An evaluation of debayering algorithms on GPU for real-time panoramic video recording

Ragnar Langseth, Vamsidhar Reddy Gaddam, Håkon Kvale Stensland,
Carsten Griwodz, Pål Halvorsen
University of Oslo / Simula Research Laboratory

***Abstract*—Modern video cameras normally only capture a single color per pixel, commonly arranged in a Bayer pattern. This means that we must restore the missing color channels in the image or the video frame in post-processing, a process referred to as debayering. In a live video scenario, this operation must be performed efficiently in order to output each frame in real-time, while also yielding acceptable visual quality. Here, we evaluate debayering algorithms implemented on a GPU for real-time panoramic video recordings using multiple 2K-resolution cameras.**

***Keywords*-Debayering, demosaicking, panorama, real-time, GPU, CUDA**

I. Introduction

Most modern video cameras sample only a single color per pixel. The photosensors used for capturing the image only record the lights intensity, and color is determined by first passing the light through a wavelength filter. Instead of capturing multiple colors in every pixel, a color filter array is used to create a consistent pattern of colored pixels.

The most common color filter array used today is the *Bayer filter* [2], where each pixel is either red, green or blue. To obtain a multi-color image from this pattern, we must interpolate the missing values, a process referred to as *debayering*. Cameras can perform a hardware conversion in the device, limited to a simple non-adaptive algorithm. In our real-time panorama video system [5], we use multiple industrial cameras with a gigabit ethernet interface. With 2K resolution, a full rate data stream in the RGB color model is limited by bandwidth to about 18 fps. We deem this too low, which means that we need an efficient algorithm for debayering several high resolution video streams, while also providing a good visual result. In a moving scene, artifacts as a result of the debayering process will rarely persist across enough frames to be visible. As a result, most video systems can make due with a simple algorithm. However, the scene in our target application is primarily static background, where artifacts remain consistent over time.

Our panorama video processing pipeline is installed in a Norwegian elite soccer club stadium and is intended for live streaming and immediate video access, thus requiring real-time performance. In this paper, we therefore evaluate different debayering algorithms for real-time panoramic video recording using a statically placed camera array. To offload the main CPU and improve performance, we have used GPUs to accelerate the processing, and we here evaluate both processing overhead and visual quality. Our experimental results show that there is a trade-off between quality and execution time.

The rest of the paper is organized as follows: Next, in section II, we briefly describe our system. Then, we present our selected algorithms in section III and detail their implementations in section IV. Section V shows our experimental results, which we discuss further in section VI before we conclude the paper in section VII.

II. System Overview

In [5], we described our panorama pipeline. Here, we record raw 2040×1080 video frames in bayer format from five cameras at 50 fps, and each of these camera streams must be debayered in real-time.

Modern GPU's can provide significantly better performance than a CPU for certain tasks. They are optimized for applying small transformations to every single pixel or texture element, with hundreds or thousands of threads performing the same task in parallel, with minimal inter-thread communication. Debayering is an inherently parallel operation, as each pixel, or block of pixels, can typically be calculated locally. Hence, with our high data rate and real-time requirements, we found the GPU to be better suited to perform this task. However, more complex algorithms that require a greater level of contextual information about the entire image will not achieve the same performance increase.

In this system, we utilize the Nvidia CUDA framework and have focused on the Kepler architecture [13]. CUDA was selected, as opposed to for instance OpenCL, for achieving the best performance on a target architecture. Given that our system is intended as a server and content provider, efficiency is prioritized over portability. In our implementations, we have also prioritized execution speed over memory requirements.

III. Debayering algorithms

There exist several debayering algorithms in literature, e.g., [7], [1], [6], [3], [10], [11], [12], [9]. However, not every algorithm is well suited to the GPU architecture or real-time processing. The GPU is most efficient when each

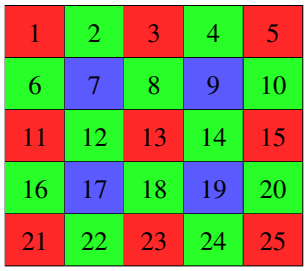


Figure 1: An example bayer pattern

pixel is computed the same way, without depending on the values or result of other pixels. More complex algorithms will adapt based on neighbouring pixels to reduce visual artifacts, which usually creates inter-pixel dependencies.

