

Efficient, High-Quality Bayer Demosaic Filtering on GPUs

Morgan McGuire*
Williams College

Abstract

This paper describes a series of optimizations for implementing the high-quality Malvar-He-Cutler Bayer demosaicing filter on a GPU in OpenGL. Applying this filter is the first step in most video processing pipelines, but is generally considered too slow for real-time on a CPU. The optimized implementation contains 66% fewer ALU operations than a direct GPU implementation and can filter 40 simultaneous HD 1080p video streams at 30 fps (2728 Mpix/s) on current hardware. It is 2-3 times faster than a straightforward GPU implementation of the same algorithm on many GPUs. Most of the optimizations are applicable to other kinds of processors that support SIMD instructions, like CPUs and DSPs.

1 Introduction

1.1 Demosaicing

Real-time color video is an essential part of most computational videography pipelines, from machine vision and military surveillance to consumer cell-phone video and videoconferencing. Most color cameras don't capture RGB values at each pixel. They contain a monochrome sensor overlaid with a mosaiced optical filter designed by Bryce Bayer, like the one shown in figure 1. Each pixel in a Bayer camera measures only one color channel. The two leftmost-images in figure 2 show detail views of a ground truth color image and the raw values (depicted in grayscale) that a Bayer camera would capture when imaging the same scene. The raw values are what would be processed on a CPU or uploaded to a GPU as a grayscale or luminance-only texture.

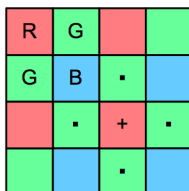


Figure 1: Diagram of the Bayer optical filter with the first red pixel at (0, 0). Not shown are the three other variations, with the first red pixel at (0, 1), (1,0), and (1,1) respectively.

Bayer demosaicing is the process of digitally filtering a raw frame of video to reconstruct an image with R, G, and B values at each pixel. There are many demosaicing algorithms, the results of some of which are shown in figure 2. A toy *label* algorithm shows only the known color value at each pixel. This is useful for understanding how the Bayer mosaic works but is poor as a reconstruction: it has only 1/3 the intensity of the original, degenerates into a dot pattern unless viewed from very far away, and shifts color edges by one pixel. Due to dithering algorithms in LCD monitors, this picture can't even be shown on most displays.

The simplest reasonable reconstruction filter replaces the missing two color channels at each pixel with those from its *nearest neighbors*. This has the correct intensity but produces a 1-pixel shift of hue edges and results in false colors along value edges. Using *bilinear* interpolation of neighboring pixels corrects the shift and improves edge color but blurs high frequencies. There are many more sophisticated reconstruction filters. One of the best is by *Malvar, He, and Cutler* [?]. They assumed that hue changes more slowly than value in an image. It follows that adjacent raw pixel values should therefore be correlated even though they arise from different color channels. From this insight, Malvar et al. solved for the optimal 5×5 linear filter kernels to reconstruct all color channels. On a standard benchmark their reconstruction is 5.5 dB better than bilinear in PSNR and outperforms even non-linear methods. Compare the roofline and bricks in the right-most image of figure 2 to those areas under other reconstruction filters.

