) \tag{2.16}$$

(a) Horizontal Homogeneity (b) Vertical Homogeneity (c) Direction Selection

Figure 2.7: The horizontal and vertical homogeneity maps. Brighter areas represent higher homogeneity, which indicates fewer artifacts. Figure (c) shows the directions chosen for the final image (black is horizontal, white is vertical). The decision was made based on which direction had the higher average homogeneity over a small neighbourhood.

$$\epsilon_L(i,j) = \min(\epsilon_{LH}(i,j), \epsilon_{LV}(i,j)) \tag{2.17}$$

The ϵ_{ab} threshold is obtained in a similar manner.

Direction selection The homogeneity metric is used to determine which interpolation direction should be used for the result. For each pixel, the direction with the lowest average homogeneity over a 3x3 neighbourhood is selected. Using the spatial average of the homogeneity metric avoids discontinuities in straight lines and reduces in the influence of noise in the image.

Artifact Reduction The final step in the algorithm is artifact removal. There may be some artifacts remaining in the image after direction selection, but these can be removed by applying a noise reduction filter. Hirakawa and Parks use a median filter, which removes noise while preserving edges. The filter is applied to the difference between the channels, instead of to the

channels directly, as the difference changes at a lower frequency than the channel itself. They recommend applying the filter three times for optimal results.

$$R = \text{median}(R - G) + G \tag{2.18}$$

$$B = \text{median}(B - G) + G \tag{2.19}$$

$$G = \frac{\text{median}(G - R) + \text{median}(G - R) + B + R}{2} \tag{2.20}$$

AHD Shortcomings

AHD is one of the most effective and popular demosaicing algorithms. However, its high quality comes at the expense of long processing times. Run times vary between five and twenty seconds per photo, depending on the resolution and processor speed. This makes AHD unsuitable for interactive applications and makes batch processing of images time-consuming.

Method	Execution Time (secs)
Bilinear Interpolation	0.4
Variable Number of Gradients	4.7
Patterned Pixel Grouping	0.8
AHD	4.3

Table 2.1: Comparative performance of different interpolation methods: bilinear interpolation, variable number of gradients[CCP99], patterned pixel grouping[Lin03] and AHD[HP05]. They are ordered in terms of their effectiveness. AHD is ten times slower than bilinear, and fives time slower than the next most effective method, patterned pixel grouping.

While AHD produces superior results compared to other methods, it is significantly slower. Table 2.1 on page 19 shows the performance of the demosaicing algorithms implemented by Dave Coffin as part of dcraw. The

Execution Time (%)	Function
65	Remove artifacts
21	Build homogeneity map
6	Convert RGB to CIELAB
4	Interpolate red and blue (horizontal and vertical)
1	Choose interpolation direction
0.5	Interpolate green (vertical)
0.4	Interpolate green (horizontal)

Table 2.2: Breakdown of AHD execution time. Shows the execution time of each function as a percentage of the total execution time. The results were obtained using `callgrind`, the program was run with the parameters recommended by Hirakawa and Parks, which are three artifact removal iterations and a 5x5 homogeneity neighbourhood. Results less than 0.4% have been omitted

methods are ordered by quality, from the least effective method, bilinear, to the most effective, AHD. AHD is ten times slower than bilinear, the most basic algorithm and five times slower the patterned pixel grouping[Lin03], the next most effective method.

The execution time of AHD is broken down into its major functions in Table 2.2 on page 20, which shows how much time is spent at each stage of the algorithm. It can be seen that the most computationally expensive part of the algorithm is the final artifact removal step, which involves several applications of a median filter. Building the homogeneity map also makes up a large percentage of the execution time.

In practise the conversion from RGB colour space to CIELAB can be avoided, and the homogeneity metric can be calculated in RGB colour space without a significant impact on the quality of the results[HP05, Kil08].

CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA's platform for general purpose computing on GPUs. It allows developers to write code which executes on the graphics card without specialised knowledge of the hardware. CUDA programs typically produce a 10x speedup compared to programs executed on the CPU and depending on the level of parallelism accelerations of 25x to 400x have been achieved[RRB+08, HSS09, SPF+07]. The rate of advances in GPU design are continuing to follow Moore's Law, while the rate of CPU performance has levelled off[OLG+07], meaning that in the future GPUs will offer even larger performance gains.

Image processing is particularly suited to this platform as it is easily parallelised. Experiments have shown that anti-aliasing, an image processing algorithm similar to demosaicing, improved in performance by more than 200x when moved to the GPU[RGC+09].

CUDA is supported by NVIDIA graphics cards with the GeForce 8 or later, Tesla and Quadro architecture. CUDA programs are written in 'C for CUDA' [Cor10c] which is compiled with `nvcc`, NVIDIA's C compiler. C for CUDA is a subset of C with extensions for executing functions in parallel. The most significant restriction is that it does not allow recursion or function pointers. GPU programs do not have a call stack, all functions are inlined which makes recursion impossible. Also, it does not support variadic functions, functions which take a variable number of arguments.

CUDA Execution

CUDA programs execute the sequential parts on the CPU, the 'host', and the parallel parts on the GPU, the 'device'. Device functions are called kernels. A kernel is invoked on a certain number of threads with a specific thread layout, the number of threads necessary is determined at runtime. A CUDA program may create thousands of threads, there is very little overhead associated with thread creation. The threads are arranged in a hierarchy.

Groups of cooperating threads are contained in a block, a three dimensional structure. Blocks are arranged in a two dimensional grid. The grid contains all the blocks for a particular invocation of the kernel.

The block is an important concept, as threads within a block may communicate with one another, while there is no direct interaction between threads in different blocks (or different grids). CUDA provides access to fast shared memory for threads within a block. The cost of accessing shared memory is only one cycle, the same cost as accessing registers. However, if there are bank conflicts, that is if multiple threads access the same address, the requests will be serialised, incurring a performance penalty. CUDA also provides a barrier synchronisation method between threads in the same block. After synchronisation all writes to device memory and shared memory are guaranteed to be visible.

At runtime each block is assigned to an available streaming multiprocessor on the GPU for execution. Each multiprocessor has eight streaming processors, and can accommodate up to eight blocks. Each multiprocessor has 8,192 registers which are divided among the threads.

The number of blocks on a multiprocessor is referred to as the 'occupancy'. Knowing the level of occupancy is important for optimisation, as a higher occupancy means the processor will be better able to schedule the blocks to hide memory latency. The occupancy level is determined by the block size, and also the number of registers and shared memory needed by the threads. Therefore the number of resources available on the multiprocessor can become a a limiting factor in the number of blocks that can be scheduled.

Blocks are scheduled by the runtime system according to the best configuration for the hardware. CUDA offers no guarantees about how the blocks are distributed, or the order in which they are executed. This allows the scheduler enough flexibility to choose the best strategy for the particular hardware model, completely transparently to the developer.

Blocks are split into warps for execution. The warp size is dependent on the hardware, but a typical size is 32 threads. The execution of warps will be

interleaved, so that the multiprocessor is not idle while one warp is waiting for a memory access to return. Threads in a warp execute in parallel, in a single-instruction multiple-thread, SIMT[Cor10a] manner. This is similar to SIMD, except each thread has its own instruction pointer and set of registers. The same instructions are broadcast to all threads, which execute in lockstep. However, if threads take diverging branches, the threads on the alternate branch will be temporarily disabled. When the branches re-converge, the threads will be re-enabled. A side effect of this model is that diverging branches incur a performance penalty, as both branches must be issued to all threads. This only applies to threads within the same warp. If all threads in the warp take the same execution path, there is no penalty.

CUDA Memory Layout

CUDA programs contain two addresses spaces, the host memory and the device memory. Device memory is allocated using the cudaMalloc functions provided the CUDA runtime library, data is copied to and from the device using cudaMemcpy. Copies may be synchronous or asynchronous, synchronous copies can take advantage of DMA for faster transfers. Nevertheless, there is a signification overhead copying data to and from the device, programs that perform only a few instructions on the device can actually be slower overall than the same program executed solely on the host.

There are several kinds memory available on the device. Firstly, each thread can use registers for local variables. Register provides the fastest access, taking 1 cycle. As discussed in the previous section, registers are local to a thread. They are the best choice for storing intermediate results. However, if a thread uses more registers than are available, the values will be spilled to local memory.

Threads may use a small amount of local memory, up to 16K. Local memory is private to a thread. However, like global memory, it is located in DRAM and access may take 400-600 cycles. Local memory is used for arrays and for storing spilled results from registers. However, accessing local memory is

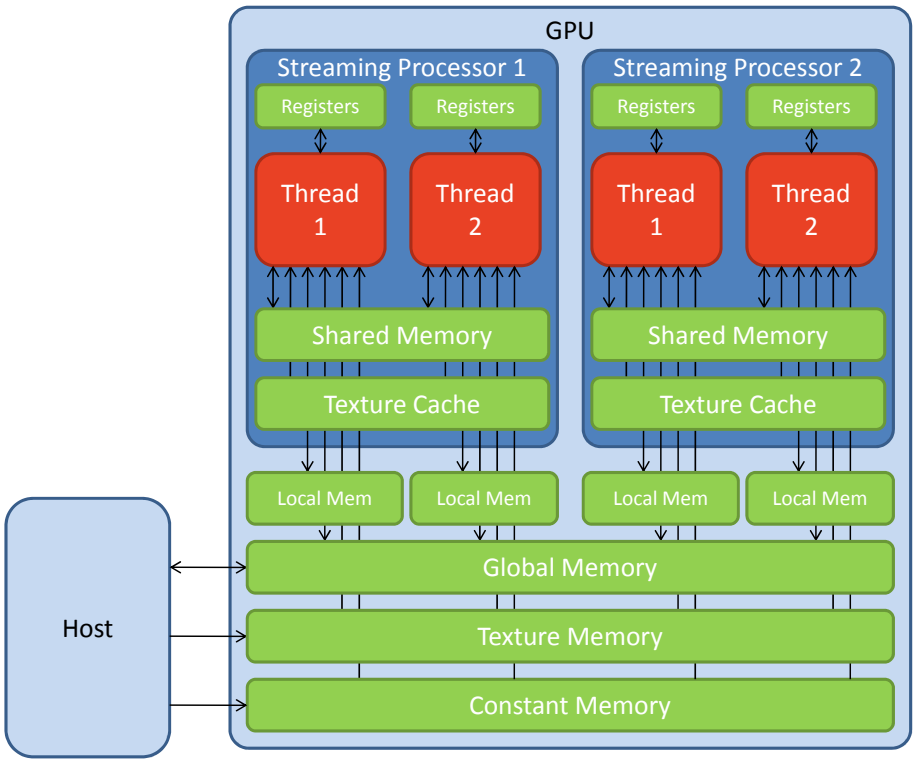


Figure 2.8: CUDA Memory Layout. Threads have fast access to shared memory and the texture cache, which are nearby on the streaming processor. Accessing other types of memory incurs a large penalty as they are stored in DRAM. Only global, constant and texture memory may be accessed by the host.

slow, and it should be avoided where ever possible.