Below, we have selected algorithms that we deemed most promising, considering our high data rate and real-time requirements. When explaining the different algorithms, we will be referring to figure 1 in example equations, identifying each pixel with a number and each color value with R(ed), G(reen) or B(lue).

A. Bilinear interpolation

Bilinear interpolation uses the average value of the two or four nearest neighbour pixels of the specific color, e.g., the blue values for pixel 8 and pixel 13 are found by

$$B8 = \frac{B7 + B9}{2} \qquad B13 = \frac{B7 + B9 + B17 + B19}{4}$$

It is generally considered the cheapest of the acceptable algorithms, often used in real-time video systems due to its low complexity. Therefore, we have included this as our baseline algorithm.

B. Smooth hue transition

Smooth hue transition is a two pass algorithm [4] that first uses the bilinear interpolation described above to reconstruct the green channel. Then, a second pass uses the relation between the green channel and the red/blue channel within a pixel to reconstruct the remaining channels. For example, the blue value in pixel 13 is calculated thus

$$B13 = \frac{G13}{4}\left(\frac{B7}{G7} + \frac{B9}{G9} + \frac{B17}{G17} + \frac{B19}{G19}\right)$$

This utilizes the principle that the difference between two channels within a pixel only changes gradually and rapid transitions will cause visual artifacts.

C. High-quality linear interpolation

High-quality linear interpolation is a single pass algorithm [11] that performs bilinear interpolation, but uses the color information already present in the pixel to correct the result, e.g.,

$$\Delta r = R13 - \frac{R3 + R11 + R15 + R23}{4}$$
$$G13 = \frac{G8 + G12 + G14 + G18}{4} + \frac{\Delta r}{2}$$

If the interpolated red value differs significantly from the real red value, there is likely a significant change in luminosity in this pixel.

D. Edge directed interpolation

Edge directed interpolation is a two pass algorithm [1] that tries to avoid interpolating across edges, averaging two widely different values. It uses the laplacian, i.e., the divergence of the gradient between enclosing pixels, of the green channel and the gradient of the red or blue channel to determine the presence of an edge when reconstructing the green channel. The horizontal gradient is determined by

$$Grad13_H = |G12 - G14| + |2R13 - R11 - R15|$$

and the vertical gradient is calculated similarly. The algorithm performs linear interpolation of either the two enclosing vertical samples, or horizontal, depending on the smallest gradient. When interpolating the red and blue channel, it performs linear interpolation of the pixel hue in the nearby samples. We mentioned that hue, i.e., the difference between two color channels, changes gradually, but luminosity may transition rapidly from one pixel to the next. Since the green channel carries most of the luminosity information, we use the difference between the, already interpolated, green channel and the missing red/blue channel for a better estimation, as such

$$R13 = G13 + \frac{(G7 - B7) + (G9 - B9) + (G17 - B17) + (G19 - B19)}{4}$$

Here, it is implied that we trust the correctness of the initial green interpolation. This approach is also used by the next algorithm.

E. Homogeneous edge directed interpolation

Homogeneous edge directed interpolation is a three pass algorithm, designed as a simplification of the adaptive homogeneity directed demosaicking algorithm [6]. When interpolating in only one direction, it may be visually apparent if single pixels choose a different direction compared to neighbouring pixels. This algorithm therefore computes the directional gradients in the first pass, before selecting the direction based on the local directional preference in a second pass. The final pass for interpolating the red and blue channels is equal to that of the edge directed.