*e-mail: morgan@cs.williams.edu

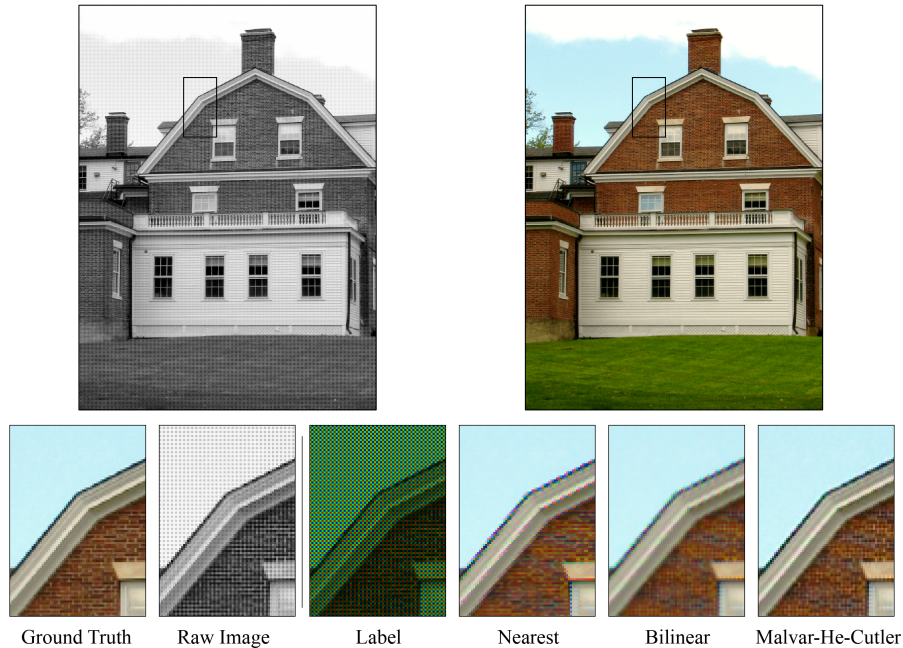


Figure 2: Comparison of common demosaicing methods applied to an image of a brick house and blue sky. Note reconstruction artifacts at the high-frequency bricks and along the roofline. I produced the ground truth image by filtering down a very high-resolution image (that was itself demosaiced), and then discarded two values at each pixel to create a simulated raw image.

1.2 GPU Filters

The Malvar-He-Cutler algorithm is the first step in many video processing pipelines. Unfortunately, most CPU implementations cannot apply the 5×5 filters in real-time at 640×480 resolution, so live video processing tends to use the inferior nearest-neighbor or bilinear methods. A direct GPU implementation of Malvar-He-Cutler using the OpenGL Shading Language (GLSL) processes 166 Mpix/s on an NVIDIA GeForce 8600M GT GPU, which is the GPU of a typical graphics laptop today (e.g., the 2007 Apple MacBook Pro). That is sufficient for reconstructing a HD 1080p (1920×1080) video stream at 60 fps, with some overhead left over for normal UI rendering.

This paper describes how to cut 2/3 of the arithmetic operation count and increase the performance of Malvar-He-Cutler to 280 Mpix/s on a GPU. Why care about demosaicing more pixels than visible on the screen, faster than real-time? Because demosaicing is the “ray-triangle intersection” of video processing. It is the keystone supporting the structure of building blocks that form a video processing algorithm. Demosaicing must be as fast as possible in order to decrease the overhead applied to more complex algorithms, and it must be fast enough for multiple simultaneous video streams to support many computational photography applications. This implementation can demosaic two high-definition streams *and* estimate the stereo depth disparity from them at 60 fps on the previously mentioned laptop, for example.

It is likely possible to process video in GLSL slightly faster than the numbers quoted here by optimizing for that specific GPU. However, truly optimal implementations are too dependent on the architecture of a specific processor to be useful for more than a few months. So I instead focus on algorithmic changes that are relevant to a broad class of architectures that will likely be relevant for many years, including modern CPUs with SIMD instructions.

The efficient implementation is based on three kinds of optimizations: symmetry, vectorization, and use of rasterizer interpolators (OpenGL “varying” variables). I assume a target architecture with most of the following properties:

1. Register-to-register and constant-to-register moves (MOV) have negligible cost.
2. Conditional assignment (CMOV) is significantly faster than branch (CJMP).

3. ALU operations can swizzle, mask and negate inputs at zero cost.
4. Fused multiply-add (MADD) has approximately the same cost as multiplication.
5. ADD is at least as fast as MADD.

This list broadly describe most modern GPUs and CPUs. Furthermore, on architectures with the following additional properties additional optimizations are possible.

6. 4-way SIMD MADD and ADD have the same cost as scalar equivalents.
7. Interpolators are present in fixed function rasterizer, so using them has zero time cost.