There are two types of read-only memory, constant memory and texture memory. They are visible to threads in all blocks. Both types of memory are cached on the multiprocessor. Any structure can be stored in constant memory, while textures are more specialised. They are designed for one, two or three dimensional arrays. The elements of a texture must obey certain alignment rules. They are optimised for spatially local access. In addition, the hardware provides a built-in mechanism for interpolating values at non-integer coordinates.

The final type of memory provided by CUDA is global memory, this is visible to all threads and is read-write. It is typically used for storing results. Global memory is not cached, so a single access can take several hundred cycles. However, if threads within a warp access nearby addresses, the multiprocessor can coalesce the operations into one or more 128 byte wide transactions. Coalescing will occur if the threads access memory sequentially according to their thread ID. So, for example thread 0 accesses an array at index 0, thread 1 access the array at index 1, and so on for each thread in the warp. This restriction on the access pattern has been relaxed slightly in the most recent NVIDIA series 10 GPUs. However, as these only became available in late 2009, this dissertation will only consider series 9 or earlier GPUs. In CUDA terms, these are devices which support compute capability 1.0 or 1.1.

Chapter 3

Design and Implementation

CUDA Implementation of AHD

This section will describe how I implemented a parallel version of the AHD algorithm on CUDA. The AHD algorithm has four main stages: interpolation, calculating the homogeneity metric, direction selection, and artifact removal. Each of these steps depends on results from the previous stage, so they must be executed sequentially. They are implemented as separate kernels described below.

1. **Interpolate green values**
   This first step in the AHD algorithm is to interpolate the missing green values, for the horizontal and vertical directions, according to formulae (2.9) and (2.10). The kernel reads its input from the Bayer image, which is bound to a texture. The results are written to the horizontal and vertical images, stored in global memory.

2. **Interpolate red and blue values**
   Now that green values are available at all coordinates in the image, the red and blue values can be interpolated using formula (2.12). This kernel is invoked twice, once for the horizontal image and once for the vertical image. The RGB values are converted to CIELAB values, and

written to global memory.

3. **Build homogeneity map**
   This kernel determines the luminance and chrominance thresholds, according to formula (2.17). Using these thresholds, it calculates the homogeneity value for both the horizontal and vertical images. The homogeneity value is obtained by counting the number of pixels within a 5x5 neighbourhood which are within the two thresholds. The homogeneity values are written to separate horizontal and vertical homogeneity maps.

4. **Select direction**
   The direction used for the final result is the one which has the larger homogeneity value, averaged over a 3x3 neighbourhood. Depending on the direction selected, the kernel reads the appropriate CIELAB values from the interpolated image. It converts the CIELAB values back into RGB, and writes the result to the final image stored in global memory.

5. **Artifact removal**
   The artifact removal kernel applies a median filter to each of the channels. The filter is applied to the difference between the red or blue and green channels. This kernel executed three times. After each execution the results are bound to the input texture, so that they are available for the next iteration.

Memory Layout

The Bayer image needs to be available to the GPU threads, so it is copied to the global memory and bound to a two dimensional texture. Textures provide faster access than global memory as they provide caching, and are optimised for two dimensional access. This is ideal for the AHD kernels, as they will operate on two dimensional data: the source image, interpolated images and the homogeneity map.

The results of each kernel will be written to an array in global memory. As

global memory is visible to all threads, the results are immediately available to the following kernel. The arrays are stored in memory in row-major order. This layout allows them to be bound to a texture to be used an input for the next kernel.

For best performance global memory reads and writes should be coalesced, that is, consecutively numbered threads should access consecutive addresses. All data read by the kernels is from texture memory, where coalescing is unnecessary. So for this application, only writes to global memory need to be coalesced. The two interpolation kernels write tri-stimulus values, RGB and LAB values. The three values need to be combined into a single struct, `char4` for RGB values and `float4` for CIELAB values to fit the alignment requirements for coalescing. These structs can be written to memory as 16-bit and 128-bit words. The structs are wider than is strictly necessary, however the performance gains from coalescing outweigh the overhead of writing wider values.

I found that a straightforward implementation of the algorithm runs into the memory limit of the most GPUs. AHD makes extensive use of buffers for intermediate results. There are two buffers for the interpolated images, and two buffers for the homogeneity maps. For a typical 12 megapixel photo, this amounts to around 400MB. A mid-range GPU has 512MB of memory, some of which may already be allocated for graphics rendering. In this situation the program will often fail to allocate the large contiguous buffers necessary. To overcome this issue, I added an initial step which would split the image into tiles. The algorithm would then be run over each tile separately, and the results combined back into a single image. The tile size is chosen to be the largest possible, based on the amount of free memory available on the GPU. This ensures that attempts to allocate memory will be successful, while maintaining the highest achievable level of parallelism.

Threads

The implementation was designed to operate with one thread per pixel. Each thread is assigned a coordinate in the image and is responsible for updating that coordinate in the result. Threads are able to differentiate between themselves by their position in the grid and block, which are available in the `blockIdx` and `threadIdx` variables respectively. From these values, their coordinate in the image can be calculated using the following method:

$$\mathrm{x} = \mathrm{blockIdx.x} \times \mathrm{blockDim.x} + \mathrm{threadIdx.x} \tag{3.1}$$

$$\mathrm{y} = \mathrm{blockIdx.y} \times \mathrm{blockDim.y} + \mathrm{threadIdx.y} \tag{3.2}$$

The typical photo size is 12 megapixels, which will use on the order of 10^7 threads. This is well within the limits of CUDA, which allows a maximum of threads 10^{12} threads. The maximum number of threads per block is 512, and the maximum grid size is 65535×65535 blocks, altogether giving 10^{12} threads.

Optimal block size The block size used in the implementation was chosen to maximise occupancy and make best use of the resources. It should be a multiple of 32, as it will be divided into warps of size 32 and this ensures that all warps fully populated. Additionally, the number of warps per block should divide 24, which is the maximum number of warps per multiprocessor. One configuration which fits these requirements is a block size of 256, which is the size used in the implementation.

A large block size is preferable for kernels which are memory bound, as there will be more threads available for scheduling to hide memory latency. However, with larger block sizes the occupancy becomes limited by the number registers available. The kernels used by AHD are quite complex, using between 10 and 25 registers, out of a maximum of 32. Figures 3.1 and 3.2 show

the effects that varying the block size and the number registers has on the occupancy.

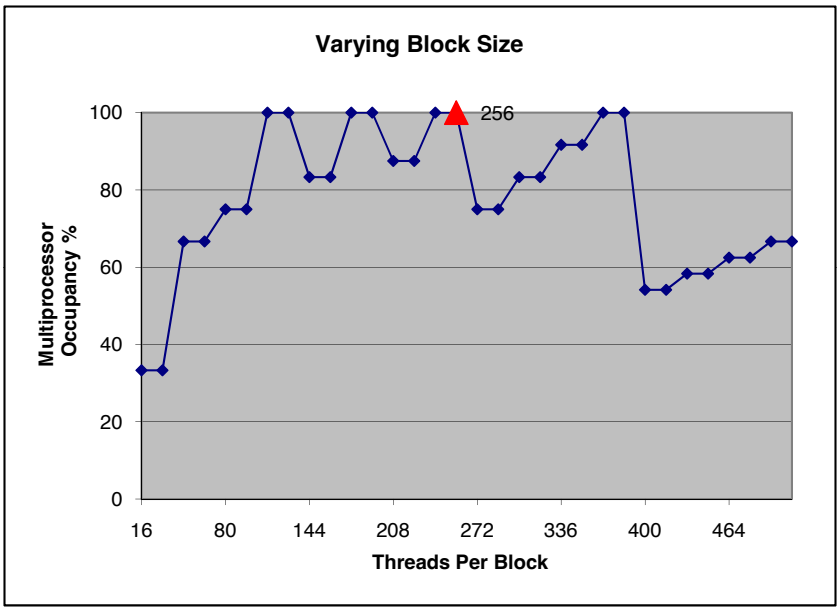


Figure 3.1: Shows the effect that block size has on occupancy, assuming no other restrictions such as register usage. Block sizes which are multiples of 32 achieve 100% occupancy. The red marker indicates the block size chosen for the implementation.

Kernel Level Optimisations

CUDA kernels are written in C for CUDA, which is largely compatible with C. Developing kernels is intended to be straightforward for developers with knowledge of C. However, an implementation that does not take the nature of the CUDA architecture into account will have unexpected performance problems. This section describes some of the optimisations made at the function level for better performance on CUDA.

Figure 3.3 on page 33 shows the effect each of the optimisations had on execution time. The results were obtained using the CUDA Visual Profiler. The profiler collects data from the hardware profile counters, and can record the

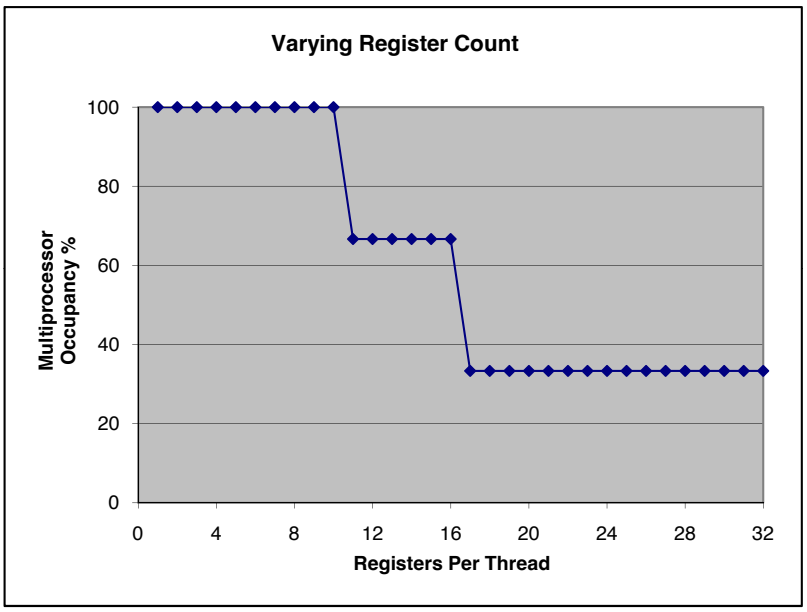
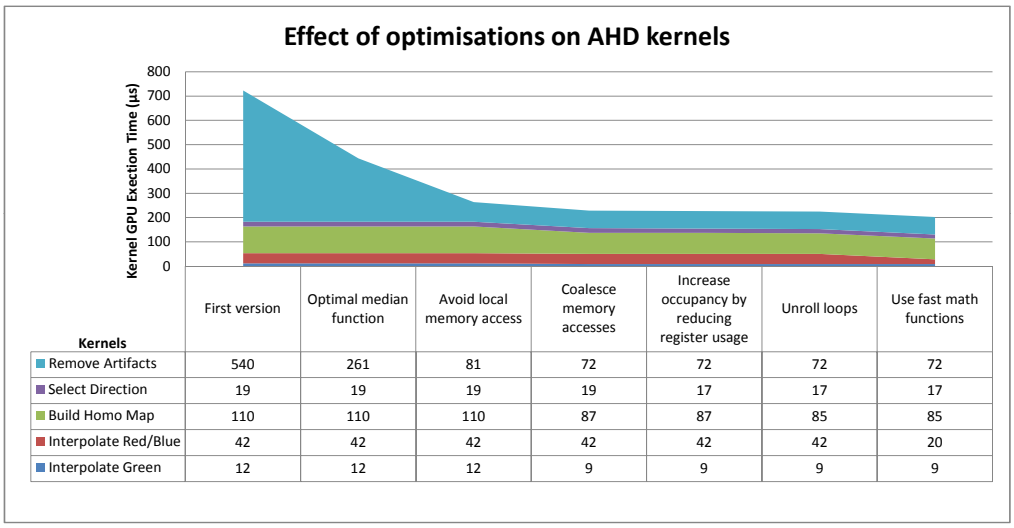