F. Weighted directional gradients

Weighted directional gradients is a two pass algorithm [10] that uses a weighted average of pixels in four directions in the initial green interpolation, weighted based on the inverse gradient in its direction. The algorithm determines the value contribution G of each direction right/left-/up/down, and its weight α. For example, the right direction of pixel 7 is determined by

$$G_r = G8 + \frac{B7 - B9}{2}$$

$$\alpha_r = \frac{1}{|G6 - G8| + |G8 - G10| + |B7 - B9| + \frac{|G2-G4|+|G12-G14|}{2}}$$

similar for each direction. The final green value can be computed by

$$G7 = \frac{\alpha_l G_l + \alpha_r G_r + \alpha_u G_u + \alpha_d G_d}{\alpha_l + \alpha_r + \alpha_u + \alpha_d}$$

This is performed similarly when interpolating the red and blue channel, while then also taking advantage of having the full green channel. It performs a similar directional weighted average independently for each channel.

IV. Implementations

We have implemented the algorithms described in the previous section, optimizing for execution speed, not memory requirements. This section assumes a basic knowledge of the CUDA architecture and terminology.

The algorithms are all implemented using the same base principles, as they are primarily differentiated by the number of required passes and the number of pixel lookups per pass. Every kernel is executed with 128 threads per CUDA block, the minimum required to allow for maximum occupancy on the Kepler architecture. Every active kernel is also able to achieve more than 95% occupancy since debayering is a highly parallelizable workload. The initial bayer image is bound to a two dimensional texture, giving us the benefit of two dimensional caching when performing multiple texture lookups per pixel. The use of textures is essential, as many of the algorithms would be difficult to implement with good memory coalescing using the GPU's global memory.

This could also have been accomplished using shared memory, but it would be harder to coordinate and more device specific. In order to accommodate the ideal 128 threads per block for maximum occupancy, using a 5×5 pixel lookup grid, we would need to load a total of $5 \times (128 \times 2 + 4) = 1284$ bytes per block. This becomes problematic when crossing image row boundaries, and may prove difficult to optimize for most horizontal resolutions. We believe that the quality of caching from using a single texture is more beneficial, and produces a better result than shared memory. By opting to not use shared memory, we could also have utilized a larger general cache, as this typically uses the same memory pool, though in our pipeline we need shared memory available for other modules.

Many of the algorithms require multiple passes, most commonly an initial green pass followed by one red and blue pass. The initial green pass is implemented similarly across algorithms, using a temporary texture with two bytes per pixel, for saving the green value and either a red, blue or empty value. Using a single texture for this provides better data locality and cache efficiency, increasing performance over using two separate textures. In order to write the temporary values, we use surface memory, which utilizes the same 2D caching as textures.

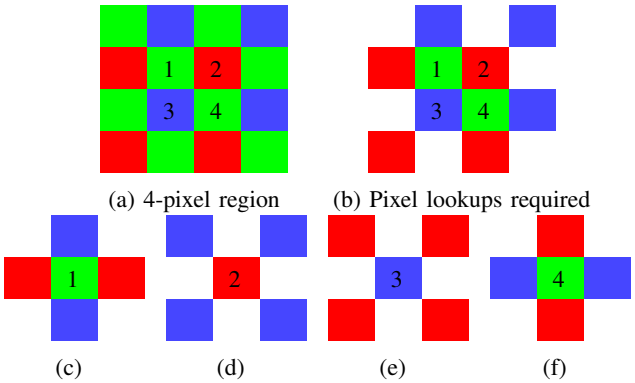
The homogeneous edge directed algorithm uses two passes to interpolate the green channel. In the first pass, the green value is computed both based on the horizontal and the vertical interpolation method. Additionally, we calculate the directional preference. These values, along with the original red/blue value are written to surface memory with 4 bytes per pixel. It proved faster to keep this data localized in one array, despite having to perform nine texture lookups when we determine the localized directional homogeneity.

The original weighted directional gradients uses two passes to interpolate the red and blue channels. The second pass fully interpolates the red and blue pixels, leaving the green pixels untouched. This data is then used in the third pass to complete the remaining red and blue values. This implementation uses a full four bytes per pixel to ensure data locality for the final pass, but this may not be ideal. It is generally considered more efficient to use four bytes per pixel instead of three, due to memory alignment, but in our case, we have only half the pixels carrying three values and the other half (green pixels) carrying a single value. We opted to implement two variations of this algorithm, the original and a modified version that borrows the constant hue correction-based approach of the edge directed algorithms.