The implementation is designed so that most optimizations at least do no harm when the target architecture does not satisfy all of the above properties. For example, no current CPU contains a rasterizer (although Intel's proposed Larrabee architecture could be considered a CPU + rasterizer), but all of the CMOV and SIMD optimizations are still applicable on CPUs. Some GPUs perform only scalar operations, but vector operations on those are no slower than the equivalent number of scalar operations and the vectorization in this paper avoids introducing unused vector components for padding.

The optimized implementation is given in OpenGL shader code at the end of this article. It can be applied without understanding the derivation. The derivation is interesting because the same kinds of high-level pattern manipulation and low-level arithmetic optimization are likely applicable to other video filtering problems. The explanation of the implementation it is complicated by the number of cases involved: four Bayer pattern conventions \times five filter kernels \times three color channels \times four pixels in the Bayer pattern. The optimizations take advantage of redundancy between those cases to produce an efficient and compact implementation. To keep the cases straight while reading I recommend frequently checking the tables that show the patterns and to the actual implementation.

2 GPU Details

Although I try to take the long view in choosing optimizations in the rest of the paper, this section presents some details of OpenGL and GPUs that are important for peak performance today.

GPUs can operate in two modes: general-purpose (GPGPU) and graphics (OpenGL). Although many of the optimizations discussed in this paper are relevant to modes, I chose to focus on graphics mode. A GPGPU programming language like NVIDIA's CUDA or AMD's CTM (or equivalently, targeting a general-purpose streaming architecture like Cell) exposes the GPU's underlying memory model and instruction set and disables fixed-function graphics hardware like blending and rasterization. One can demosaic multiple pixels simultaneously and avoid the overhead of fetching the same raw texture values multiple times. The implementation in this paper has such a low operation count that it is likely limited by memory bandwidth, so amortizing the fetch cost may increase performance by as much as a factor of four. When building a pure 2D image processing pipeline that operates solely on full images and a single architecture, GPGPU therefore is the preferred approach.

There are several advantages to using GLSL under OpenGL instead of a GPGPU approach for general computational photography applications. GLSL programs are portable, from cell phones to rackmounted-server GPUs to gaming consoles and across vendors, whereas GPGPU programs must be written for a specific machine architecture and vendor. GLSL programs can integrate video into the rendering pipeline without the latency of buffer copies. Two common examples of where that is important are correcting lens distortion by rendering a 2D textured mesh and mapping video texture onto a 3D object. The texture fetches have very good cache coherence, so there may only be a factor of two to be gained in GPGPU memory access, and that is for the full rectangle case. OpenGL will only demosaic the required pixels when operating on general shapes—such as a distorted, masked, cropped, or scaled-down region—so it can outperform GPGPU in those cases.

The performance numbers in this paper do not include the cost of transferring the raw video to the GPU. That process involves loading the raw input as a 1-channel `GL_LUMINANCE` texture. The minification and magnification filters should both be set to `GL_NEAREST` to allow indexing the texture like a 2D array, with no fixed-function texture filtering (it also conveniently avoids the cost of two run-time floor operations, since one does not have to sample exactly at the pixel center). When transferring textures to a GPU, be sure to disable auto-mipmap generation and use a `GL_NPOT` (non-power-of-two) texture to avoid the cost of rescaling the data in the driver. There are many methods for efficiently streaming texture data to a GPU.

While `glTexImage2D` works fine, on some GPUs it is faster to use shared memory or vendor-specific upload extensions.

GPU textures are addressed with floating-point values, where each ordinate is on the range $[0, 1]$. The pixels of a $w \times h$ image therefore have dimensions $1/w \times 1/h$ in texture space. I assume that $w, h, 1/w, 1/h$ are computed only once per texture and are passed in to the pixel shader as a constant.

Note that I compute texture coordinates in the pixel shader, which means using floating point registers to store what should ideally be integer pixel coordinates values. Fortunately, there is sufficient precision in the 23-bit mantissa of a 32-bit float for this for textures, which are limited to 8192^2 or fewer pixels on most hardware anyway.