Figure 3.2: Shows how using a large number of registers can restrict the occupancy. As threads use more registers, fewer warps can coexist on the multiprocessor. (The occupancy is a function of both the block size and the number of registers, this graph shows the occupancy for a block size of 256).

number of registers used, the number of coalesced and uncoalesced memory accesses, the number of cache hits and misses and the number of diverging branches. One caveat however, is that only one streaming multiprocessor keeps profiling data. So, profiling information is only collected for a sample of the threads, the results for the entire grid are extrapolated from this data.

Execution Path As discussed previously one performance pitfall is diverging branches. Divergent branches cannot be executed in parallel as the same set of instructions are broadcast to all active threads. For this application, the largest source of diverging branches was in the artifact removal kernel. The kernel applies a median filter to the differences between the channels, and replaces the current pixel with the median value from its 3x3 neighbourhood. The median value is found by sorting the values using an insertion sort, and then retrieving the middle value. However, the insertion sort routine contains a conditional while loop which can cause the threads to diverge,



Kernels	First version	Optimal median function	Avoid local memory access	Coalesce memory accesses	Increase occupancy by reducing register usage	Unroll loops	Use fast math functions
Remove Artifacts	540	261	81	72	72	72	72
Select Direction	19	19	19	19	17	17	17
Build Homo Map	110	110	110	87	87	85	85
Interpolate Red/Blue	42	42	42	42	42	42	20
Interpolate Green	12	12	12	9	9	9	9

Figure 3.3: The effect of various optimisations on the GPU execution time of the kernels.

Algorithm 3.1.

However, as the median function will always be applied to an array of nine elements, it is possible to write an optimal median function that contains the minimum number of compares and swaps[Smi96]. It performs a partial sort of the array, leaving the median element in the middle position. It is particularly well suited for CUDA applications as it contains no diverging branches and performs the minimum number of memory accesses, Algorithm 3.2.

Another situation where diverging branches arise is boundary checking at the edges of the image. This affects all the kernels, as their results are based on the neighbourhood of the current pixel. Accessing neighbouring pixels requires boundary checking to avoid reading outside of the image. However, this will cause divergence around the borders of the image. As these reads are from texture memory, the checks can be avoided completely by taking advantage of the texture addressing mode. The addressing mode specifies how out of bounds accesses should be handled. They can be clamped to the minimum and maximum values, or wrap around to the opposite side of the

Algorithm 3.1 Original version of median function used by the artifact removal kernel. The while loop on lines 6-9 causes the threads to diverge.

```
1  DEVICE int median(int *a, int n) {
2    /* Perform insertion sort */
3    for (int i = 1; i<n; i++) {
4      int val = a[i];
5      int j = i;
6      while (j > 0 && a[j-1] > val) {
7        a[j] = a[j-1];
8        j--;
9      }
10     a[j] = val;
11   }
12   /* Return median value */
13   return a[n/2];
14 }
```

texture. For this application, clamped is the most suitable mode. By using this approach, all branching is avoided. Additionally, it provides a further degree of parallelism as the range checking is now performed on the texture unit instead of the stream processor.

Local Memory Another performance issue was the use of local memory. This was an additional contributing factor to the slowness of the artifact removal kernel. The kernel applies a median filter to the difference between two channels. The two channels are subtracted, and values are stored in a local array, which is an automatic variable. Using the method in Algorithm 3.3, the array will reside in local memory. Local memory is actually located in DRAM, like global memory, meaning each access will take several hundred cycles. Ideally, these values should be stored in registers where the cost of access is only one cycle. However, the compiler cannot perform this optimisation while the array is indexed by a variable as it cannot predict the contents of the variable, and it cannot predict which positions in the array will be accessed. This can be overcome by manually unrolling the roll and using

Algorithm 3.2 Improved version of the median function, based on code from [Dev98]. This is the optimal method for finding the median of nine values. Additionally, it contains no diverging branches.

```
1  #define SORT(a,b) { if ((a)>(b)) SWAP((a),(b)); }
2  #define SWAP(a,b) { int temp=(a);(a)=(b);(b)=temp; }
3
4  DEVICE int median9(int * a) {
5      SORT(a[1],a[2]); SORT(a[4],a[5]); SORT(a[7],a[8]);
6      SORT(a[0],a[1]); SORT(a[3],a[4]); SORT(a[6],a[7]);
7      SORT(a[1],a[2]); SORT(a[4],a[5]); SORT(a[7],a[8]);
8      SORT(a[0],a[3]); SORT(a[5],a[8]); SORT(a[4],a[7]);
9      SORT(a[3],a[6]); SORT(a[1],a[4]); SORT(a[2],a[5]);
10     SORT(a[4],a[7]); SORT(a[4],a[2]); SORT(a[6],a[4]);
11     SORT(a[4],a[2]); return a[4];
12 }
```

only constant indexes. The compiler provides a pragma to perform this optimisation automatically, `#pragma unroll`, but unfortunately it cannot unroll nested loops.

Fast Math Functions The CUDA instruction set includes several specialised instructions for mathematical functions. These include sin, cos, log2 and exp2. They execute in one cycle[Cor10b]. These come in useful for the second interpolation kernel which converts RGB values to CIELAB values. The conversion method, which is described in Appendix A, includes several cube root operations. The first version of the kernel used `pow(x,1/3)` to compute cube roots. The `pow()` function, which is provided by the CUDA standard library, is compiled into basic arithmetic instructions. However, there is a faster but slightly less accurate version, `__pow`. It is implemented using the formula $2^{y*log_2 x}$, which in CUDA code becomes `exp2f(y * __log2f(x))`. The `exp2` and `__log2f` functions compile directly to PTX instructions which execute in one cycle. This greatly reduces the overall number of instructions, and halves the execution of the kernel. Although this method provides slightly lower accuracy, it does not have a perceptible affect on the quality

Algorithm 3.3 This function returns the median of the difference of two channels. The array `diffs` resides in local memory, causing expensive accesses to DRAM. The compiler is prohibited from optimising this function and storing the array in registers by line 10, where the array is indexed by a variable.

```
1  DEVICE int median_diff(int x,int y,int chan1,int chan2){
2      int diffs[9];
3      int i = 0;
4      for (int dy = -1; dy <= 1; dy++) {
5          for (int dx = -1; dx <= 1; dx++) {
6              pixel val1 =
7                  tex_get_color(src,x+dx,y+dy,chan1);
8              pixel val2 =
9                  tex_get_color(src,x+dx,y+dy,chan2);
10             diffs[i++] = val1 - val2;
11         }
12     }
13     return median9(diffs);
14 }
```

of the result.

Unsuccessful Optimisations There are several other optimisations recommend by the NVIDIA's Best Practises Guide[Cor10b] such as reducing the number of registers, in order to increase occupancy. However, I found this to be a particularly difficult and unreliable optimisation. They are several approaches to reducing the number of registers used by a kernel, such as reordering calculations, unrolling loops and re-computing results instead of the storing them. However, it is often unclear where the registers are used and making changes to the source code can have unpredictable results on the register usage. It is possible to view the intermediate assembly code generated by the compiler. But, register allocation is performed by the assembler, not the compiler, and so the generated assembly code contains only references to virtual registers. The physical registers allocated will not correspond directly to the virtual registers, obscuring the actual register usage.

Adding to the difficulty, reducing the number of registers does not always increase performance. It is possible to increase occupancy while actually degrading performance. I have found that often the changes necessary to reduce the number of registers will increase the number of instructions issued leading to a longer overall execution time.

Mask Demosaicing

The most effective demosaicing algorithms, like AHD, avoid avoid zippering artifacts by determining the orientation of edges in the image. However, in flat areas where artifacts do not occur these methods do not provide any advantage over simple bilinear interpolation. The extensive computation performed to estimate the edge direction or avoid misguidance is unnecessary in regions where there are no edges. Given that flat regions make up the majority of the area of natural images[OF96, LZW03], a large portion of that computation could be avoided.

In this section I propose a new demosaicing method, Mask Demosaicing, which aims to improve on the performance of AHD while maintaining image quality. Mask Demosaicing achieves better performance by selectively applying a high quality demosaicing method to the areas in the image where artifacts are likely to occur. The demosaicing method chosen for these areas is AHD. However, the proposed method is not specific to AHD, any high quality but prohibitively slow technique could be used.

The remaining areas in the image are demosaiced using bilinear interpolation for its speed. As bilinear interpolation is only applied to flat or slowly varying parts of the image it will not introduce artifacts.

Algorithm 3.4 Mask Demosaicing outline

1. Perform a bilinear interpolation on the Bayer image.
2. Identify edges in the interpolated image.
3. Dilate the edge areas, this becomes the mask which will used in the next step.
4. Perform AHD demosaicing on the regions covered by the mask.

The biggest difficulty lies in identifying the areas where AHD should be applied and from these creating the mask. I propose two methods for detecting these areas, the first is based on the gradient, and the second is uses areas of high contrast.

Both methods operate on the full colour RGB image that has been produced

using bilinear interpolation. It is possible to detect edges directly in the Bayer image[JJ97]. However, during testing I found that performing edge detection in the interpolated image proved more reliable, as it is possible to detect higher frequency changes. This introduces an overhead into the method. Bilinear interpolation must be performed on the whole image, even though only part of the result will be used in the final image. However, as seen in Table 2.1 on page 19, bilinear interpolation is roughly ten times faster than AHD. So, if less than 90% of the image is covered by the mask, the overhead will be offset by the performance gained by avoiding AHD.