When implementing the kernels it was essential to avoid branching code, based on the color of each pixel. A naive approach would run the kernel on each pixel and perform one of four branches, depending on the color of that pixel. Because each branching operation within a single thread warp must be executed by all threads in that warp, it would be guaranteed that at least half of the executing threads would idle due to branching. Instead, our kernels process 2×2 pixels in each iteration. This introduces zero branching as a result of selecting pixels. These four pixels will also need to access a lot of the same pixels, so we load these at once for local computations.

We tried two different implementations for the final pass of every algorithm. Initially, each pixel was calculated separately, performing all pixel lookups required. However, we ensured that we always processed two pixels consecutively. The kernel would first determine if the two pixels are of a green/red row, or a blue/green row. This evaluation would always yield the same branch within a warp, except for warps that crossed row boundaries. With our target horizontal resolution of 2040, this meant that only one warp out of $\frac{2040pixels}{32 \times 2pixels} = 31.86$ would encounter branching. Most common image resolutions are divisible by 32, meaning that this would yield zero branching in these situations.

However, we saw that the previous kernel usually covered a lot of overlapping pixels. Figure 2 shows an example of overlapping pixel lookups, highlighting a four-pixel region and the required lookups for each individual pixel. In the edge directed final pass, each pixel must perform five



(a) 4-pixel region (b) Pixel lookups required

(c) (d) (e) (f)

Figure 2: Visualization of 4-pixel kernel implementation, using edge directed as an example. The four pixels to calculate are numbered in (a), while (b) shows which of the surrounding pixels must be read. Figures (c, d, e, f) show the individual lookups required. Note that all red/blue pixels also contain a green value, previously calculated.

lookups. However, we can see that if we interpolate these four pixels together and use temporary storage, we only need a total of ten lookups. For other algorithms that require more pixel lookups, this overlap is even greater. Performing a texture lookup requires multiple cycles, depending on cache efficiency and access pattern, while accessing local registers is only a single cycle. Therefore, we interpolate four pixels at the same time, covering the four possible branches, and perform as few texture lookups as possible, relying on temporary storage. The exception is the original weighted directions, where this increased the local register requirements for each thread too much, reducing the number of concurrent threads that could execute. Instead, we observed better results when performing duplicate texture lookups.

V. Experimental results

To compare the different algorithms and implementations, we have performed a number of experiments that we will present in this section.

A. Visual quality

We performed a simple evaluation on the visual quality of each algorithm by subsampling existing images, imposing the bayer pattern, to see how accurately the image can be reconstructed. When reconstructing the images, we typically see interpolation artifacts, primarily in the form of false colors and zippering along edges. Figure 3 shows how each algorithm handles a region prone to false colors. We see that all algorithms produce interpolation artifacts, though with varying frequency and intensity.

Peak signal-to-noise ratio (PSNR) is a simple metric for evaluating image reconstruction. We computed the PSNR of each of the reconstructed images, filtering away homogeneous areas that rarely produce visible errors with an edge detection filter. The result can be found in table I. Although PSNR can yield inconsistent results with many image transformations, we observed a very strong correlation

Algorithm	PSNR	
	Green	Red/blue
Bilinear	28.43	23.51
Smooth hue transition	28.43	27.07
High-quality linear	34.44	29.67
Edge directed	35.61	34.62
Homogeneous edge directed	36.22	34.89
Weighted directions original	37.97	31.02
Weighted directions modified	37.97	36.25

Table I: Measured PSNR for the Lighthouse image [8].

Algorithm	Execution time (ms)	
	Mean	Std-div
Bilinear	40.71	0.39
High-quality linear	137.59	0.92
Edge directed	110.76	0.72

Table II: CPU execution time for three algorithms (run 1000 times) on a 2040×5400 pixel image, simulating our panorama system.