A mask is a vector operation that selects a subset of elements, for example `V.xy` creates a 2-vector from the first two components of a 4-vector. GPUs support swizzling in the masks, which allows replication and order changing, e.g., `V.zxxw`. Although there is no compact GLSL syntax for it, most also allow individual components to be negated or replaced by 0 or 1 as part of loading a vector register into the ALU.

3 Baseline Implementation

This section describes the direct implementation of Malvar-He-Cutler demosaicing that many people would write given the original paper and basic knowledge of GLSL. To provide a useful baseline, it is straightforward but not completely naïve, e.g., since the kernels are known a priori, most people would unroll the 2D loop of the filter application and then realize that they did not need to fetch or process pixels that correspond to a filter coefficient of zero.

For a specific Bayer filter (e.g., the one with the red pixel at position $(0, 0)$), there are four kinds of Bayer pixels: $\{\text{even or odd row}\} \times \{\text{even or odd column}\}$. For each kind of pixel there are three color channels to reconstruct, so there are a total of 12 filter kernels. Malvar et al.'s kernels for all of these are shown in table 1, with the inset images taken directly from their paper.

Now consider these filter kernels in the context of the full set of four possible Bayer patterns. These are just 1-pixel phase shifts of the original. Let $r = (r_x \in \{0, 1\}, r_y \in \{0, 1\})$ be the pixel position of the first “red” pixel in the Bayer optical filter. Using an “alternating” phase variable

$$\alpha = \lfloor (u, v) * (w, h) + r \rfloor \bmod (2, 2), \tag{1}$$

the texel with coordinate (u, v) is in an even column with respect to the filter kernels iff $\alpha_x = 0$, and is in an even row iff $\alpha_y = 0$.

The shader uses two branches to handle the four possible cases of α . These branches are taken a different way at every adjacent pixel, which is the worst behavior for a GPU (or branch-predicting CPU).

In each, case there are thirteen texture fetches. All but the center one require computation of the texture coordinate of the neighboring texel by

$$(\Delta x, \Delta y) * (1/w, 1/h) + (u, v), \tag{2}$$

so there are 12 MADDs as part of the fetches. Once the raw values are in registers, actually applying the filter kernels requires an average¹ of 19 MADD operations. See table 3 for the total operation count and comparisons with the optimized implementation.

4 Symmetry Optimization

The redundancy in table 1 is an opportunity for optimization. The filter kernels have horizontal and vertical reflection symmetry. This groups the neighbors around pixel position (x, y) into five categories shown in table 2. All pixels within a “group” are designated by the same letter. They will be multiplied by the same coefficient, regardless of the kind of pixel or color channel. Thus, summing the pixels in the same group before weighing them by the coefficient can save several multiplications. The subscripts on the letters distinguish the different values within a group. That is necessary for the actual implementation (below, in equation eqn:P), but for understanding the optimization, the patterns of the letters is what is important.

¹ 17 at EE (i.e., even row, even col) and OO pixels, 21 at EO and OE pixels.


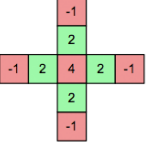
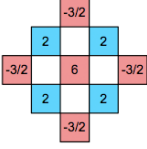
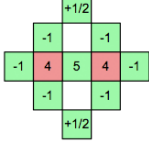

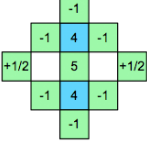
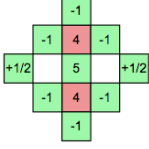

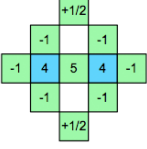
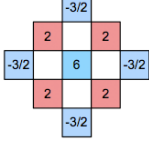
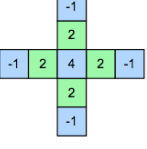

	R	G	B
EE: Even row, Even col	 Identity	 Cross	 Checker
EO: Even row, Odd col	 Θ	 Identity	 Φ
OE: Odd row, Even col	 Φ	 Identity	 Θ
OO: Odd row, Odd col	 Checker	 Cross	 Identity

Table 1: MHC filter kernels (each coefficient must be divided by 8 for normalization).