Gradient based edge detection

The first edge detection method is based is a standard technique from computer vision, edge detection using first order gradients. The Sobel operator[SF73] is used to calculate the horizontal and vertical gradients. The image is convolved with the 3x3 operators shown below, to produce the gradient maps G_x and G_y.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * L' \tag{3.3}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * L' \tag{3.4}$$

The magnitude of the gradient is obtained by:

$$G = \sqrt{G_x^2 + G_y^2} \tag{3.5}$$

However, a simpler approximation is sufficient for this purpose:

$$|G| = |G_x| + |G_y| \tag{3.6}$$

A threshold is applied to the gradient map G to give a binary image. This binary image is the first step in creating the mask to be used with AHD. The values in the gradient map range from zero to 255. The threshold used in the implementation was 40. This figure was chosen so as to include perceptible edges and avoid small gradients caused by noise.

Edge detection is performed in grayscale image. The interpolated image is converted to grayscale using the following luminance method[GW92], where the RGB components are weighted according to the response of the human eye.

$$L = 0.59G + 0.3R + 0.11B \tag{3.7}$$

It also possible to perform edge detection using only the green channel. This takes advantage of the fact it contains twice as much information as the red or blue channels and avoids the extra operations needed for grayscale conversion. However, I found that edge detection in the green channel was slightly less effective and the minor improvement in performance did not justify the reduction in quality.

Artifact based edge detection

The second method for creating the mask is based on artifact detection rather than edge detection. Instead of attempting to identify edges in the interpolated image, it identifies regions that contain artifacts. The presence of artifacts indicates where bilinear interpolation has failed, and where AHD should be applied. This has the potential to be more reliable than straight forward edge detection as it directly identifies the problematic regions.

The method works on the assumption that artifacts are characterised by abrupt and large changes in colour. I define the variation in colour around

a point to be the average colour distance between the point and its eight neighbours. A larger variation in colour indicates that the point is more likely to contain an artifact. The variation V at a point i, j is given by

$$V(i,j) = \frac{1}{9} \sum_{x,y \in \aleph_{i,j}} \text{dist}(I'_{i,j}, I'_{x,y}) \tag{3.8}$$

where $\aleph_{i,j}$ is set of coordinates of the eight neighbours of i, j, and $\text{dist}(I'_{i,j}, I'_{x,y})$ is euclidean distance between the two RGB values $I'_{i,j}$ and $I'_{x,y}$.

Applying this operation to the interpolated image gives a map of the variation in colour. A threshold is applied to this map to create a binary image, representing the areas where artifacts occurred. The threshold selected for the implementation was 50, this figure was determined by experimentation.

Dilation

A region surrounding the edge is required by AHD to calculate the homogeneity value. To ensure that a sufficient area around is available within the mask, the regions in the binary image are grown by performing a dilation. The structuring element used in the dilation is:

$$K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.9}$$

The structuring element K is applied to each pixel p in the binary image. If the intersection of K with the binary image is non-empty, if there is at least one white pixel inside K, then the pixel p is set to white. This ensures that there is at least two pixels surrounded each edge pixel. This is sufficient to create a homogeneity map with the recommended 5x5 neighbourhood.

Applying AHD

The original AHD algorithm is applied to the areas of the image covered by the mask. For simplicity of implementation the interpolation steps are performed on the entire image. This is not expected to have a significant effect on the performance of the final algorithm as it accounts for only a small percentage of the overall runtime. However, the mask is applied in the more expensive steps which follow: colour space conversion, homogeneity calculation and artifact removal.

The bilinearly interpolated image is updated with the results of the AHD algorithm following the direction selection step. This is before the final stage, artifact removal has been applied. Artifact removal is applied to each pixel under the mask. However, as it operates in a 3x3 neighbourhood, at the edges of the mask the bilinear interpolated results are included in the filter. This has the effect of smoothing together the two methods and avoids discontinuities where they meet.

Results

The results of both methods can be seen in Figure 3.4 on page 44 and Figure 3.5 on page 45. Both methods are largely successful in identifying the regions in the image where AHD should be applied. In the areas where bilinear interpolation is used there is no perceptible difference in quality. Also, there is no seam where the two methods meet.

The edge detection method shows some problems detecting edges on part of the fence. In this region the magnitude of the gradient is quite low. It falls below the threshold used to create the binary image, and the area is not included in the mask. This shows the limits of what can be achieved with a simple thresholding method, to overcome this issue a more sophisticated method for creating the binary image would be necessary.

The artifact detection method does not have this problem. It correctly identi-

fies the fence as containing an edges, despite the fact the edges in the original image are quite weak. The results produced by bilinear interpolation in this area are highly artifacted,Figure 2.2 on page 7, which is recognised by the artifact detection method.

The results also show that the artifact detection method has included a larger part of the foreground in the mask. This is because artifact detection is more sensitive to textured and noisy regions. Textured regions may not contain strong horizontal or vertical edges, and so they may be recognised by the gradient based edge detector. However, bilinear interpolation will cause blurring and a loss of detail that is undesirable. For this reason it more appropriate to use AHD in these regions.

A full evaluation of the quality of the images produced can be found in the following chapter, along with an evaluation of the runtime performance.

(a) Mask produced by the edge detection method

(b) Final result

Figure 3.4: Results of Mask Demosaicing using method (1), edge detection.

(a) Mask produced by the artifact detection method

(b) Final result

Figure 3.5: Results of Mask Demosaicing using method (2), artifact detection.

Chapter 4

Evaluation

Performance Analysis

The section compares the performance of a single-threaded implementation of AHD using the host machine's CPU, and a parallel version running on the GPU. As an indicator of baseline performance, bilinear interpolation is also included in the evaluation. The runtime performance of Mask Demosaicing is also evaluated. The final part of the chapter examines the quality of images produced with Mask Demosaicing using both quantitative and subjective measures.

Method

The performance of the two AHD demosaicing routines were compared, one implemented in C which executes on the host CPU, and one implemented in CUDA. Similarly, two bilinear interpolation routines were compared, a CUDA version and host version written in C.

The execution time of the routines were recorded using the times(2) system call[Ker09], which records the CPU time spent executing the process. It includes both user time and system time. The resolution of the timer on

the test machine was 10 milliseconds. To minimise the effect of random errors, each test was executed six times and the results were averaged. The recorded result is the execution time for the demosaicing operation, excluding time spent reading the image from disk or saving the results. It should be noted that for CUDA programs the time includes copying the image to device memory and retrieving the results.

The sample images used for testing were selected from a repository of RAW images[1]. The photos were chosen to represent a variety of scenes and camera models. RAW images are encoded in proprietary manufacturer specific formats. The images were decoded to uncompressed Bayer images using the routines included in dcraw. Demosaiced versions of the sample images can be seen in Figure 4.4 on page 53.

The images varied in resolution from 12 megapixels to 5 megapixels. In order to test the performance for various image sizes, the original images were cropped down to a range of sizes, between 12 and 0.1 megapixels.

Experiment environment

Tests were performed on an Intel Core2 Quad CPU Q9400 at 2.66GHz running Ubuntu Linux 9.10, with kernel version 2.6.31-21-generic. The programs were compiled with gcc 4.3.4, and nvcc 3.0. CUDA tests were carried out on an GeForce 9600 GT graphics card which has 6 multiprocessors and a total of 48 stream processors (8 per multiprocessor) and 512MB of memory. Version 3.0 of the CUDA runtime and driver were used.

[1] `http://raw.fotosite.pl`

Results

Bilinear Interpolation

Bilinear interpolation is the fastest demosaicing algorithm. The number of instructions per pixel is 2 compares, 8 adds and 2 right-shifts. The execution times for bilinear interpolation running on the host and on CUDA are shown in Figure 4.1 on page 50.

As the amount of computation required so low, CUDA is not expected to provide much an improvement. The results show that for images under six megapixels, the CPU version is actually faster than CUDA version. This is caused by the overheads associated with running CUDA programs. These include copying the image data to the GPU, launching the kernel, creating CUDA threads, distributing data to the streaming processors, retrieving the results and finally copying the results back to host memory.

However, despite the large overhead which impacts performance on small data-sets, the CUDA version scales better than the CPU version. As the size of the image increases the execution time on CUDA grows more slowly compared to on the CPU. For images above six megapixels, the CUDA version overtakes the CPU one. It should be said though that it is only a modest improvement, execution time for the largest image, ~12 megapixels, takes 270ms on the CPU compared to 200ms on CUDA.

From these results we can conclude that for large data-sets, in this case images over 6 megapixels, CUDA offers a performance benefit. In this situation, as the computation performed is so small the performance gain is quite minor. However, for more complex calculations the performance gain is expected to be much larger.

AHD Demosaicing

The results of the AHD experiments are shown in 4.2. The execution time for AHD running on the host machine and on the GPU were tested. The

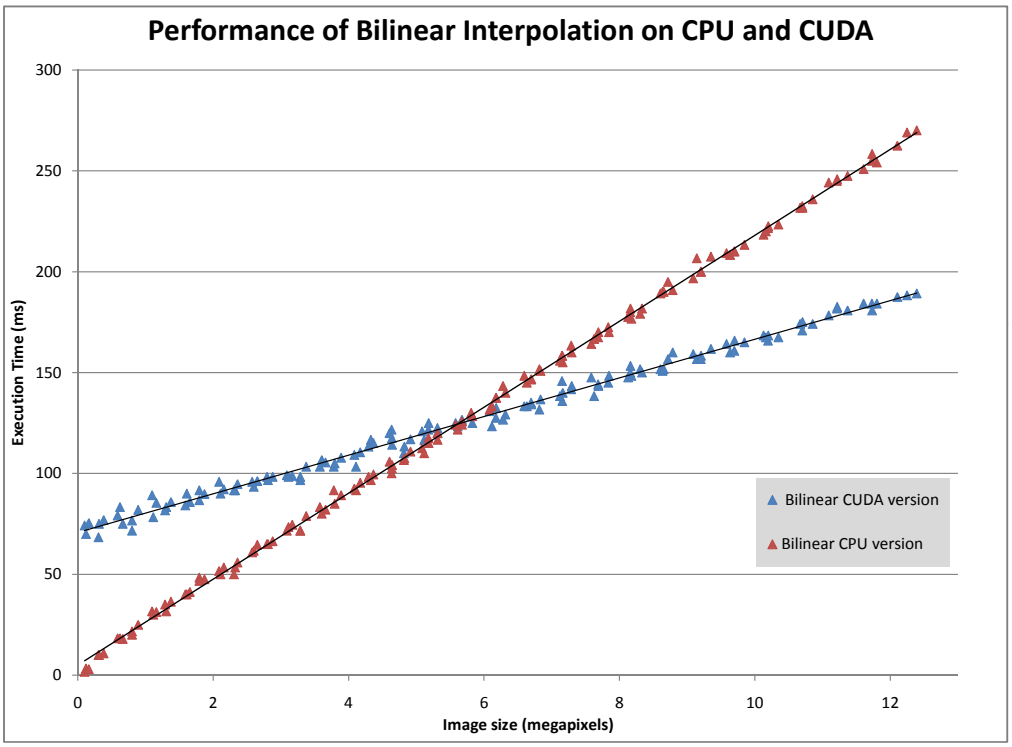


Figure 4.1: Comparison of the performance of bilinear interpolation running on the CPU and on CUDA.