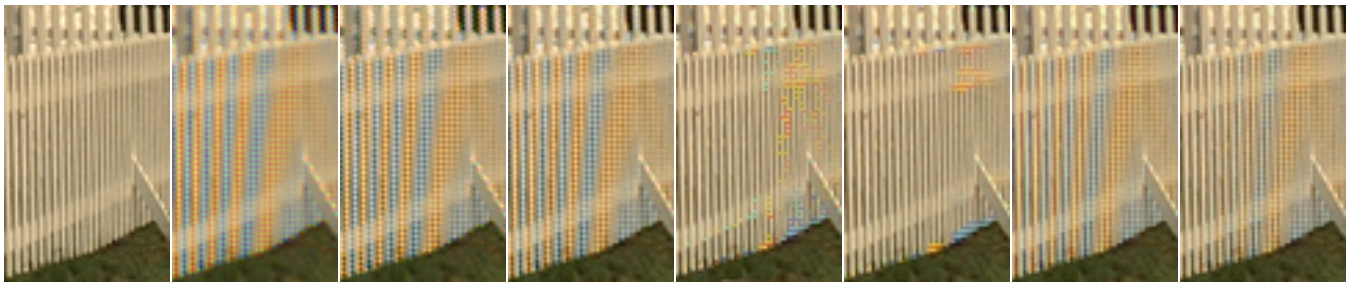
between the PSNR and the visual result of figure 3. High PSNR in the green channel was common in those algorithms that avoided zippering artifacts and maintained the greatest level of detail. Low PSNR in the red and blue channel normally meant a great level of false colors. We could see that the best performing algorithms all use the same, simple final pass for interpolating the blue and red channels. This shows that if the green channel is accurately reconstructed, we can use the concept of constant hue to reconstruct the more sparsely sampled channels.

B. Execution performance

The primary requirement in our real-time panorama system is the overall execution time of the algorithm. The algorithms presented have a varying degree of computational complexity, but this is not necessarily the only requirement for performance efficiency. In order to determine a baseline, table II shows the execution time of a few of the algorithms implemented on CPU. Note that general optimizations were applied, but no threads or SIMD instructions were utilized. We see that these implementations are far below the real-time limit of 20ms per frame of our system. Multiple threads could be used, but this still takes away valuable resources from the rest of our system, primarily the video encoding.

Table III shows the mean execution time of each algorithm on both a high-end and a mid-range GPU, along with some simple complexity metrics unique to our implementations. If we compare to the CPU implementations in table II, we see that every algorithm is significantly faster. However, the high quality linear is now a lot faster than the edge directed, which shows that the GPU is a very different architecture. Only mean execution times have been included since the GPU provides extremely consistent numbers, with a negligible standard deviation. It is worth noting that these numbers include a conversion into the YUV colorspace in their final pass, required by the rest of our system.

We see that most algorithms are nearly equally fast, and within the 20 ms real-time limit of 50 fps, on a GeForce GTX 680. The original weighted directions proved extremely inefficient, due to its slow red and blue channel interpolation. However, we observed that we could achieve



(a) Original (b) Bilinear (c) Smooth hue transition (d) High-quality linear (e) Edge directed (f) Homogeneous edge directed (g) Weighted directions original (h) Weighted directions modified

Figure 3: Visual assessment of zippering and false colors in an error prone region

a better visual result by exchanging these two passes with the final pass of the edge directed algorithm, at a lower processing cost. Overall, the execution time seems most affected by the number of texture lookups, with an added penalty for each pass. An exception appears to be the second pass of the smooth hue transition, which is slowed down by having to perform four division operations per pixel. Here, we also utilize CUDAs native division approximation function, which uses only half the cycles of a full division. This reduced the execution of this pass by 20%, and because we are working with 8-bit color values the loss of precision never affected the final result.