	$x-2$	$x-1$	x	$x+1$	$x+2$
$y-2$			A_0		
$y-1$		D_0	B_0	D_1	
y	E_0	F_0	C	F_1	E_1
$y+1$		D_2	B_1	D_3	
$y+2$			A_1		

Table 2: Patterns of matching filter coefficients relative to a source pixel at (x, y) .

Furthermore, there are only four unique kernels (plus the trivial identity) in table 1. Call these cross, checker, Θ , and Φ based on their shapes. By computing all four filters and storing the results in a 4-vector, regardless of the pixel shader can then select the appropriate using swizzles. This makes the entire shader into a single case and reduces the two branches to two conditional moves. Those moves are conditioned on the condition codes (zero or non-zero) of α and are likely to be as fast as simple register assignment.

Some of the coefficients are $\pm 1/8$. The MADDs associated with those can be reduced to ADDs at the added cost of a single multiplication by $1/8$ after applying the coefficients.

Both pixel offsets in equation 2 are on the small integer range $[-2, 2]$. Storing precomputed values $1/w, 1/h, 2/w, 2/h$ in a single 4-vector constant register eliminates the multiplication and thus reduces 12 MADDs to 12 ADDs.

These symmetry optimizations eliminate the branches. They actually increase the arithmetic (ALU) operation count from 32 to 36, but do so in order to reduce the number of MADDs from 32 to 13.

5 Vectorization

Many CPUs and GPUs are SIMD processors that can process array operations such as MADD and ADD on 4-vectors at the same speed as on scalars. On such processors, vectorizing an algorithm so that it operates on multiple scalars with a single instruction reduces the instruction count and increases performance by up to a factor of four.

Let the filter coefficients organized by pixel group be

$$\begin{aligned}
 k_A &= (-1.0, -1.5, -0.5, -1.0) \\
 k_B &= (2.0, 0.0, 0.0, 4.0) \\
 k_C &= (4.0, 6.0, 5.0, 5.0) \\
 k_D &= (0.0, 2.0, -1.0, -1.0) \\
 k_E &= (-1.0, -1.5, -1.0, 0.5) \\
 k_F &= (2.0, 0.0, 4.0, 0.0)
 \end{aligned} \tag{3}$$

and a pattern 4-vector of filter terms be:

$$P = \frac{1}{8} \left(k_A \sum_{i=0}^1 A_i + k_B \sum_{i=0}^1 B_i + k_C C + k_D \sum_{i=0}^3 D_i + k_E \sum_{i=0}^1 E_i + k_F \sum_{i=0}^1 F_i \right), \tag{4}$$

Depending on type of filter at a specific pixel and color channel, that channel’s demosaiced value is either C or one of the four elements of P . For a channel where the filter is the identity, the demosaiced value is just C . The cross, checker, Θ and Φ filters correspond to $P[0\dots3]$, respectively. For example, consider an Even-Even pixel in a Bayer pattern with the first red value at $(0,0)$. According to table 1, at this pixel, the filters to be applied to obtain the R, G, and B channels are identity, cross, and checker respective. Thus the demosaiced pixel’s RGB values are $(C, P[0], P[1])$.

The recent NVIDIA 8-and 9-series GPUs contain purely scalar ALUs. The final challenge in vectorizing is doing so in a way that does not hurt scalar GPUs like these by introducing needless multiplications by zero or computing redundant products. The code listed at the end of this paper uses masks to avoid introducing extra scalar computation while vectorizing.

6 Interpolator Optimization

GPUs have fixed-function interpolator units that linearly interpolate vectors (GLSL “varying” global variables) across the surface of primitive (triangle or quadrilateral). These units compute the texture coordinate at each pixel, for example. Because interpolators are built into the rasterizer as dedicated hardware, they can interpolate any vector quantity across the surface of an image at no additional per-pixel cost on most architectures.