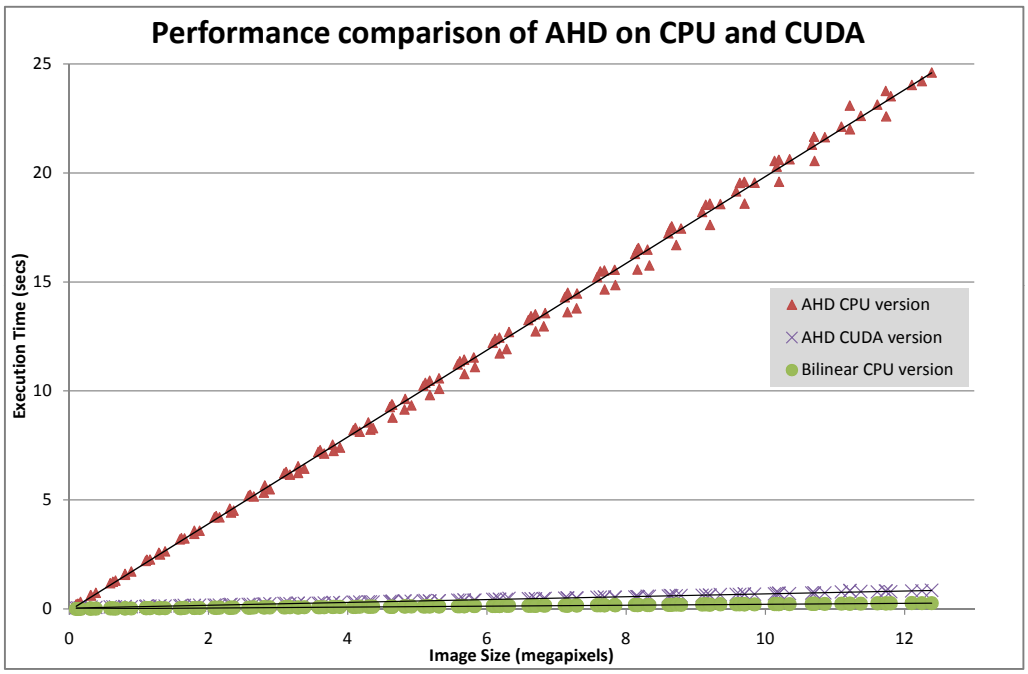


Figure 4.2: Comparison of the performance of AHD on CPU and CUDA.

performance of bilinear interpolation on the CPU is also included for comparison. The CUDA version shows a massive improvement in performance. It is 30x faster than the CPU version for image over six megapixels. For a typical 12 megapixel image the processing time on CUDA is approximately 850ms, compared to 23 seconds for the CPU version.

The performance of AHD on CUDA can be seen more clearly in Figure 4.3 on page 52, which omits the larger CPU results. It can be seen that for all image sizes the CUDA implementation outperforms the CPU version. Even for the smallest workload, the 0.1 megapixel image, the CUDA version offers a performance improvement.

Figure 4.3 on page 52 shows a spike in the execution time of the CUDA version around the 11 megapixel size. For images of this size and above, the memory needed for intermediate results was greater than the memory available on the GPU. In this situation the image is split into tiles and each tile is demosaiced separately. As a result, the level of parallelism is reduced

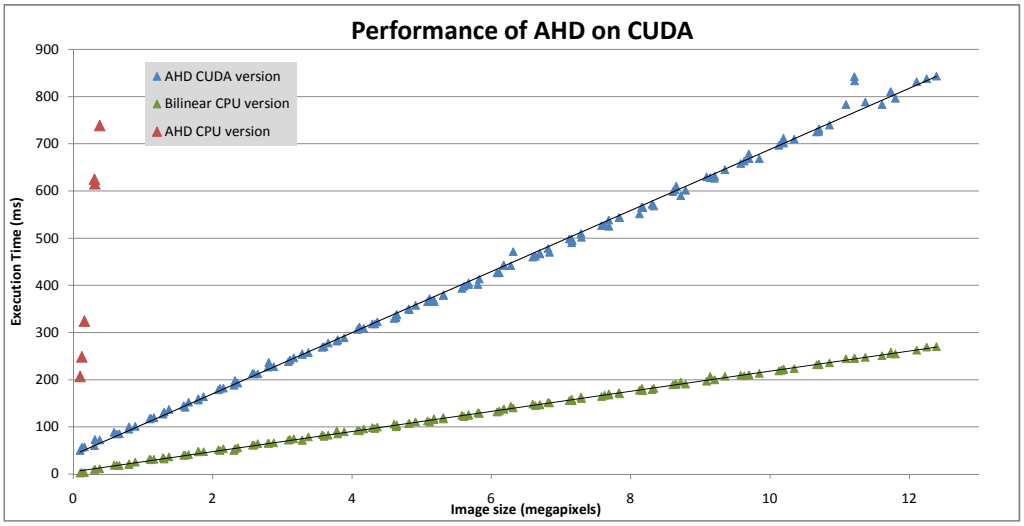


Figure 4.3: Performance of AHD on CUDA, bilinear interpolation is included for comparison. For clarity, the CPU results above 900ms are omitted.

causing the drop in performance.

CUDA offers a substantial improvement in performance over the single-threaded version. There are also some other benefits that are not evident from the timing results. The CUDA version stores all intermediate results on the GPU. Aside from the buffer used to hold the initial raw image data, it does not consume any host memory. AHD requires at least four intermediate buffers, for the two interpolated images and the two homogeneity maps. Real world implementations of AHD try to avoid excessive use of memory by performing demosaicing in tiles or a sliding window[Kil08]. Regardless of the method employed, a certain amount of memory required. The CUDA version is able to avoid any extra memory use by taking advantage of the memory available on the GPU, which would not otherwise be practically accessible.

Another benefit of having the results available on the GPU is that they can be displayed to the screen directly without the need to copy the data back to host memory. CUDA provides interoperability with OpenGL, so OpenGL textures can be mapped into the CUDA device address space. This provides an additional performance benefit for applications such as image editing software, where the results are displayed to the screen immediately.

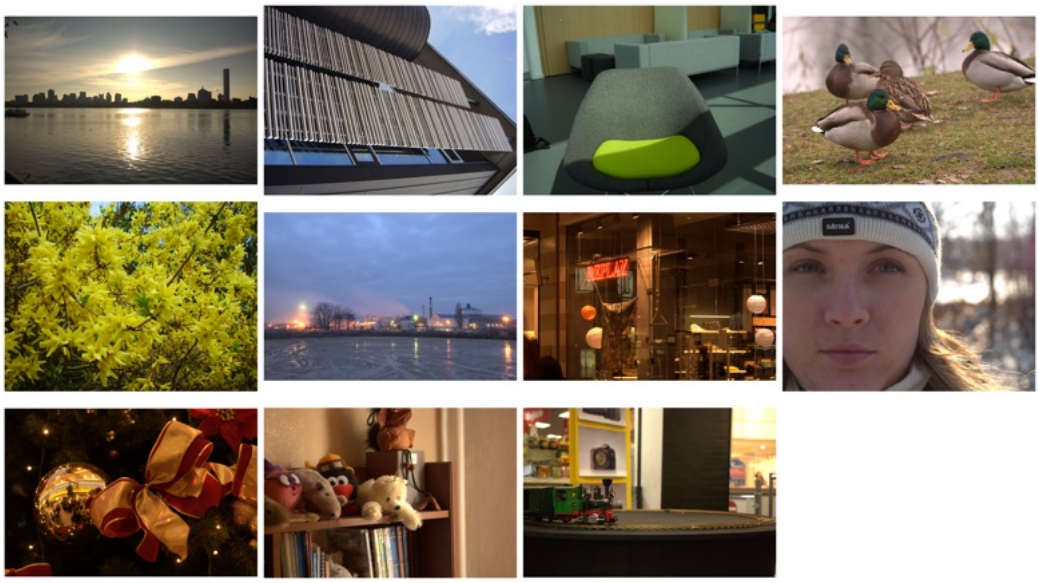


Figure 4.4: Sample images used for performance analysis.

For a 12 megapixel image, the time spent copying results from the device to the host is approximately 25ms. This could be avoided to give an extra 3% speedup to the demosaicing routine.

Mask Demosaicing

The Mask Demosaicing method works by detecting areas where artifacts are likely to occur and creating as mask of these areas. The area covered by the mask is demosaiced using AHD and bilinear interpolation is applied to the remaining areas. As a result, the performance is dependent on the content of the image. Images with a lot of detail or very noisy images will take longer to demosaic as more areas will be identified as edges. The images used in the performance analysis are shown in Figure 4.4 on page 53. The images are intended to represent a wide range of image types, including images with high and low amounts of detail, as well as photographs taken indoors and outdoors, under natural and artificial light.

The results of performance tests of Mask Demosaicing, 4.5, show the re-

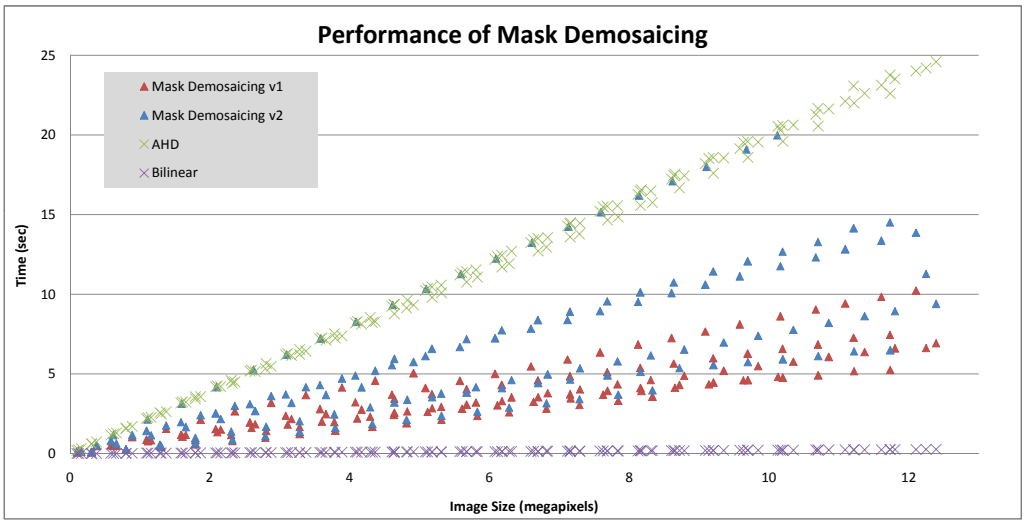


Figure 4.5: Shows the performance of the proposed Mask Demosaicing method compared to AHD and bilinear interpolation. The edge detection method is referred to as v1, and the artifact detection method as v2.