We described two different kernel implementations for the final pass of each algorithm, a 2-pixel variant where each pixel was calculated separately and a 4-pixel variant where we reduced the number of texture lookups. In table III, it is quite apparent that the original weighted directions was not using this optimization, based on its disproportionate number of texture lookups. In figure 4, we compare the implementations of these two approaches. Here, we see that the 4-pixel variant is superior for all implementations. The smooth hue transition performed exceptionally better with the 2-pixel variant, primarily because it's alternative implementation required too many registers and was limited to 67% occupancy. Additionally, we see a correlation with the number of bytes to look up and the benefit of the 2-pixel kernel. We believe this is why the high-quality linear and smooth hue transition algorithms achieved the highest performance gain.

We also experimented with how many threads to launch. We have mentioned that each kernel will compute 2 or 4 pixels sequentially, but each thread can also iterate through multiple pixel-blocks in its lifetime. One can either launch very many threads, with short lifetimes, or fewer threads that compute more pixels. Note that "few" threads in this context is still several thousand. For the 2-pixel kernels we consistently saw best results when each thread computed 32 pixels, i.e. iterate 16 times. The 4-pixel kernels performed best when computing only 8 pixels each, iterating twice. These numbers may be very device specific, and should be determined dynamically based on the architecture and resolution of the images.

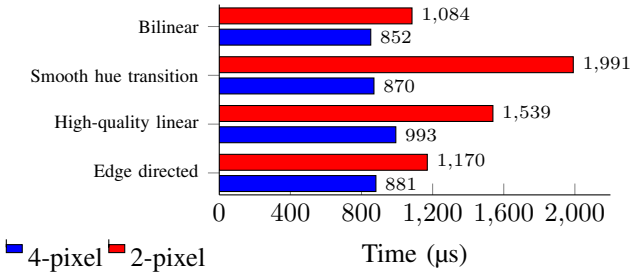


Figure 4: Performance evaluation of the final pass kernels, using the 4-pixel variant and the 2-pixel variant. Note that the modified weighted directional gradients and homogeneous edge directed algorithm also use the edge directed kernel for their final pass.

VI. Discussion

In the previous section, we saw that most algorithms were well below our real-time threshold on a GeForce GTX 680. The original weighted directions proved extremely inefficient, due to its slow red and blue channel interpolation. However, we saw that by changing the final pass of the algorithm we could achieve better visual results at only a third of the execution time. This shows that if the green channel is accurately reconstructed, we can use the concept of constant hue to reconstruct the more sparsely sampled channels. The algorithms that utilized this method, differing only by the initial green interpolation, performed best visually. The execution time seems to be primarily determined by the number of texture lookups required, with an added penalty for each pass. The exception to this is the smooth hue transition algorithm, which has a very quick green pass compared to the others.

When performing debayering on the five images, we treat them as a single image. The edges of the images are never visible in the final panorama. This allows us to launch fewer kernels, with less time spent on synchronization and kernel scheduling. We also perform no edge handling, as CUDA textures clamp reference lookups outside image boundaries. This causes odd colors around image borders, but these are removed when we stitch the images into a panorama.

In our panorama pipeline, the debayering of all images is performed on a single GPU, despite using multiple recording

Algorithm	Execution time (µs)		µs / pass (GTX680)			Lookups / 2×2			Temporary memory
	Quadro K2000	GTX680	$1st$	$2nd$	$3rd$	$1st$	$2nd$	$3rd$	
Bilinear	5516	929	852			14×1			
Smooth hue transition	10183	1979	747	1102		12×1	14×2		$2 \times x \times y$
High-quality linear	6370	1073	993			24×1			
Edge directed	10941	2025	987	881		20×1	10×2		$2 \times x \times y$
Homogeneous edge directed	20029	3184	1106	1032	905	18×1	18×4	10×2	$4 \times x \times y$
Weighted directions original	49212	9061	2112	2087	4696	76×1	92×2	124×4	$6 \times x \times y$
Weighted directions modified	19052	3094	2050	894		76×1	10×2		$2 \times x \times y$

Table III: Summary of each algorithms resource requirements. Execution was measured with a 2040×5400 resolution image. Note that, in addition to each pass, some CPU overhead is required for preparing buffers and launching kernels. For each pass, we show the number of texture lookups per 2×2 pixels, i.e., 2 green, 1 blue & 1 red, of either 1, 2 or 4 bytes each.