Twelve of the ADD operations in the demosaicing shader are spent computing the offset coordinates for the texture fetches using equation 2 with precomputed products. Because each of the offsets varies linearly across the image, the interpolators can remove this cost. Since there are four rows and four columns of unique ordinates in addition to the center, the shader packs these into two 4-vectors. The vertex shader computes not only the texture coordinate for the center texel but a vector of x - and y - displaced texture coordinates. The pixel shader then uses combinations of these vectors as the texture fetch locations.

7 Analysis

The final algorithm requires 6 MADDs, 3 ADDs, one floor operation, and one modulo operation. The branches have been eliminated and the arithmetic operation count has been reduced by 2/3. In practice, the actual instruction counts may be slightly different at the hardware level. This is because most GPUs and CPUs translate even assembly code for their public interfaces into alternative instructions for their true (and rarely revealed) internal instruction sets. GPUs translate mainly at the driver level, while CPUs have the translators embedded in the processor itself. See table 5 for real-world results on one particular driver and GPU.

Table 4 shows measured demosaicing rate for eight methods on three GPUs. The measurement process was:

Implementation	Branches	CMOVE	MADD	ADD	Other ALU*	ALU Total
Direct	2	0	32	0	2	34
+ Symmetry	0	2	13	23	2	38
+ Vectorized	0	2	7	14	2	23
+ Interpolators	0	2	6	3	2	11

*All versions also perform 13 texture fetches, 1 modulo, and 1 floor operation that are inherent in demosaicing.

Table 3: Vector-4 operation count comparison after specific optimizations.

1. Create a raw $w \times h$ 8-bit luminance texture
2. Bind the appropriate shader and raw texture
3. Prime the system
4. Repeat 20 times (“main loop”):
 - (a) Rendering 200 quads covering $w \times h$ pixels each to the back buffer
 - (b) Swap the front and back buffers

I primed the system by running the body of the main loop 10 times stabilize the system, specifically the shader cache and dynamically-clocked GPU. I measured the elapsed time t for the entire main loop using the processor’s cycle counter. Given this elapsed time, the demosaicing rate is $200 \times w \times h$ pix/frame * 100 frame/ t .

The None column is a luminance texture with no shader. This shows the maximum processing rate of the GPU. Label, Nearest, Bilinear, and Direct Malvar-He-Cutler are the demosaicing algorithms in order of increasing quality and implementation complexity. The “+” columns represent the performance after the three classes of optimizations described in this paper. These are cumulative moving to the right, so the +Interpolator column contains all optimizations. The final column shows the optimized implementation performance relative to the baseline. Not shown in the table, I also tested on a SLI-3 machine with three NVIDIA GeForce GTX 280 GPUs, to verify that optimizations still functioned in the presence of multiple GPUs. They did and the net speedup was 134% with a +Interp throughput of 3271 MPix/s for that particular machine.

	None	Label	Nearest	Bilinear	Baseline	+Sym.	+Vector	+Interp.	Interp/Base
640 × 480 Pixels									
Radeon X1600	750	113	110	80	40	66	75	74	185%
Radeon X1800	4424	541	200	110	88	185	176	190	215%
Radeon HD 4870	11542	11581	6891	3022	1957	2102	2109	2078	106%
GeForce 8600M GT	1072	693	581	171	166	260	251	323	195%
GeForce Go 7800 GTX	4369	1277	995	293	83	308	308	336	404%
GeForce 9800 GX2	8148	6650	5102	2915	1273	2102	2242	2743	216%
1920 × 1080 Pixels									
Radeon X1600	330	58	110	79	58	65	73	73	126%
Radeon X1800	4507	529	199	109	88	184	176	189	215%
Radeon HD 4870	11769	11774	6963	3040	1961	2112	2115	2086	106%
GeForce 8600M GT	1081	695	583	171	166	259	250	324	195%
GeForce Go 7800 GTX	4683	1288	1002	289	93	311	310	337	363%
GeForce 9800 GX2	8188	5103	5103	3055	1209	2109	1947	2728	226%

Table 4: Performance in Mpix/s of demosaic algorithm and implementation variations on several GPU architectures by AMD (Radeon) and NVIDIA (GeForce). Higher is better.