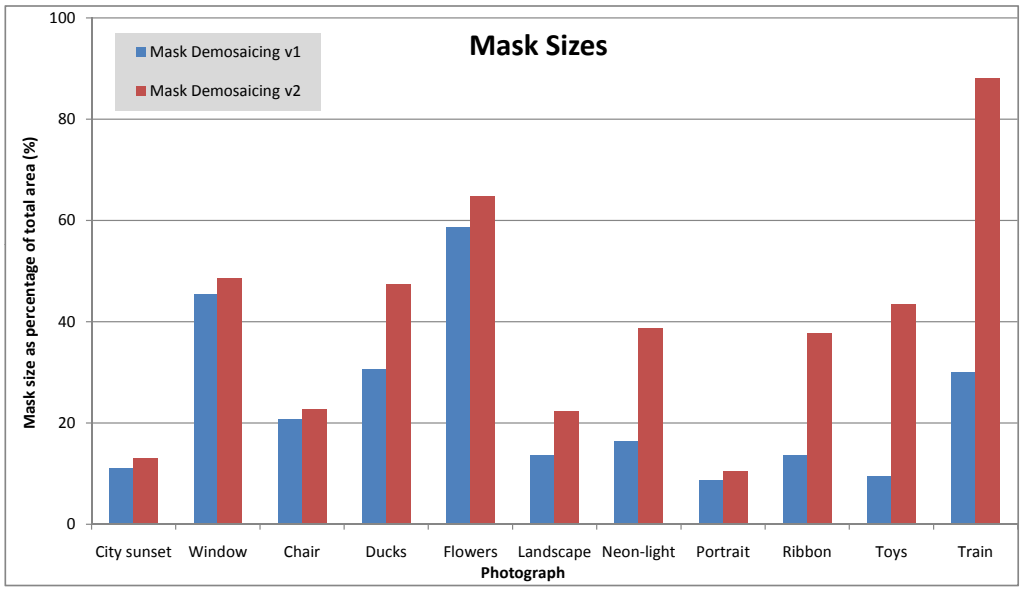


Figure 4.6: Shows the size of the mask for each of the sample images. The mask size is expressed as a percentage of the total image size. For the outdoor and well-lit photographs both methods produce masks of roughly the same size. However, the artifact detection method (v2) creates much larger masks for low light and noisy photographs.

sults vary according to the image content. All the images tested showed an improvement in performance compared to AHD with the edge detection. However, the performance of the artifact detection method is more varied, with some images experiencing an speedup of 4x compared to AHD, while others show no improvement.

The performance improvement is proportional to the mask size. The mask sizes for the sample images are shown in 4.6. For the outdoor and well lit images, both methods produce roughly similar sized masks. In this situation, the size is dependent on the amount of detail in the image, for example the 'City sunset' image, which has almost no detail in the foreground, has a mask size of around 15%; while the 'Flowers' photograph, which contains much more fine detail, has a much larger mask size, approximately 60%.

However, for low-light and indoor images the artifact detection method creates a much larger mask than the edge detection method. This can be seen in the mask sizes for the 'Toys', 'Ribbon' and 'Train' photographs. Photographs taken under these conditions contain a lot of noise. The artifact detection method cannot distinguish between noise caused by the sensors and artifacts introduced by bilinear interpolation, causing the larger mask sizes.

The 'Train' photograph is an example of the worst case performance of Mask Demosaicing with the artifact detection method. The photograph was taken indoors using a short exposure time to avoid blur on the moving train, and high sensitivity setting on the sensor, ISO 1600. In these circumstances the performance of Mask Demosaicing is almost equal to the performance of AHD.

In these situations, where it is known beforehand from the ISO setting that the image will be very noisy, it may be preferable to abandon any attempt to use Mask Demosaicing, or to fall back to Mask Demosaicing with edge detection. Another alternative might be to remove noise from the image before demosaicing. The latest research has found that denoising and demosaicing can be performed simultaneously[HP06, LLHL06, ZWZ07, PFBK08]. Further examination of this problem is beyond the scope of this dissertation,

Figure 4.7: Detail from the 'Train' image, demosaiced using bilinear interpolation. The photograph was taken with ISO 1600 with causes significant noise in the raw image.

but is an interesting topic for further research.

Mask Demosaicing Image Quality

This section compares the quality of images produced with Mask Demosaicing to those produced with AHD. Both objective and subjective quality measures are employed. The two methods were applied to a set of test images which had been down-sampled with a Bayer filter. Both sets of results were compared to original image to asses the quality of the demosaicing. The objective measures used are the mean square error, MSE, and a measure of zipper effect[LT03] designed specifically for evaluating demosaicing algorithms.

Quantitative Measures

The mean square error, MSE, is a widely used method of measuring the error introduced into a signal. It based on the difference between a reference signal and a noisy signal. It is commonly used to measure the effectiveness of image restoration or compression techniques.

The standard test suite for demosaicing algorithms is the Kodak Lossless True Color Image Suite[2]. It consists of 24 photos with a resolution of 768x512, showing various natural scenes. The images were down-sampled with the Bayer pattern, and then demosaiced using Mask Demosaicing, AHD and bilinear interpolation. The MSE was calculated as the difference between the original image and demosaiced image.

$$MSE = \frac{1}{NM} \sum_{i,j=1,1}^{N \times M} (I_{i,j} - I'_{i,j})^2 \tag{4.1}$$

where I is the reference image with dimensions $N \times M$, and I' is the interpolated image. The MSE was calculated separately for each colour channel.

The average MSE results for each method can be seen in Figure 4.8 on

[2]Available from http://r0k.us/graphics/kodak/

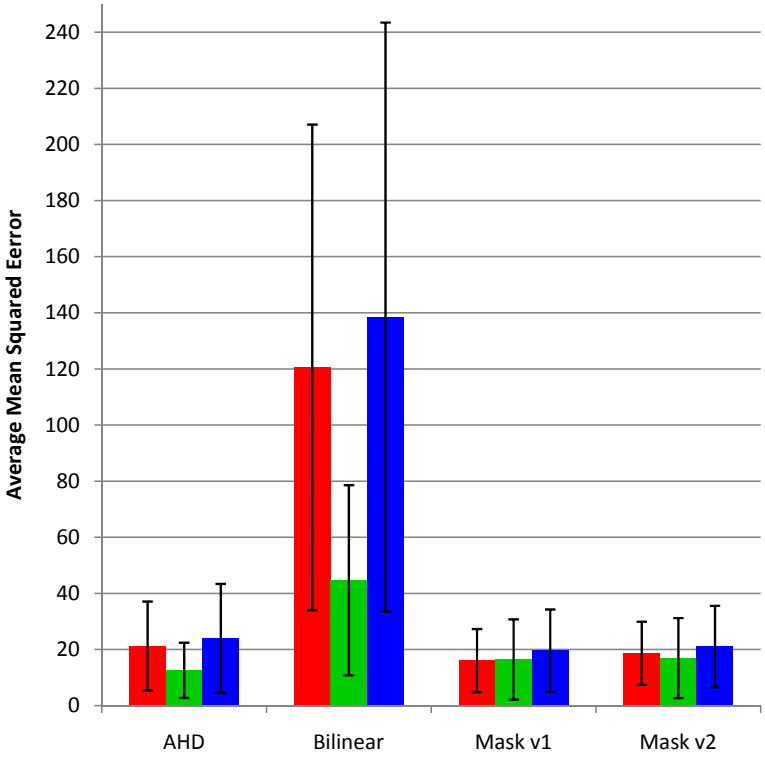


Figure 4.8: The mean square error averaged over 24 images in the Kodak image suite. The MSE is presently separately for each colour channel. The error bars show the standard deviation.

page 58. The results show only minor difference between the quality of AHD and Mask Demosaicing. The same evaluation was performed on other demosaicing techniques by Gunturk et al[GGA+]. They found that for other traditional demosaicing methods such as edge-directed[Kim99, JJ97, Hib95], constant-hue based[Cok87] and pattern matching[CCP99], the average MSE ranged between 60 and 20. They found that AHD was second most effective method of the ones they tested, it was out-performed only by Gunturk's alternating projections method[GAM02]. The results for Mask Demosaicing show that it out performs these traditional methods and is closer in quality to AHD than any of the other techniques.

Despite the widespread use of MSE in image processing, it is not always a good measure of the perceived image quality[WB09]. This is especially true for demosaicing algorithms where zippering artifacts may be under represented as they only make up a small percentage of the total area of the image.

The zippering measure was designed to detect checker-board artifacts introduced during demosaicing. The method detects zippering by looking for neighbouring pixels which had similar colour in the reference image, but which have perceptibly different colour in the demosaiced image. The most similar coordinate $s(i,j)$ of a point i,j is the neighbouring pixel with the smallest CIELAB colour distance, $\triangle E^*_{ab}$

$$s(i,j) = \min_{x,y \in \aleph_{i,j}} \triangle E^*_{ab}(I_{i,j}, I_{x,y}) \tag{4.2}$$

where $\aleph_{i,j}$ are the coordinates of the eight neighbours of i,j. Zippering is present at i,j if the colour difference between i,j and $s(i,j)$ in the reference image and the interpolated image is greater than a threshold δ

$$|\triangle E^*_{ab}(I_{i,j}, I_{s(i,j)}) - \triangle E^*_{ab}(I'_{i,j}, I'_{s(i,j)})| > \delta \tag{4.3}$$

The threshold, δ, is chosen to be 2.3 based on the limits of human colour

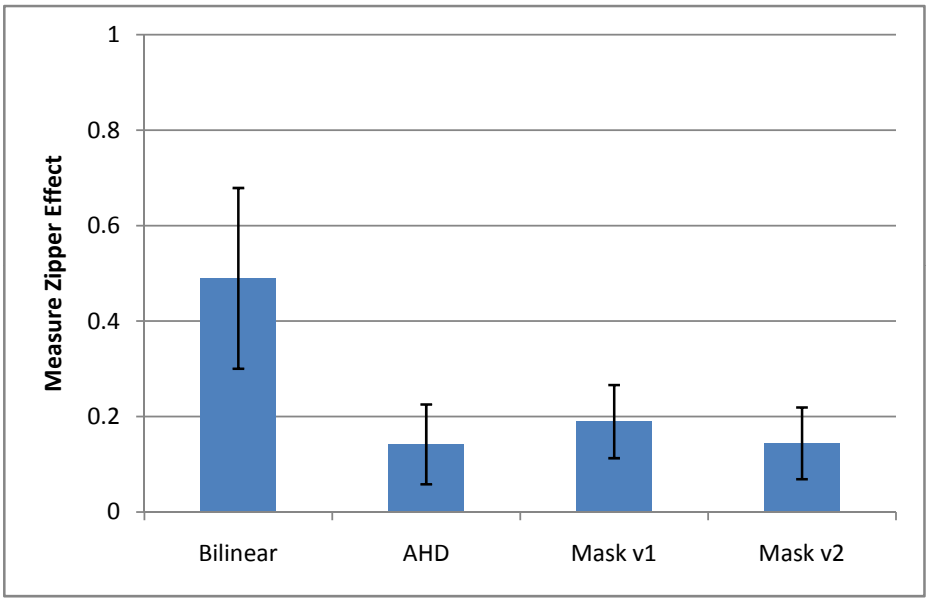


Figure 4.9: Measure of zipper effect averaged over 24 images in the Kodak image suite. The error bars show the standard deviation.

perception. The measure of zipper effect for the entire image is the number of pixels affected by zippering divided by the size of the image. This gives a result in the range zero to one, representing the estimated fraction of the image that is affected by zippering.