machines. This could be offloaded to the recording machines to free up resources. However, this would mean having to transfer all the color channels between machines. This would require three times the bandwidth, and the following module would be slowed down by not having 4-byte aligned pixel addressing. Our current setup deals with only 5 cameras, but if the system is extended to include more cameras such offloading would enable use of distributed resources more efficiently. The debayering module does take away resources from the remaining panorama pipeline. Therefore, even though all algorithms run far below real-time limits, we may opt to use a faster algorithm if the system is extended.

VII. Conclusion

In this paper, we have looked at debayering algorithms for real-time recording of panorama videos. We have modified and implemented several algorithms from the literature onto GPUs and evaluated both the real-time capabilities and the visual quality. Many of the algorithms are viable, yielding a tradeoff between quality and run-time. Every algorithm was capable of maintaining real-time constraints, but some proved inefficient compared to the resulting visual quality, such as the original weighted directions and the smooth hue transition. However, many of the visual artifacts were significantly reduced by the video encoding step in our panorama pipeline, or made invisible by the high framerate. This means that, in our system, the intensity of the false colors were often more important than the frequency. This made the weighted directions a very good choice, as it had the least intensive false colors. We also found that the mazing artifacts created by the edge directed algorithm were rarely as visually apparent after the encoding process. These two algorithms both perform very well in homogeneous areas, and provide only minor false colors around white lines. Therefore, we saw the best tradeoff between quality and runtime with the edge directed algorithm, the faster alternative of the two.

Acknowledgements

This work has been performed in the context of the *iAD* centre for Research-based Innovation (project number 174867) funded by the Norwegian Research Council.

References

[1] J. Adams. Design of practical color filter array interpolation algorithms for digital cameras .2. In *Proc. of IEEE ICIP*, volume 1, pages 488–492, Oct 1998.

[2] B. Bayer. Color imaging array, July 1976. US Patent 3,971,065.

[3] E. Chang, S. Cheung, and D. Y. Pan. Color filter array recovery using a threshold-based variable number of gradients. volume 3650, pages 36–43, 1999.

[4] D. Cok. Signal processing method and apparatus for producing interpolated chrominance values in a sampled color image signal, Feb. 1987. US Patent 4,642,678.

[5] V. R. Gaddam, R. Langseth, S. Ljødal, P. Gurdjos, V. Charvillat, C. Griwodz, and P. Halvorsen. Interactive zoom and panning from live panoramic video. In *Proc. of ACM NOSSDAV*, pages 19:19–19:24, 2014.

[6] K. Hirakawa and T. W. Parks. Adaptive homogeneity-directed demosaicing algorithm. *IEEE Transactions on Image Processing*, 14(3):360–369, 2005.

[7] R. Kimmel. Demosaicing: image reconstruction from color ccd samples. *IEEE Transactions on Image Processing*, 8(9):1221 – 1228, 1999.

[8] Kodak. Kodak lossless true color suite. http://r0k.us/graphics/kodak/.

[9] B. Leung, G. Jeon, and E. Dubois. Least-squares luma-chroma demultiplexing algorithm for bayer demosaicking. *IEEE Transactions on Image Processing*, 20(7):1885–1894, July 2011.

[10] W. Lu and Y.-P. Tan. Color filter array demosaicking: new method and performance measures. *IEEE Transactions on Image Processing*, 12(10):1194–1210, Oct 2003.

[11] H. S. Malvar, L. wei He, and R. Cutler. High-quality linear interpolation for demosaicing of bayer-patterned color images. In *Proc. of IEEE ICASSIP*, 2004.

[12] D. Menon, S. Andriani, and G. Calvagno. Demosaicing with directional filtering and a posteriori decision. *IEEE Transactions on Image Processing*, 16(1):132–141, 2007.

[13] Nvidia. Kepler tuning guide. http://docs.nvidia.com/cuda/kepler-tuning-guide/, 2013.