The performance on the GeForce 9800 GX2 is what one would expect from the instruction counts. This is a desktop gaming GPU containing two physical processor microchips. Here, the significant reduction of MADD operations from the symmetry optimizations improves performance by about 170%, and subsequent changes create incremental improvements build up to a net speedup of approximately 220%. These gains are largely independent of resolution. This GPU has only scalar ALUs, so vectorizing has little impact: it seems to give a slight boost at low resolution, possibly from reduced program size improving cache

performance, and detract slightly at high resolution where the vectorization may make other driver-level optimizations more difficult.

The Radeon 4870 has tremendous processing power compared to its memory bandwidth. It is a VLIW architecture where each ALU performs five scalar operations per instruction, which for the purpose of this shader is equivalent to a vector processor. As a result of the high computation capability, the GPU is memory bound and these optimizations have little impact. Analyzing the shaders under the AMD GPU Shader Analyzer (<http://developer.amd.com/gpu/shader/Pages/default.aspx>) shows that the changes are indeed reducing the ALU load, as shown in table 5. Therefore, in the typical usage case where the demosaicing is followed by additional computation within the same shader, like color correction or segmentation, the optimizations made here would produce a net performance increase on such a card. The seemingly low ALU utilization for the Baseline shader is due to the large dynamic branches. Unlike the optimized shaders, this does not mean that the GPU is 49% available for more computation; instead, most of a threadgroup is stalled on an incoherent branch 49% of the time.

Note that in table 5 the symmetry optimizations more than halve the operation count, yet vectorization actually increases the count slightly. This is because the driver for the 4870 correctly makes many of the vectorization optimizations independently, and because it is targeting a single GPU architecture is in fact able to make some additional vectorization changes that would not benefit other architectures. Here, our manual vectorization improves register allocation and performance for a (good) net decrease in ALU Utilization, yet is likely a few percent short of optimal for the specific architecture since it increases ALU operation count.

	Baseline	+Symmetry	+Vectorize	+Interpolators
Registers	18	12	10	11
Fetches	13	13	13	13
ALU Ops	56	23	27	22
Interpolators	2	2	2	4
ALU Utilization	51%	89%	84%	79%

Table 5: GPU Shader Analyzer results for the Radeon HD 4870.

After the interpolator optimization, the shader is bandwidth-bound even on the lower-end GPUs, so further performance increases must come from reducing memory load and not computation. It is tempting to try and reduce the memory load by storing the raw values in an RGBA texture, where every one pixel stores four horizontally adjacent raw values. This would halve the number of texture fetches required. Unfortunately, GPUs don't support relative addressing (or mod 2 via integer bitwise-AND), so actually extracting the values from a packed RGBA texture requires a large number of branches or conditional moves that cancel the savings on texture fetches. Another way to improve memory access is to pack four frames of video into the RGBA channels of a single image and write to multiple output buffers in the shader. When working with multiple cameras this is an attractive way to handle their streams simultaneously if the interlacing can be performed efficiently on the CPU side.

Acknowledgements

This project was supported by an equipment grant from NVIDIA Corporation and advice from David Luebke. I am also extremely grateful to Chris Oat of AMD for extended discussions about these shaders and for running the Radeon performance experiments. Many of the insights in the analysis section are due to him.

8 GLSL Implementation

8.1 Vertex Shader

```

/** (w,h,1/w,1/h) */
uniform vec4      sourceSize;

/** Pixel position of the first red pixel in the Bayer pattern.  [{0,1}, {0, 1}]*/
uniform vec2      firstRed;

/** .xy = Pixel being sampled in the fragment shader on the range [0, 1]
    .zw = ...on the range [0, sourceSize], offset by firstRed */
varying vec4      center;

```



```

/** center.x + (-2/w, -1/w, 1/w, 2/w); These are the x-positions of the adjacent pixels.*/
varying vec4      xCoord;

/** center.y + (-2/h, -1/h, 1/h, 2/h); These are the y-positions of the adjacent pixels.*/
varying vec4      yCoord;

void main(void) {

    center.xy = gl_MultiTexCoord0.xy;
    center.zw = gl_MultiTexCoord0.xy * sourceSize.xy + firstRed;

    vec2 invSize = sourceSize.zw;
    xCoord = center.x + vec4(-2.0 * invSize.x, -invSize.x, invSize.x, 2.0 * invSize.x);
    yCoord = center.y + vec4(-2.0 * invSize.y, -invSize.y, invSize.y, 2.0 * invSize.y);

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