The measure of zipper effect for each demosaicing method is shown in Figure 4.9 on page 60. The results are averaged over the 24 images in the Kodak image suite. AHD and Mask Demosaicing using artifact detection show very similar results. Mask Demosaicing using edge detection performs slightly worse. This may be caused by low level gradients which fall below the threshold for edge detection, but which produce noticeable artifacts in the result. These are more easily identified by the artifact detection mechanism, as artifacts cause very large changes in colour within a small area.

Subjective Quality Measurement

The quality of Mask Demosaicing was also evaluated in terms of its subjective quality. The aim of the study was to determine whether there were perceptible differences between images demosaiced using AHD and Mask Demosaicing. Subjects were shown two copies of an image, one produced with AHD and the other produced with Mask Demosaicing. They were asked which they preferred or if they had no preference for either of the images. The images were viewed as large prints to give the most realistic testing scenario and ensure consistent conditions across tests.

Task

Subjects were presented with two photos of the same scene, labelled 'A' and 'B'. They were asked which photo they would prefer to have or if they have no preference. This task was repeated for six different pairs of images, shown in Figure 4.10 on page 62. The subject was given as much time as they wanted to look at each set of prints. They were free to move the prints around, to examine them at close range or compare them side by side.

The images were chosen to be representative of a variety of different scenes, both indoor and outdoor, with natural and artificial lighting. They also included images with extensive detail, lots of horizontal and vertical lines, noisy images and images including human faces.

Two version of each image were shown to the subjects, one produced with AHD and and one with Mask Demosaicing. The Mask Demosaicing images were created using the edge detection method. Both methods used the recommended parameters for best quality for the AHD algorithm. The letters 'A' and 'B' were assigned to the two images are random.

The images were printed on 10"x8" FujiFilm Crystal Archive glossy photographic paper at 300 DPI using a Fuji Frontier printer. Before printing, the images were converted from uncompressed PPM files to JPEG by ImageMagick using the highest quality setting. The prints were viewed under natural

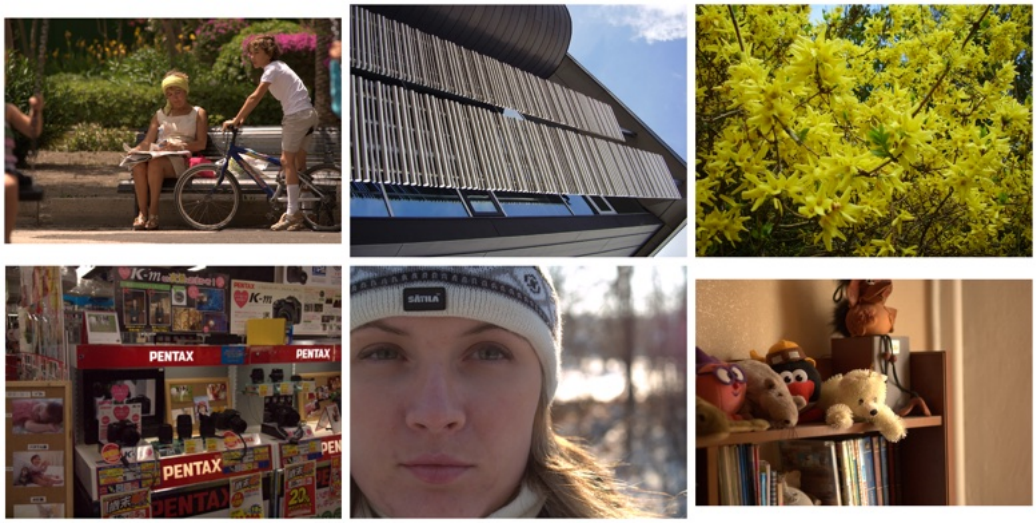


Figure 4.10: Images used in the user study. Subjects were presented with two version of each image, one produced with AHD and one with Mask Demosaicing.

lighting conditions.

The participants in the study were eight students from the Computer Laboratory aged between 18 and 30, with no knowledge of demosaicing or of the nature of the experiment. All subjects reported that they did not suffer from colour blindness. No incentive was offered to take part in the study.

Results

The total sum of the responses for all candidates and for all of the sample images is shown in Figure 4.11 on page 63. The results show that subjects showed the largest preferences for AHD, but that they also expressed no preference for a large number of the tests. This suggests that AHD and Mask Demosaicing produce perceptibly very similar results.

In general, the subjects remarked that they had difficulty finding any difference between the images, even after extensive study. When asked for the reason for their decision the majority of the subjects referred to difference in colour or lighting, stating that one version was 'brighter', or 'more saturated'.

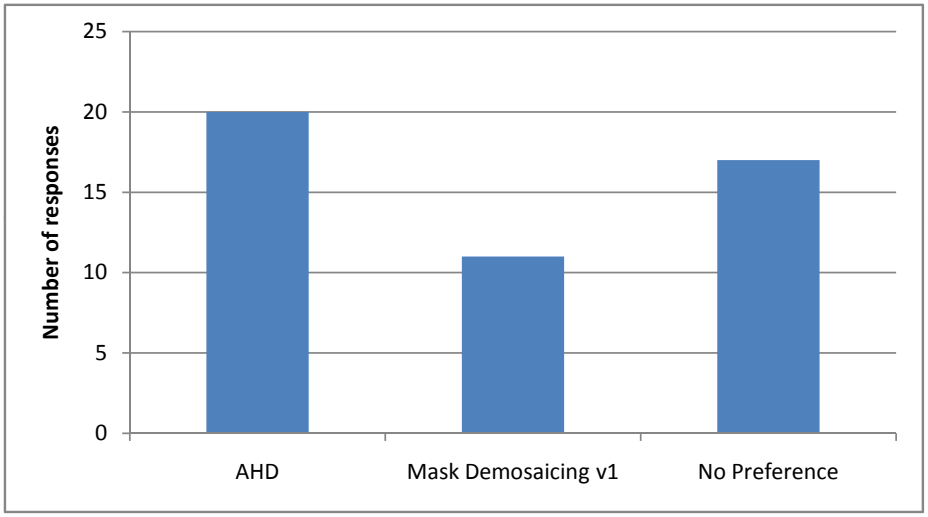


Figure 4.11: Results of the user study

This may be a side effect of bilinear interpolation. As bilinear interpolation works by averaging nearby colours, it may have the effect of blurring out highlights in the areas where it was applied. This is interesting, as previous studies of demosaicing methods[LDB02] have found that the subjects usually judge images by their sharpness. This suggests that Mask Demosaicing maintains quality in the edge regions compared to AHD.

The results show that perceptible differences between AHD and Mask Demosaicing are not due to typical demosaicing artifacts such as zippering or misguidance. Instead, any perceptible differences are due to subtle differences between bilinear interpolation and AHD in slowly varying parts of the images.

Chapter 5

Conclusion

The resolution of digital cameras is increasing all the time, current high-end models boast resolutions of over 20 megapixels. As the size of raw images increases, the performance of image processing techniques becomes more and more pertinent. The aim of this report has been to improve the performance the demosaicing methods. I have described two means of improving performance, through the use of the highly parallel architecture provided by the GPU, and by a novel demosaicing technique that concentrates computation only on areas where demosaicing is likely to fail.

I have shown how AHD demosaicing was implemented on the CUDA platform, and how the method was adapted to make best use of the GPU architecture. A performance evaluation of the GPU implementation showed a 30x speedup compared to a single-threaded CPU version.

Demosaicing is just one step in the image processing pipeline, other tasks include sharpening, denoising and colour-correction. The performance evaluation of bilinear interpolation showed that even relatively low computation tasks when applied to large data-sets achieve better performance on the GPU. Based on these results, and the performance improvements of demosaicing, it would be worthwhile to implement the entire image pipeline on the GPU, from demosaicing to the final image.

I also proposed a new demosaicing technique, Mask Demosaicing, that improves the performance of current demosaicing methods, without reducing image quality. I describe two variations of the method, one based on edge detection and one based on artifact detection.

The evaluation of a representative set of sample images showed that Mask Demosaicing with edge detection improves the performance of AHD by between 2.5x and 5x. However, the edge detection method's weakness is that it uses a fixed threshold, which can lead to failures in borderline areas in the image. According to Hirakawa and Parks[HP05], for the AHD algorithm the single greatest influence on image quality was the use of an adaptive threshold. The edge detection method would also likely benefit from an adaptive threshold, and the best method for this application is an interesting question.

The second variation of Mask Demosaicing used artifact detection. This method proved more robust than edge detection and produces results of almost equal quality to AHD. It offers a 2x to 4x speedup for low noise images. However, for images with a large amount of noise, the performance degrades until it is roughly equal to that of AHD.

The idea of applying specialised demosaicing routines to different regions in the image could be extended further in future work. For example, while AHD is the considered the best demosaicing method, it is biased toward straight lines and tends to artificially introduce edges into textured regions. A different demosaicing method, for example patterned pixel grouping[Lin03], would produce more accurate results in these kinds of areas. An extension to Mask Demosaicing might identify not just edges but other image features and apply the most appropriate demosaicing technique.

Until recently demosaicing algorithms have ignored or tended to underestimate the effect of noise in the raw image. However, noise can introduce errors during demosaicing which are exaggerated by further processing steps. Hirakawa and Parks[HP06] have proposed a method for simultaneously demosaicing and denoising, and other researchers are also working on the problem[LLHL06, ZWZ07, PFBK08]. The presence of the noise in the raw

images has impacted the effectiveness of some of the performance improvements proposed. As a certain degree of noise is unavoidable in real world data, it would be interesting to see how Mask Demosaicing could be made more robust in the face of noisy images.

Appendices

Appendix A

CIELAB colorspace

The CIE 1976 (L*, a*, b*) colour space[ICC06], CIELAB, was designed to represent all visible colours in a device independent manner. The L^* value represents luminance, it is in the range 0 to 100. The a^* value represents the position of the colour between red and green, and b^* represents the position of the colour between blue and yellow. The range of the two chrominance components is -128 to 128. The euclidean distance between two colours in CIELAB approximates their perceptual difference.

CIELAB colour space conversion

RGB values are converted to CIELAB via the standardised XYZ colour space using the following method:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{A.1}$$

$$L^* = \begin{cases} 116 \left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - 16 & \text{when } \frac{Y}{Y_n} > 0.008856 \\ 903.3 \left(\frac{Y}{Y_n}\right) & \text{otherwise} \end{cases} \tag{A.2}$$

$$a* = 500 \left(f\left(\frac{X}{X_n}\right)^{\frac{1}{3}} - f\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} \right) \tag{A.3}$$

$$b* = 200 \left(f\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - f\left(\frac{Z}{Z_n}\right)^{\frac{1}{3}} \right) \tag{A.4}$$

$$f(x) = \begin{cases} 7.787x + \frac{16}{116} & \text{when } x \leq 0.008856 \\ x & \text{otherwise} \end{cases} \tag{A.5}$$

where X_n, Y_n and Z_n are the normalised X, Y and Z values for the reference white point.

Bibliography

[Ada97] James E Adams. Design of practical color filter array interpolation algorithms for digital cameras. *Proc. SPIE*, 3028, 1997.

[AGLM93] L. Alvarez, F. Guichard, P.L. Lions, and J.M. Morel. Axioms and fundamental equations of image processing. *Archive for Rational Mechanics and Analysis*, 123(3):199–257, 1993.

[APS98] Jim Adams, Ken Parulski, and Kevin Spaulding. Color Processing in Digital Cameras. *IEEE Micro*, pages 20–30, 1998.

[ASH05] David Alleysson, Sabine Süsstrunk, and Jeanny Hérault. Linear demosaicing inspired by the human visual system. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 14(4):439–49, April 2005.

[Bay76] Bryce E. Bayer. *Color imaging array.* US Patent 3971065, July 1976.

[CCP99] E. Chang, S. Cheung, and D. Y. Pan. Color filter array recovery using a threshold-based variable number of gradients. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 3650 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 36–43, March 1999.

[Cok87] David R. Cok. *Signal processing method and apparatus for producing interpolated chrominance values in a sampled color image signal.* US Patent 4642678, February 1987.

[Cok94] D.R. Cok. Reconstruction of CCD images using template matching. In *Proc IS&T Annual Conf. / ICPS*, 1994.

[Cor10a] NVIDIA Corporation. *NVIDIA Compute PTX : Parallel Thread Execution ISA Version 2.0.* NVIDIA Corporation, 2.0 edition, January 2010.

[Cor10b] NVIDIA Corporation. *NVIDIA CUDA Best Practices Guide.* NVIDIA Corporation, 3.0 edition, April 2010.

[Cor10c] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 3.0.* NVIDIA Corporation, 3.0 edition, February 2010.

[Dev98] Nicolas Devillard. Fast median search : an ANSI C implementation, 1998.

[DLK78] P.L.P. Dillon, D.M. Lewis, and F.G. Kaspar. Color imaging system using a single CCD area array. *IEEE Transactions on Electron Devices*, 25(2):102–107, February 1978.

[Fre87] W.T. Freeman. *Method and apparatus for reconstructing missing color samples.* US Patent 4663655, May 1987.

[FZY09] Ivan Olaf Hernandez Fuentes, Miguel Enrique Bravo Zanoguera, and Guillermo Galaviz Yanez. FPGA implementation of the bilinear interpolation algorithm for image demosaicking. *Electronics, Communications, and Computers, International Conference on*, 0:25–28, 2009.

[GAM02] B K Gunturk, Y Altunbasak, and R M Mersereau. Color plane interpolation using alternating projections. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 11(9):997–1013, January 2002.

[GGA+] Bahadir K Gunturk, John Glotzbach, Yucel Altunbasak, Ronald W. Schafer, and Russel M. Mersereau. Demosaicking: Color Filter Array Interpolation. *IEEE Signal Processing Magazine*, (January 2005):44–54.

[GGAS05] B. K. Gunturk, J. Glotzbach, Y. Altunbasak, and R. W. Schafer. Demosaicking: color filter array interpolation. *IEEE Signal processing magazine*, 22:44–54, 2005.

[GLAAWV08] Jair Garcia-Lamont, Miguel Aleman-Arce, and Julio Waissman-Vilanova. A digital real time image demosaicking implementation for high definition video cameras. *Electronics, Robotics and Automotive Mechanics Conference*, 0:565–569, 2008.

[GW92] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley Pub, 3rd edition, 1992.

[Hib95] R.H. Hibbard. *Apparatus and method for adaptively interpolating a full color image utilizing luminance gradients*. US Patent 5,382,976, January 1995.

[HP05] Keigo Hirakawa and Thomas W. Parks. Adaptive homogeneity-directed demosaicing algorithm. *IEEE Trans. Image Processing*, 14:360–369, 2005.

[HP06] Keigo Hirakawa and TW Parks. Joint demosaicing and denoising. *IEEE Transactions on Image Processing*, 2006.

[HSS09] David J. Hardy, John E. Stone, and Klaus Schulten. Multilevel summation of electrostatic potentials using graphics processing units. *Parallel Computing*, 35(3):164 – 177, 2009. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[ICC06] Specification ICC.1:2004-10 (Profile version 4.2.0.0) Image technology colour management - Architecture, profile format, and data structure, 2006.

[JJ97] Hamilton J.F. Jr and Adams J.E. Jr. *Adaptive color plan interpolation in single sensor color electronic camera*. US Patent 5,629,73, May 1997.

[jr.]

[KB02] R Kakarala and Z. Baharav. Adaptive demosaicking with the principal vector method. *IEEE Trans. Consumer Electron.*, 48:932–937, November 2002.

[Ker09] Michael Kerrisk, editor. *Linux Programmer's Manual: Time - overview of time and timers*. 3.21 edition, 2009.

[KHO00] Oren Kapah and Hagit Z. Hel-Or. Demosaicking using artificial neural networks. In *Applications of Artificial Neural Networks in Image Processing V*, pages 112–120. SPIE, 2000.

[Kil08] Theodore Kilgore. A Practical Bayer Demosaicing Algorithm for Gphoto. In *IPCV*, pages 614–620. CSREA Press, 2008.

[Kim99] Ron Kimmel. Demosaicing: Image Reconstruction from Color CCD Samples. *IEEE Trans. Image Processing*, 8:1221—-1228, 1999.

[KTK98] J.E. Adams K. Topfer and B.W. Keelan. Modulation transfer functions and aliasing patterns of CFA interpolation algorithms. In *IS&T PICS Conference*, pages 367–370, 1998.

[LDB02] P. Longere, P.B. Delahunt, and D.H. Brainard. Perceptual assessment of demosaicing algorithm performance. *Proceedings of the IEEE*, 90(1):123–132, 2002.

[LH02] R.F. Lyon and P.M. Hubel. Eyeing the camera: into the next century. In *Proc. IS&T/TSID 10th Color Imaging Conf.*, pages 349–355, 2002.

[Lin03] Chuan-kai Lin. Pixel Grouping for Color Filter Array Demosaicing, 2003.

[LLHL06] Hung-Yi Lo1, Tsung-Nan Lin, Chih-Lung Hsu2, and Cheng-Hsien Lee. Directional weighting-based demosaicking algorithm for noisy cfa environments. *IEEE Transactions on Circuits and Systems*, pages 489–492, 2006.

[LP94] C.A. Laroche and M.A. Prescott. *Apparatus and method for adaptively interpolating a full color image utilizing chrominance gradients*, volume 5373. US Patent 5,373,322, December 1994.

[LT03] Wenmiao Lu and Yap-Peng Tan. Color filter array demosaicking: new method and performance measures. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 12(10):1194–210, January 2003.

[LZW03] Anat Levin, A. Zomet, and Y. Weiss. Learning to perceive transparency from the statistics of natural scenes. *Advances in Neural Information Processing Systems*, 1(1):1271–1278, 2003.

[OF96] Bruno A. Olshausen and David J. Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

[OLG+07] J.D. Owens, David Luebke, Naga Govindaraju, Mark Harris, J. Kruger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[PFBK08] Dmitriy Paliy, Alessandro Foi, Radu Bilcu, and Vladimir Katkovnik. Denoising and interpolation of noisy Bayer data

with adaptive cross-color filters. *Proceedings of SPIE*, pages 68221K–68221K–13, 2008.

[PTAB10] Harold Phelippeau, H. Talbot, M. Akil, and S. Bara. Green Edge Directed Demosaicing Algorithm. *Laboratoire d'informatique, Gaspard-Monge Internal Report*, 2010.

[RGC+09] Nicolas Robidoux, Minglun Gong, John Cupitt, Adam Turcotte, and Kirk Martinez. CPU, SMP and GPU implementations of Nohalo level 1, a fast co-convex antialiasing image resampler. In *C3S2E '09: Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering*, pages 185–195, New York, NY, USA, 2009. ACM.

[RRB+08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming - PPoPP '08*, pages 73–82, New York, New York, USA, 2008. ACM Press.

[RSBS02] Rajeev Ramanath, Wesley E. Snyder, Griff L. Bilbro, and William A. Sander. Demosaicking methods for Bayer color arrays. *Journal of Electronic Imaging*, 11(3):306, 2002.

[SF73] I. Sobel and G. Feldman. *A 3x3 isotropic gradient operator for image processing*, pages 271–272. John Wiley and Sons, 1973.

[Smi96] J.L. Smith. Implementing median filters in XC4000E FPGAs. *Xcell: The Quarterly Journal for Xilinx Programmable Logic Users*, 23, 1996.

[SPF+07] John E Stone, James C Phillips, Peter L Freddolino, David J Hardy, Leonardo G Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of computational chemistry*, 28(16):2618–40, December 2007.

[WB09] Zhou Wang and A.C. Bovik. Mean squared error: love it or leave it? A new look at signal fidelity measures. *IEEE Signal Processing Magazine*, 26(1):98–117, 2009.

[Wel89] J. A. Weldy. Optimized design for a single-sensor color electronic camera system. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 1071 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, May 1989.

[Woo05] Cliff Wootton. *A practical guide to video and audio compression: from sprockets and rasters to macroblocks*. Focal Press, 2005.

[WS00] G. Wyszecki and WS Stiles. *Color science: concepts and methods, quantitative data and formulae*. Wiley-Interscience, 2000.

[ZWZ07] Lei Zhang, Xiaolin Wu, and David Zhang. Color reproduction from noisy CFA data of single sensor digital cameras. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 16(9):2184–97, September 2007.

Ω