```

8.2 Pixel Shader

```

/** Monochrome RGBA or GL_LUMINANCE Bayer encoded texture.*/
uniform sampler2D      source;

varying vec4      center;
varying vec4      yCoord;
varying vec4      xCoord;

void main(void) {

    #define fetch(x, y) texture2D(source, vec2(x, y)).r

    float C = texture2D(source, center.xy).r; // ( 0, 0)
    const vec4 kC = vec4( 4.0,  6.0,  5.0,  5.0) / 8.0;

    // Determine which of four types of pixels we are on.
    vec2 alternate = mod(floor(center.zw), 2.0);

    vec4 Dvec = vec4(
        fetch(xCoord[1], yCoord[1]), // (-1,-1)
        fetch(xCoord[1], yCoord[2]), // (-1, 1)
        fetch(xCoord[2], yCoord[1]), // ( 1,-1)
        fetch(xCoord[2], yCoord[2])); // ( 1, 1)

    vec4 PATTERN = (kC.xyz * C).xyzz;

    // Can also be a dot product with (1,1,1,1) on hardware where that is
    // specially optimized.
    // Equivalent to: D = Dvec[0] + Dvec[1] + Dvec[2] + Dvec[3];
    Dvec.xy += Dvec.zw;
    Dvec.x  += Dvec.y;

    vec4 value = vec4(
        fetch(center.x, yCoord[0]), // ( 0,-2)
        fetch(center.x, yCoord[1]), // ( 0,-1)
        fetch(xCoord[0], center.y), // ( 1, 0)
        fetch(xCoord[1], center.y)); // ( 2, 0)

    vec4 temp = vec4(
        fetch(center.x, yCoord[3]), // ( 0, 2)
        fetch(center.x, yCoord[2]), // ( 0, 1)
        fetch(xCoord[3], center.y), // ( 2, 0)
        fetch(xCoord[2], center.y)); // ( 1, 0)

    // Even the simplest compilers should be able to constant-fold these to avoid the division.
    // Note that on scalar processors these constants force computation of some identical products twice.
    const vec4 kA = vec4(-1.0, -1.5,  0.5, -1.0) / 8.0;

```

```

const vec4 kB = vec4( 2.0,  0.0,  0.0,  4.0) / 8.0;
const vec4 kD = vec4( 0.0,  2.0, -1.0, -1.0) / 8.0;

// Conserve constant registers and take advantage of free swizzle on load
#define kE (kA.xyzw)
#define kF (kB.xyzw)

value += temp;

// There are five filter patterns (identity, cross, checker,
// theta, phi). Precompute the terms from all of them and then
// use swizzles to assign to color channels.
//
// Channel  Matches
// x      cross  (e.g., EE G)
// y      checker (e.g., EE B)
// z      theta  (e.g., EO R)
// w      phi    (e.g., EO R)

#define A (value[0])
#define B (value[1])
#define D (Dvec.x)
#define E (value[2])
#define F (value[3])

// Avoid zero elements. On a scalar processor this saves two MADDs and it has no
// effect on a vector processor.
PATTERN.yzw += (kD.yz * D).xyy;

PATTERN += (kA.xyz * A).xyzx + (kE.xyw * E).xyzx;
PATTERN.xw += kB.xw * B;
PATTERN.xz += kF.xz * F;

gl_FragColor.rgb = (alternate.y == 0.0) ?
    ((alternate.x == 0.0) ?
        vec3(C, PATTERN.xy) :
        vec3(PATTERN.z, C, PATTERN.w)) :
    ((alternate.x == 0.0) ?
        vec3(PATTERN.w, C, PATTERN.z) :
        vec3(PATTERN.yx, C));
}

```